FOREIGN KEY (*column-name***,...)**
Defines a referential constraint with the specified *constraint-name*.

Let T1 denote the object table of the statement. The foreign key of the referential constraint is composed of the identified columns. Each name in the list of column names must identify a column of T1 and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their stored lengths must not exceed 1024 (refer to "Byte Counts" on page 757 for the stored lengths). No LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on one of these types, or structured type column may be used as part of a foreign key (SQLSTATE 42962). There must be the same number of foreign key columns as there are in the parent key and the data types of the corresponding columns must be compatible (SQLSTATE 42830). Two column descriptions are compatible if they have compatible data types (both columns are numeric, character strings, graphic, date/time, or have the same distinct type).

references-clause
Specifies the parent table and parent key for the referential constraint.

REFERENCES *table-name*
The table specified in a REFERENCES clause must identify a base table that is described in the catalog, but must not identify a catalog table.

A referential constraint is a duplicate if its foreign key, parent key, and parent table are the same as the foreign key, parent key and parent table of a previously specified referential constraint. Duplicate referential constraints are ignored and a warning is issued (SQLSTATE 01543).

In the following discussion, let T2 denote the identified parent table and let T1 denote the table being created[82](T1 and T2 may be the same table).

The specified foreign key must have the same number of columns as the parent key of T2 and the description of the *n*th column of the foreign key must be comparable to the description of the *n*th column of that parent key. Datetime columns are not considered to be comparable to string columns for the purposes of this rule.

(*column-name***,...)**
The parent key of a referential constraint is composed of the identified columns. Each *column-name* must be an unqualified

82. or altered, in the case where this clause is referenced from the description of the ALTER TABLE statement.

name that identifies a column of T2. The same column must not be identified more than once.

The list of column names must match the set of columns (in any order) of the primary key or a unique constraint that exists on T2 (SQLSTATE 42890). If a column name list is not specified, then T2 must have a primary key (SQLSTATE 42888). Omission of the column name list is an implicit specification of the columns of that primary key in the sequence originally specified.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

rule-clause
Specifies what action to take on dependent tables.

ON DELETE
Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are four possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

SET NULL must not be specified unless some column of the foreign key allows null values. Omission of the clause is an implicit specification of ON DELETE NO ACTION.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the new relationship would form a cycle and T2 is already delete connected to T1, then

the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. The NO ACTION and RESTRICT actions are treated identically. Thus, if T1 is a dependent of T3 in a relationship with a delete rule of *r*, the referential constraint cannot be defined when *r* is SET NULL if any of these conditions exist:

- T2 and T3 are the same table
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3.

If *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

In applying the above rules to referential constraints, in which either the parent table or the dependent table is a member of a typed table hierarchy, all the referential constraints that apply to any table in the respective hierarchies are taken into consideration.

ON UPDATE
Specifies what action is to take place on the dependent tables when a row of the parent table is updated. The clause is optional. ON UPDATE NO ACTION is the default and ON UPDATE RESTRICT is the only alternative.

The difference between NO ACTION and RESTRICT is described under CREATE TABLE in "Notes" on page 753.

check-constraint
Defines a check constraint. A *check-constraint* is a *search-condition* that must evaluate to not false.

CONSTRAINT *constraint-name*
Names the check constraint. See page 734.

CHECK (*check-condition***)**

Defines a check constraint. A *check-condition* is a *search-condition* except as follows:

- A column reference must be to a column of the table being created
- The *search-condition* cannot contain a TYPE predicate
- It cannot contain any of the following (SQLSTATE 42621):
  - subqueries
  - dereference operations or DEREF functions where the scoped reference argument is other than the object identifier (OID) column.
  - CAST specifications with a SCOPE clause
  - column functions
  - functions that are not deterministic
  - functions defined to have an external action
  - user-defined functions using the SCRATCHPAD option
  - user-defined functions using the READS SQL DATA option
  - host variables
  - parameter markers
  - special registers
  - an alias
  - references to generated columns other than the identity column

If a check constraint is specified as part of a *column-definition* then a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns previously defined in the CREATE TABLE statement. Check constraints are not checked for inconsistencies, duplicate conditions or equivalent conditions. Therefore, contradictory or redundant check constraints can be defined resulting in possible errors at execution time.

The check-condition ″IS NOT NULL″ can be specified, however it is recommended that nullability be enforced directly using the NOT NULL attribute of a column. For example, CHECK (salary + bonus > 30000) is accepted if salary is set to NULL, because CHECK constraints must be either satisfied or unknown and in this case salary is unknown. However, CHECK (salary IS NOT NULL) would be considered false and a violation of the constraint if salary is set to NULL.

Check constraints are enforced when rows in the table are inserted or updated. A check constraint defined on a table automatically applies to all subtables of that table.

Rules

- The sum of the byte counts of the columns, including the inline lengths of all structured type columns, must not be greater than the row size limit that is based on the page size of the table space (SQLSTATE 54010). Refer to "Byte Counts" on page 757 and Table 33 on page 1102 for more information. For typed tables, the byte count is applied to the columns of the root table of the table hierarchy and every additional column introduced by every subtable in the table hierarchy (additional subtable columns must be considered nullable for byte count purposes, even when defined as not nullable). There is also an additional 4 bytes of overhead to identify the subtable to which each row belongs.
- The number of columns in a table cannot exceed 1 012 (SQLSTATE 54011). For typed tables, the total number of attributes of the types of all of the subtables in the table hierarchy cannot exceed 1010.
- An object identifier column of a typed table cannot be updated (SQLSTATE 42808).
- A partitioning key column of a table cannot be updated (SQLSTATE 42997).
- Any unique or primary key constraint defined on the table must be a superset of the partitioning key (SQLSTATE 42997).
- A nullable column of a partitioning key cannot be included as a foreign key column when the relationship is defined with ON DELETE SET NULL (SQLSTATE 42997).
- The following table provides the supported combinations of DATALINK options in the *file-link-options* (SQLSTATE 42613).

Table 22. Valid DATALINK File Control Option Combinations

INTEGRITY	READ PERMISSION	WRITE PERMISSION	RECOVERY	ON UNLINK
ALL	FS	FS	NO	Not applicable
ALL	FS	BLOCKED	NO	RESTORE
ALL	FS	BLOCKED	YES	RESTORE
ALL	DB	BLOCKED	NO	RESTORE
ALL	DB	BLOCKED	NO	DELETE
ALL	DB	BLOCKED	YES	RESTORE
ALL	DB	BLOCKED	YES	DELETE

The following rules only apply to partitioned databases.

- Tables composed only of columns with types LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on one of these types, or structured type can only be created in table spaces defined on single-partition nodegroups.
- The partitioning key definition of a table in a table space defined on a multiple partition nodegroup cannot be altered.
- The partitioning key column of a typed table must be the OID column.

Notes

- Creating a table with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- If a foreign key is specified:
  - All packages with a delete usage on the parent table are invalidated.
  - All packages with an update usage on at least one column in the parent key are invalidated.
- Creating a subtable causes invalidation of all packages that depend on any table in table hierarchy.
- VARCHAR and VARGRAPHIC columns that are greater than 4 000 and 2 000 respectively should not be used as input parameters in functions in SYSFUN schema. Errors will occur when the function is invoked with an argument value that exceeds these lengths (SQLSTATE 22001).
- The use of NO ACTION or RESTRICT as delete or update rules for referential constraints determines when the constraint is enforced. A delete or update rule of RESTRICT is enforced before all other constraints including those referential constraints with modifying rules such as CASCADE or SET NULL. A delete or update rule of NO ACTION is enforced after other referential constraints. There are very few cases where this can make a difference during a delete or update. One example where different behavior is evident involves a DELETE of rows in a view that is defined as a UNION ALL of related tables.

```
Table T1 is a parent of table T3, delete rule as noted below
Table T2 is a parent of table T3, delete rule CASCADE

CREATE VIEW V1 AS SELECT * FROM T1 UNION ALL SELECT * FROM T2

DELETE FROM V1
```

  If table T1 is a parent of table T3 with delete rule of RESTRICT, a restrict violation will be raised (SQLSTATE 23001) if there are any child rows for parent keys of T1 in T3.

  If table T1 is a parent of table T3 with delete rule of NO ACTION, the child rows may be deleted by the delete rule of CASCADE when deleting rows

from T2 before the NO ACTION delete rule is enforced for the deletes from T1. If deletes from T2 did not result in deleting all child rows for parent keys of T1 in T3, then a constraint violation will be raised (SQLSTATE 23504).

Note that the SQLSTATE returned is different depending on whether the delete or update rule is RESTRICT or NO ACTION.

- For tables in table spaces defined on multiple partition nodegroups, table collocation should be considered in choosing the partitioning keys. Following is a list of items to consider:
  - The tables must be in the same nodegroup for collocation. The table spaces may be different, but must be defined in the same nodegroup.
  - The partitioning keys of the tables must have the same number of columns, and the corresponding key columns must be partition compatible for collocation. For more information, see "Partition Compatibility" on page 114.
  - The choice of partitioning key also has an impact on performance of joins.If a table is frequently joined with another table, you should consider the joining column(s) as a partitioning key for both tables.
- The NOT LOGGED INITIALLY clause can not be used when DATALINK columns with the FILE LINK CONTROL attribute are present in the table (SQLSTATE 42613) .
- The NOT LOGGED INITIALLY option is useful for situations where a large result set needs to be created with data from an alternate source (another table or a file) and recovery of the table is not necessary. Using this option will save the overhead of logging the data. The following considerations apply when this option is specified:
  - When the unit of work is committed, all changes that were made to the table during the unit of work are flushed to disk.
  - When you run the Rollforward utility and it encounters a log record that indicates that a table in the database was either populated by the Load utility or created with the NOT LOGGED INITIALLY option, the table will be marked as unavailable. The table will be dropped by the Rollforward utility if it later encounters a DROP TABLE log. Otherwise, after the database is recovered, an error will be issued if any attempt is made to access the table (SQLSTATE 55019). The only operation permitted is to drop the table.
  - Once such a table is backed up as part of a database or table space back up, recovery of the table becomes possible.
- A REFRESH DEFERRED summary table defined with ENABLE QUERY OPTIMIZATION may be used to optimize the processing of queries if CURRENT REFRESH AGE is set to ANY. A REFRESH IMMEDIATE summary table defined with ENABLE QUERY OPTIMIZATION is always

considered for optimization. In order for this optimization be able to use a REFRESH DEFERRED or REFRESH IMMEDIATE summary table, the fullselect must conform to certain rules in addition to those already described. The fullselect must:

- be a subselect with a GROUP BY clause or a subselect with a single table reference
- not include DISTINCT anywhere in the select list
- not include any special registers
- not include functions that are not deterministic.

If the query specified when creating a summary table does not conform to these rules, a warning is returned (SQLSTATE 01633).

- If a summary table is defined with REFRESH IMMEDIATE, it is possible for an error to occur when attempting to apply the change resulting from an insert, update or delete of an underlying table. The error will cause the failure of the insert, update or delete of the underlying table.
- A referential constraint may be defined in such a way that either the parent table or the dependent table is a part of a table hierarchy. In such a case, the effect of the referential constraint is as follows:
  1. Effects of INSERT, UPDATE, and DELETE statements:
     - If a referential constraint exists, in which PT is a parent table and DT is a dependent table, the constraint ensures that for each row of DT (or any of its subtables) that has a non-null foreign key, a row exists in PT (or one of its subtables) with a matching parent key. This rule is enforced against any action that affects a row of PT or DT, regardless of how that action is initiated.
  2. Effects of DROP TABLE statements:
     - for referential constraints in which the dropped table is the parent table or dependent table, the constraint is dropped
     - for referential constraints in which a supertable of the dropped table is the parent table the rows of the dropped table are considered to be deleted from the supertable. The referential constraint is checked and its delete rule is invoked for each of the deleted rows.
     - for referential constraints in which a supertable of the dropped table is the dependent table, the constraint is not checked. Deletion of a row from a dependent table cannot result in violation of a referential constraint.
- ***Inoperative summary tables:*** An inoperative summary table is a table that is no longer available for SQL statements. A summary table becomes inoperative if:
  - A privilege upon which the summary table definition is dependent is revoked.

– An object such as a table, alias or function, upon which the summary table definition is dependent is dropped.

In practical terms, an inoperative summary table is one in which the summary table definition has been unintentionally dropped. For example, when an alias is dropped, any summary table defined using that alias is made inoperative. All packages dependent on the summary table are no longer valid.

Until the inoperative summary table is explicitly recreated or dropped, a statement using that inoperative summary table cannot be compiled (SQLSTATE 51024) with the exception of the CREATE ALIAS, CREATE TABLE, DROP TABLE, and COMMENT ON TABLE statements. Until the inoperative summary table has been explicitly dropped, its qualified name cannot be used to create another view, base table or alias. (SQLSTATE 42710).

An inoperative summary table may be recreated by issuing a CREATE TABLE statement using the definition text of the inoperative summary table. This summary table query text is stored in the TEXT column of the SYSCAT.VIEWS catalog. When recreating an inoperative summary table, it is necessary to explicitly grant any privileges required on that table by others, due to the fact that all authorization records on a summary table are deleted if the summary table is marked inoperative. Note that there is no need to explicitly drop the inoperative summary table in order to recreate it. Issuing a CREATE TABLE statement that defines a summary table with the same *table-name* as an inoperative summary table will cause that inoperative summary table to be replaced, and the CREATE TABLE statement will return a warning (SQLSTATE 01595).

Inoperative summary tables are indicated by an X in the VALID column of the SYSCAT.VIEWS catalog view and an X in the STATUS column of the SYSCAT.TABLES catalog view.

- ***Privileges:*** When any table is created, the definer of the table is granted CONTROL privilege. When a subtable is created, the SELECT privilege that each user or group has on the immediate supertable is automatically granted on the subtable with the table definer as the grantor.
- ***Row Size:*** The maximum number of bytes allowed in the row of a table is dependent on the page size of the table space in which the table is created (*tablspace-name1*). The following list shows the row size limit and number of columns limit associated with each table space page size.

Table 23. Limits for Number of Columns and Row Size in Each table space Page Size

Page Size	Row Size Limit	Column Count Limit
4K	4 005	500

Table 23. Limits for Number of Columns and Row Size in Each table space Page Size (continued)

Page Size	Row Size Limit	Column Count Limit
8K	8 101	1 012
16K	16 293	1 012
32K	32 677	1 012

The actual number of columns for a table may be further limited by the following formula:

– Total Columns * 8 + Number of LOB Columns * 12 + Number of Datalink Columns * 28 <= row size limit for page size.

- ***Byte Counts:*** The following list contains the byte counts of columns by data type for columns that do not allow null values. For a column that allows null values the byte count is one more than shown in the list.

  If the table is created based on a structured type, an additional 4 bytes of overhead is reserved to identify rows of subtables regardless of whether or not subtables are defined. Also, additional subtable columns must be considered nullable for byte count purposes, even when defined as not nullable.

Data type	Byte count
INTEGER	4
SMALLINT	2
BIGINT	8
REAL	4
DOUBLE	8
DECIMAL	The integral part of (*p*/2)+1, where *p* is the precision.
CHAR(*n*)	*n*
VARCHAR(*n*)	*n*+4
LONG VARCHAR	24
GRAPHIC(*n*)	*n**2
VARGRAPHIC(*n*)	(*n**2)+4
LONG VARGRAPHIC	24
DATE	4
TIME	3

TIMESTAMP 10

DATALINK(*n*) *n*+54

LOB types Each LOB value has a *LOB descriptor* in the base record that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the column. The following table shows typical sizes:

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 The INLINE LENGTH + 4. The INLINE LENGTH is the value specified (or implicitly calculated) for the column in the *column-options* clause.

Examples

Example 1: Create table TDEPT in the DEPARTX table space. DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT are column names. CHAR means the column will contain character data. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. The primary key consists of the column DEPTNO.

```
CREATE TABLE TDEPT
  (DEPTNO   CHAR(3)     NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   MGRNO    CHAR(6),
   ADMRDEPT CHAR(3)     NOT NULL,
   PRIMARY KEY(DEPTNO))
IN DEPARTX
```

Example 2: Create table PROJ in the SCHED table space. PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF, PRSTDATE, PRENDATE, and MAJPROJ are column names. CHAR means the column will contain character

data. DECIMAL means the column will contain packed decimal data. 5,2 means the following: 5 indicates the number of decimal digits, and 2 indicates the number of digits to the right of the decimal point. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. DATE means the column will contain date information in a three-part format (year, month, and day).

```
CREATE TABLE PROJ
  (PROJNO   CHAR(6)      NOT NULL,
   PROJNAME VARCHAR(24)  NOT NULL,
   DEPTNO   CHAR(3)      NOT NULL,
   RESPEMP  CHAR(6)      NOT NULL,
   PRSTAFF  DECIMAL(5,2)         ,
   PRSTDATE DATE                 ,
   PRENDATE DATE                 ,
   MAJPROJ  CHAR(6)      NOT NULL)
IN SCHED
```

Example 3: Create a table called EMPLOYEE_SALARY where any unknown salary is considered 0. No table space is specified, so that the table will be created in a table space selected by the system based on the rules descirbed for the *IN tablespace-name1* clause.

```
CREATE TABLE EMPLOYEE_SALARY
  (DEPTNO   CHAR(3)      NOT NULL,
   DEPTNAME VARCHAR(36)  NOT NULL,
   EMPNO    CHAR(6)      NOT NULL,
   SALARY   DECIMAL(9,2) NOT NULL WITH DEFAULT)
```

Example 4: Create distinct types for total salary and miles and use them for columns of a table created in the default table space. In a dynamic SQL statement assume the CURRENT SCHEMA special register is JOHNDOE and the CURRENT PATH is the default (″SYSIBM″,″SYSFUN″,″JOHNDOE″).

If a value for SALARY is not specified it must be set to 0 and if a value for LIVING_DIST is not specified it must to set to 1 mile.

```
CREATE DISTINCT TYPE JOHNDOE.T_SALARY AS INTEGER WITH COMPARISONS

CREATE DISTINCT TYPE JOHNDOE.MILES AS FLOAT WITH COMPARISONS

CREATE TABLE EMPLOYEE
   (ID          INTEGER NOT NULL,
    NAME        CHAR (30),
    SALARY      T_SALARY NOT NULL WITH DEFAULT,
    LIVING_DIST MILES    DEFAULT MILES(1) )
```

Example 5: Create distinct types for image and audio and use them for columns of a table. No table space is specified, so that the table will be created in a table space selected by the system based on the rules descirbed for the *IN tablespace-name1* clause. Assume the CURRENT PATH is the default.

```
CREATE DISTINCT TYPE IMAGE AS BLOB (10M)

CREATE DISTINCT TYPE AUDIO AS BLOB (1G)

CREATE TABLE PERSON
   (SSN    INTEGER NOT NULL,
    NAME   CHAR (30),
    VOICE  AUDIO,
    PHOTO  IMAGE)
```

Example 6: Create table EMPLOYEE in the HUMRES table space. The constraints defined on the table are the following:

- The values of department number must lie in the range 10 to 100.
- The job of an employee can only be either 'Sales', 'Mgr' or 'Clerk'.
- Every employee that has been with the company since 1986 must make more than $40,500.

Note: If the columns included in the check constraints are nullable they could also be NULL.

```
CREATE TABLE EMPLOYEE
(ID          SMALLINT NOT NULL,
 NAME        VARCHAR(9),
 DEPT        SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
 JOB         CHAR(5) CHECK (JOB IN ('Sales','Mgr','Clerk')),
 HIREDATE    DATE,
 SALARY      DECIMAL(7,2),
 COMM        DECIMAL(7,2),
 PRIMARY KEY (ID),
 CONSTRAINT  YEARSAL CHECK (YEAR(HIREDATE) > 1986 OR SALARY > 40500)
)
IN HUMRES
```

Example 7: Create a table that is wholly contained in the PAYROLL table space.

```
CREATE TABLE EMPLOYEE .....
   IN PAYROLL
```

Example 8: Create a table with its data part in ACCOUNTING and its index part in ACCOUNT_IDX.

```
CREATE TABLE SALARY.....
   IN ACCOUNTING INDEX IN ACCOUNT_IDX
```

Example 9: Create a table and log SQL changes in the default format.

```
CREATE TABLE SALARY1 .....
```

or

```
CREATE TABLE SALARY1 .....
   DATA CAPTURE NONE
```

Example 10: Create a table and log SQL changes in an expanded format.

```
CREATE TABLE SALARY2 .....
   DATA CAPTURE CHANGES
```

Example 11: Create a table EMP_ACT in the SCHED table space. EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, and EMENDATE are column names. Constraints defined on the table are:

- The value for the set of columns, EMPNO, PROJNO, and ACTNO, in any row must be unique.
- The value of PROJNO must match an existing value for the PROJNO column in the PROJECT table and if the project is deleted all rows referring to the project in EMP_ACT should also be deleted.

```
CREATE TABLE EMP_ACT
(EMPNO       CHAR(6) NOT NULL,
 PROJNO      CHAR(6) NOT NULL,
 ACTNO       SMALLINT NOT NULL,
 EMPTIME     DECIMAL(5,2),
 EMSTDATE    DATE,
 EMENDATE    DATE,
 CONSTRAINT EMP_ACT_UNIQ UNIQUE (EMPNO,PROJNO,ACTNO),
 CONSTRAINT FK_ACT_PROJ FOREIGN KEY (PROJNO)
                        REFERENCES PROJECT (PROJNO) ON DELETE CASCADE
)
IN SCHED
```

A unique index called EMP_ACT_UNIQ is automatically created in the same schema to enforce the unique constraint.

Example 12: Create a table that is to hold information about famous goals for the ice hockey hall of fame. The table will list information about the player who scored the goal, the goaltender against who it was scored, the date and place, and a description. When available, it will also point to places where newspaper articles about the game are stored and where still and moving pictures of the goal are stored. The newspaper articles are to be linked so they cannot be deleted or renamed but all existing display and update applications must continue to operate. The still pictures and movies are to be linked with access under complete control of DB2. The still pictures are to have recovery and are to be returned to their original owner if unlinked. The movie pictures are not to have recovery and are to be deleted if unlinked. The description column and the three DATALINK columns are nullable.

```
CREATE TABLE HOCKEY_GOALS
  ( BY_PLAYER      VARCHAR(30)   NOT NULL,
    BY_TEAM        VARCHAR(30)   NOT NULL,
    AGAINST_PLAYER VARCHAR(30)   NOT NULL,
    AGAINST_TEAM   VARCHAR(30)   NOT NULL,
    DATE_OF_GOAL   DATE          NOT NULL,
    DESCRIPTION    CLOB(5000),
    ARTICLES       DATALINK  LINKTYPE URL FILE LINK CONTROL MODE DB2OPTIONS,
```

```
    SNAPSHOT        DATALINK  LINKTYPE URL FILE LINK CONTROL
                              INTEGRITY ALL
                              READ PERMISSION DB WRITE PERMISSION BLOCKED
                              RECOVERY YES ON UNLINK RESTORE,
    MOVIE           DATALINK  LINKTYPE URL FILE LINK CONTROL
                              INTEGRITY ALL
                              READ PERMISSION DB WRITE PERMISSION BLOCKED
                              RECOVERY NO ON UNLINK DELETE )
```

Example 13: Suppose an exception table is needed for the EMPLOYEE table. One can be created using the following statement.

```
CREATE TABLE EXCEPTION_EMPLOYEE AS
  (SELECT EMPLOYEE.*,
     CURRENT TIMESTAMP AS TIMESTAMP,
     CAST ('' AS CLOB(32K)) AS MSG
   FROM EMPLOYEE
   ) DEFINITION ONLY
```

Example 14: Given the following table spaces with the indicated attributes:

```
TBSPACE            PAGESIZE    USER   USERAUTH
------------------ ----------- ------ --------
DEPT4K                    4096 BOBBY  Y
PUBLIC4K                  4096 PUBLIC Y
DEPT8K                    8192 BOBBY  Y
DEPT8K                    8192 RICK   Y
PUBLIC8K                  8192 PUBLIC Y
```

- If RICK creates the following table, it is placed in table space PUBLIC4K since the byte count is less than 4005; but if BOBBY creates the same table, it is placed in table space DEPT4K, since BOBBY has USE privilege because of an explicit grant:

  ```
  CREATE TABLE DOCUMENTS
     (SUMMARY    VARCHAR(1000),
      REPORT     VARCHAR(2000))
  ```

- If BOBBY creates the following table, it is placed in table space DEPT8K since the byte count is greater than 4005, and BOBBY has USE privilege because of an explicit grant. However, if DUNCAN creates the same table, it is placed in table space PUBLIC8K, since DUNCAN has no specific privileges:

  ```
  CREATE TABLE CURRICULUM
     (SUMMARY    VARCHAR(1000),
      REPORT     VARCHAR(2000),
      EXERCISES  VARCHAR(1500))
  ```

Example 15: Create a table with a LEAD column defined with the structured type EMP. Specify an INLINE LENGTH of 300 bytes for the LEAD column, indicating that any instances of LEAD that cannot fit within the 300 bytes are stored outside the table (separately from the base table row, similar to the way LOB values are handled).

```
CREATE TABLE PROJECTS (PID INTEGER,
  LEAD EMP INLINE LENGTH 300,
  STARTDATE DATE,
            ...)
```

Example 16: Create a table DEPT with five columns named DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, and LOCATION. Column DEPT is to be defined as an IDENTITY column such that DB2 will always generate a value for it. The values for the DEPT column should begin with 500 and increment by 1.

```
CREATE TABLE DEPT
   (DEPTNO SMALLINT NOT NULL
      GENERATED ALWAYS AS IDENTITY
      (START WITH 500, INCREMENT BY 1),
    DEPTNAME VARCHAR (36) NOT NULL,
    MGRNO CHAR(6),
    ADMRDEPT SMALLINT NOT NULL,
    LOCATION CHAR(30))
```

CREATE TABLESPACE

The CREATE TABLESPACE statement creates a new tablespace within the database, assigns containers to the tablespace, and records the tablespace definition and attributes in the catalog.

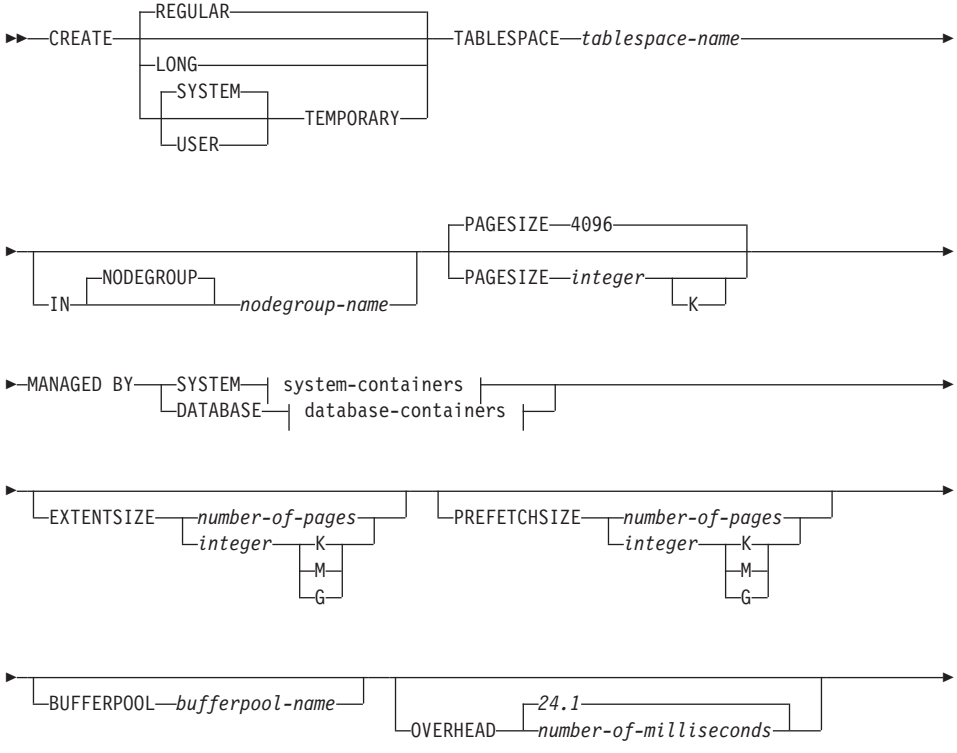
Invocation

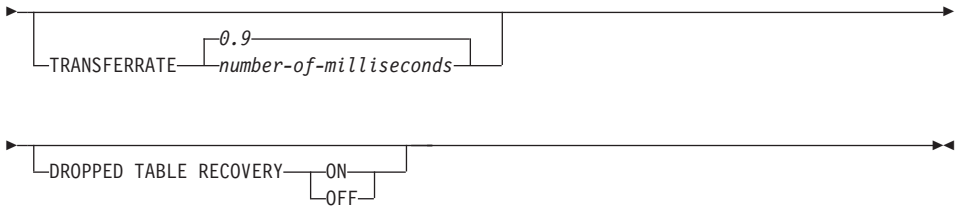
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

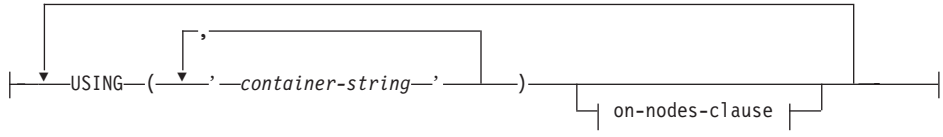
The authorization ID of the statement must have SYSCTRL or SYSADM authority.

Syntax

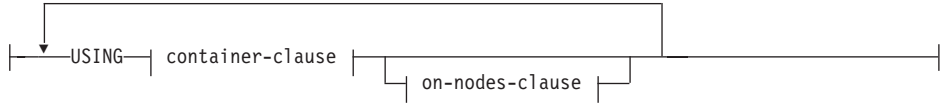




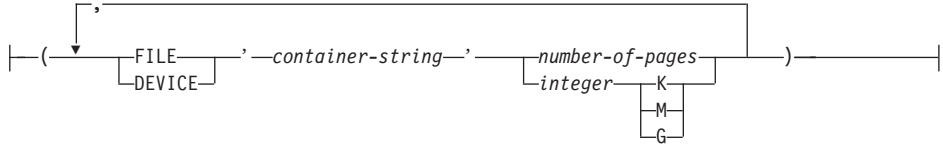
system-containers:



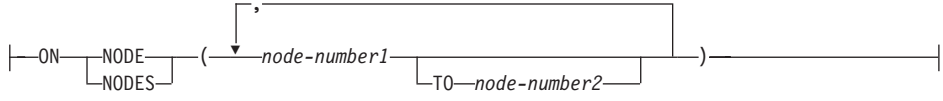
database-containers:



container-clause:



on-nodes-clause:



Description

REGULAR
Stores all data except for temporary tables.

LONG
Stores long or LOB table columns. It may also store structured type columns. The tablespace must be a DMS tablespace.

SYSTEM TEMPORARY
Stores temporary tables (work areas used by the database manager to perform operations such as sorts or joins). The keyword SYSTEM is optional. Note that a database must always have at least one SYSTEM TEMPORARY tablespace, as temporary tables can only be stored in such a tablespace. A temporary tablespace is created automatically when a database is created.

See CREATE DATABASE in the *Command Reference* for more information.

USER TEMPORARY
Stores declared global temporary tables. Note that no user temporary tablespaces exist when a database is created. At least one user temporary tablespace should be created with appropriate USE privileges, to allow definition of declared temporary tables.

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

IN NODEGROUP *nodegroup-name*
Specifies the nodegroup for the tablespace. The nodegroup must exist. The only nodegroup that can be specified when creating a SYSTEM TEMPORARY tablespace is IBMTEMPGROUP. The NODEGROUP keyword is optional.

If the nodegroup is not specified, the default nodegroup (IBMDEFAULTGROUP) is used for REGULAR, LONG and USER TEMPORARY tablespaces. For SYSTEM TEMPORARY tablespaces, the default nodegroup IBMTEMPGROUP is used.

PAGESIZE *integer* **[K]**
Defines the size of pages used for the tablespace. The valid values for *integer* without the suffix K are 4 096 or 8 192, 16 384, or 32 768. The valid values for *integer* with the suffix K are 4 or 8, 16, or 32. An error occurs if the page size is not one of these values (SQLSTATE 428DE) or the page size is not the same as the page size of the bufferpool associated with the tablespace (SQLSTATE 428CB). The default is 4 096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

MANAGED BY SYSTEM
Specifies that the tablespace is to be a system managed space (SMS) tablespace.

system-containers
Specify the containers for an SMS tablespace.

USING *('container-string',...)*
For a SMS tablespace, identifies one or more containers that will belong to the tablespace and into which the tablespace's data will be stored. The *container-string* cannot exceed 240 bytes in length.

Each *container-string* can be an absolute or relative directory name. The directory name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. When a tablespace is dropped, all components created by the database manager are deleted. If the directory identified by container-string exist, it must not contain any files or subdirectories (SQLSTATE 428B2).

The format of *container-string* is dependent on the operating system. The containers are specified in the normal manner for the operating system. For example, an OS/2 Windows 95 and Windows NT directory path begins with a drive letter and a ":", while on UNIX-based systems, a path begins with a "/".

Note that remote resources (such as LAN-redirected drives on OS/2, Windows 95 and Windows NT or NFS-mounted file systems on AIX) are not supported.

on-nodes-clause
Specifies the partition or partitions on which the containers are created in a partitioned database. If this clause is not specified, then the containers are created on the partitions in the nodegroup that are not explicitly specified in any other *on-nodes-clause*s. For a SYSTEM TEMPORARY tablespace defined on nodegroup IBMTEMPGROUP, when the *on-nodes-clause* is not specified, the containers will also be created on all new partitions or nodes added to the database. See page 769 for details on specifying this clause.

MANAGED BY DATABASE
Specifies that the tablespace is to be a database managed space (DMS) tablespace.

database-containers
Specify the containers for a DMS tablespace.

USING
Introduces a container-clause.

container-clause
Specifies the containers for a DMS tablespace.

(FILE | DEVICE *'container-string' number-of-pages,...)*
For a DMS tablespace, identifies one or more containers that will

belong to the tablespace and into which the tablespace's data will be stored. The type of the container (either FILE or DEVICE) and its size (in PAGESIZE pages) are specified. The size can also be specified as an integer value followed by K (for kilobytes), M (for megabytes) or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages for the container. A mixture of FILE and DEVICE containers can be specified. The *container-string* cannot exceed 254 bytes in length.

For a FILE container, the *container-string* must be an absolute or relative file name. The file name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. If the file does not exist, it will be created and initialized to the specified size by the database manager. When a tablespace is dropped, all components created by the database manager are deleted.

Note: If the file exists it is overwritten and if it is smaller than specified it is extended. The file will not be truncated if it is larger than specified.

For a DEVICE container, the *container-string* must be a device name. The device must already exist.

All containers must be unique across all databases; a container can belong to only one tablespace. The size of the containers can differ, however optimal performance is achieved when all containers are the same size. The exact format of *container-string* is dependent on the operating system. The containers will be specified in the normal manner for the operating system. For more detail on declaring containers, refer to the Administration Guide.

Remote resources (such as LAN-redirected drives on OS/2, Windows 95 and Windows NT or NFS-mounted file systems on AIX) are not supported.

on-nodes-clause

Specifies the partition or partitions on which the containers are created in a partitioned database. If this clause is not specified, then the containers are created on the partitions in the nodegroup that are not explicitly specified in any other *on-nodes-clause*. For a SYSTEM TEMPORARY tablespace defined on nodegroup IBMTEMPGROUP, when the *on-nodes-clause* is not specified, the containers will also be created on all new partitions added to the database. See page 769 for details on specifying this clause.

on-nodes-clause
Specifies the partitions on which containers are created in a partitioned database.

ON NODES
Keywords that indicate that specific partitions are specified. NODE is a synonym for NODES.

node-number1
Specify a specific partition (or node) number.

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

The partition specified by number and every partition (or node) in the range of partition must exist in the nodegroup on which the tablespace is defined (SQLSTATE 42729). A partition-number may only appear explicitly or within a range in exactly one *on-nodes-clause* for the statement (SQLSTATE 42613).

EXTENTSIZE *number-of-pages*
Specifies the number of PAGESIZE pages that will be written to a container before skipping to the next container. The extent size value can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for extent size. The database manager cycles repeatedly through the containers as data is stored.

The default value is provided by the DFT_EXTENT_SZ configuration parameter.

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

The default value is provided by the DFT_PREFETCH_SZ configuration parameter. (This configuration parameter, like all configuration parameters, is explained in detail in the *Administration Guide*.)

BUFFERPOOL *bufferpool-name*
The name of the buffer pool used for tables in this tablespace. The buffer pool must exist (SQLSTATE 42704). If not specified, the default buffer pool (IBMDEFAULTBP) is used. The page size of the bufferpool must match the page size specified (or defaulted) for the tablespace (SQLSTATE 428CB). The nodegroup of the tablespace must be defined for the bufferpool (SQLSTATE 42735).

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

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

DROPPED TABLE RECOVERY
Dropped tables in the specified tablespace may be recovered using the RECOVER TABLE ON option of the ROLLFORWARD command. This clause can only be specified for a REGULAR tablespace (SQLSTATE 42613). For more information on recovering dropped tables, refer to the *Administration Guide*.

Notes

- For information on how to determine the correct EXTENTSIZE, PREFETCHSIZE, OVERHEAD, and TRANSFERRATE values, refer to the *Administration Guide*.
- Choosing between a database-managed space or a system-managed space for a tablespace is a fundamental choice involving trade-offs. See the *Administration Guide* for a discussion of those trade-offs.
- When more than one TEMPORARY tablespace exists in the database, they will be used in round-robin fashion in order to balance their usage. See the *Administration Guide* for information on using more than one tablespace, rebalancing and recommended values for EXTENTSIZE, PREFETCHSIZE, OVERHEAD, and TRANSFERRATE.

- In a partitioned database if more than one partition resides on the same physical node, then the same device or specific path cannot be specified for such partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each partition or use a relative path name.
- You can specify a node expression for container string syntax when creating either SMS or DMS containers. You would typically specify the node expression if you are using multiple logical nodes in the partitioned database system. This ensures that container names are unique across nodes (database partition servers). When you specify the expression, either the node number is part of the container name, or, if you specify additional arguments, the result of the argument is part of the container name.

  You use the argument " $N" ([blank]$N) to indicate the node expression. The argument must occur at the end of the container string and can only be used in one of the following forms. In the table that follows, the node number is assumed to be 5:

Table 24. Arguments for Creating Containers

Syntax	Example	Value
[blank]$N	" $N"	5
[blank]$N+[number]	" $N+1011"	1016
[blank]$N%[number]	" $N%3"	2
[blank]$N+[number]%[number]	" $N+12%13"	4
[blank]$N%[number]+[number]	" $N%3+20"	22
Note: – % is modulus – In all cases, the operators are evaluated from left to right.		

Some examples are as follows:

Example 1:

```
CREATE TABLESPACE TS1 MANAGED BY DATABASE USING
   (device '/dev/rcont $N' 20000)
```

On a two-node system, the following containers would be used:

```
/dev/rcont0  - on NODE 0
/dev/rcont1  - on NODE 1
```

Example 2:

```
CREATE TABLESPACE TS2 MANAGED BY DATABASE USING
   (file '/DB2/containers/TS2/container $N+100' 10000)
```

On a four-node system, the following containers would be created:

```
/DB2/containers/TS2/container100  - on NODE 0
/DB2/containers/TS2/container101  - on NODE 1
/DB2/containers/TS2/container102  - on NODE 2
/DB2/containers/TS2/container103  - on NODE 3
```

Example 3:

```
CREATE TABLESPACE TS3 MANAGED BY SYSTEM USING
('/TS3/cont $N%2','/TS3/cont $N%2+2')
```

On a two-node system, the following containers would be created:

```
/TS3/cont0  - On NODE 0
/TS3/cont2  - On NODE 0
/TS3/cont1  - On NODE 1
/TS3/cont3  - On NODE 1
```

Examples

Example 1: Create a regular DMS table space on a UNIX-based system using 3 devices of 10 000 4K pages each. Specify their I/O characteristics.

```
CREATE TABLESPACE PAYROLL
   MANAGED BY DATABASE
   USING (DEVICE'/dev/rhdisk6' 10000,
      DEVICE '/dev/rhdisk7' 10000,
      DEVICE '/dev/rhdisk8' 10000)
   OVERHEAD 24.1
   TRANSFERRATE 0.9
```

Example 2: Create a regular SMS table space on OS/2 or Windows NT using 3 directories on three separate drives, with a 64-page extent size, and a 32-page prefetch size.

```
CREATE TABLESPACE ACCOUNTING
   MANAGED BY SYSTEM
   USING ('d:\acc_tbsp', 'e:\acc_tbsp', 'f:\acc_tbsp')
   EXTENTSIZE 64
   PREFETCHSIZE 32
```

Example 3: Create a temporary DMS table space on Unix using 2 files of 50,000 pages each, and a 256-page extent size.

```
CREATE TEMPORARY TABLESPACE TEMPSPACE2
   MANAGED BY DATABASE
   USING (FILE '/tmp/tempspace2.f1' 50000,
      FILE '/tmp/tempspace2.f2' 50000)
   EXTENTSIZE 256
```

Example 4: Create a DMS table space on nodegroup ODDNODEGROUP (nodes 1,3,5) on a Unix partitioned database. On all partitions (or nodes), use the device /dev/rhdisk0 for 10 000 4K pages. Also specify a partition specific device for each partition with 40 000 4K pages.

```
CREATE TABLESPACE PLANS
   MANAGED BY DATABASE
   USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn1hd01' 40000)
   ON NODE (1)
   USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn3hd03' 40000)
   ON NODE (3)
   USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn5hd05' 40000)
   ON NODE (5)
```

CREATE TRANSFORM

The CREATE TRANSFORM statement defines transformation functions, identified by a group name, that are used to exchange structured type values with host language programs and with external functions and methods.

Invocation

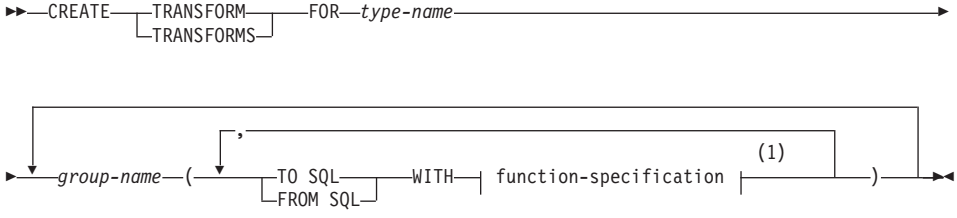
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

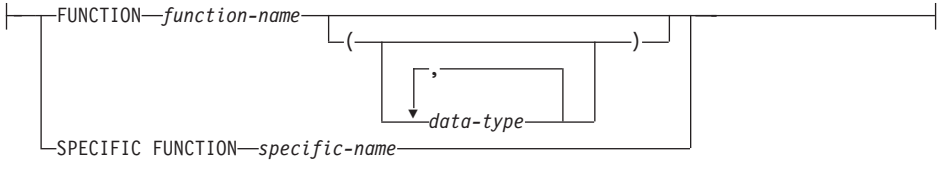
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- definer of the type identified by *type-name* and definer of every function specified.

Syntax



function-specification:



Notes:

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

Description

TRANSFORM or **TRANSFORMS**
Indicates that one or more transform groups is being defined. Either version of the keyword can be specified.

FOR *type-name*
Specifies a name for the user-defined structured type for which the transform group is being defined.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified *type-name*. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for an unqualified *type-name*. The *type-name* must be the name of an existing user-defined type (SQLSTATE 42704), and it must be a structured type (SQLSTATE 42809). The structured type or any other structured type in the same type hierarchy must not have transforms already defined with the given group-name (SQLSTATE 42739).

group-name
Specifies the name of the transform group containing the TO SQL and FROM SQL functions. The name does not need to be unique; but a transform group of this name (with the same TO SQL and/or FROM SQL direction defined) must not be previously defined for the specified *type-name* (SQLSTATE 42739). A *group-name* must be an SQL identifier, with a maximum of 18 characters in length (SQLSTATE 42622), and it may not include any qualifier prefix (SQLSTATE 42601). The *group-name* cannot begin with the prefix 'SYS', since this is reserved for database use (SQLSTATE 42939).

At most, one of each of the FROM SQL and TO SQL function designations may be specified for any given group (SQLSTATE 42628).

TO SQL
Defines the specific function used to transform a value to the SQL user-defined structured type format. The function must have all its parameters as built-in data types and the returned type is *type-name*.

FROM SQL
Defines the specific function used to transform a value to a built in data type value representing the SQL user-defined structured type. The function must have one parameter of data type *type-name*, and return a built-in data type (or set of built-in data types).

WITH *function-specification*
There are several ways available to specify the function instance.

If FROM SQL is specified, *function-specification* must identify a function that meets the following requirements:

- there is one parameter of type *type-name*
- the return type is a built-in type, or a row whose columns all have built-in types
- the signature specifies either LANGUAGE SQL or the use of another FROM SQL transform function which has LANGUAGE SQL.

If TO SQL is specified, *function-specification* must identify a function that meets the following requirements:

- all parameters have built-in types
- the return type is *type-name*
- the signature specifies either LANGUAGE SQL or the use of another TO SQL transform function which has LANGUAGE SQL.

Methods (even if specified with FUNCTION ACCESS) cannot be specified as transforms through *function-specification*. Instead, only functions that are defined by the CREATE FUNCTION statement can act as transforms (SQLSTATE 42704 or 42883).

Additionally, although not enforced, the one or more built-in types which are returned from the FROM SQL function should directly correspond to the one or more built-in types which are parameters of the TO SQL function. This is a logical consequence of the inverse relationship between these two functions.

FUNCTION *function-name*
Identifies the particular function by name, and is valid only if there is exactly one function with the *function-name*. The function 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 is raised (SQLSTATE 42704). If there is more than one specific instance of the function in the named or implied schema, an error is raised (SQLSTATE 42725). The standard function selection algorithm is not used.

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

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

(*data-type,...***)**
The data-types specified here must match the data types specified in the CREATE FUNCTION statement in the corresponding

position. Both the number of data types and the logical concatenation of the data types are used to identify the specific function.

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

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

FLOAT() cannot be used (SQLSTATE 42601), 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. For example, a CHAR FOR BIT DATA specified in the signature would match a function defined with CHAR only.

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

SPECIFIC FUNCTION *specific-name*
Identifies the particular user-defined function, using a specific name either specified or defaulted to at function creation time.

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

Notes

- When a transform group is not specified in an application program (using the TRANSFORM GROUP precompile or bind option for static SQL, or the SET CURRENT DEFAULT TRANSFORM GROUP statement for dynamic SQL), the transform functions in the transform group 'DB2_PROGRAM' are used (if defined) when the application program is retrieving or sending host variables that are based on the user-defined structured type identified by *type-name*. When retrieving a value of data type *type-name*, the FROM SQL transform is executed to transform the structured type to the built-in data type returned by the transform function. Similarly, when sending a

host variable that will be assigned to a value of data type *type-name*, the TO SQL transform is executed to transform the built-in data type value to the structured type value. If a user-defined transform group is not specified or a 'DB2_PROGRAM' group is not defined (for the given structured type), an error results.

- The built-in data type representation for a structured type host variable must be assignable:
  - from the result of the FROM SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command (using retrieval assignment rules) and
  - to the parameter of the TO SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command (using storage assignment rules).

  If a host variable is not assignment compatible with the type required by the applicable transform function, an error is raised (for bind-in: SQLSTATE 42821, for bind-out: SQLSTATE 42806). For errors that result from string assignments, see "String Assignments" on page 96.
- The transform functions identified in the default transform group named 'DB2_FUNCTION' are used whenever a user-defined function not written in SQL is invoked using the data type *type-name* as a parameter or returns type. This applies when the function or method does not specify the TRANSFORM GROUP clause. When invoking the function with an argument of data type *type-name*, the FROM SQL transform is executed to transform the structured type to the built-in data type returned by the transform function. Similarly, when the returns data type of the function is of data type *type-name*, the TO SQL transform is executed to transform the built-in data type value returned from the external function program into the structured type value.
- If a structured type contains an attribute which is also a structured type, the associated transform functions must recursively expand (or assemble) all nested structured types. This means that the results or parameters of the transform functions consist only of the set of built-in types representing all base attributes of the subject structured type (including all its nested structured types). There is no "cascading" of transform functions for handling nested structured types.
- The function (or functions) identified in this statement are resolved according to the rules outlined above at the execution of this statement. When these functions are used (implicitly) in subsequent SQL statements, they do not undergo another resolution process. The transform functions defined in this statement are recorded exactly as they are resolved in this statement.
- When attributes or subtypes of a given type are created or dropped, the transform functions for the user-defined structured type must also be changed.

- For a given transform group, the FROM SQL and TO SQL functions can be specified in either the same *group-name* clause, in separate *group-name* clauses, or in separate CREATE TRANSFORM statements. The only restriction is that a given FROM SQL or TO SQL function designation may not be redefined without first dropping the existing group definition. This allows you to define, for example, a FROM SQL transform function for a given group first, and the corresponding TO SQL transform function for the same group at a later time.

Examples

Example 1: Create two transform groups that associate the user-defined structured type polygon with a transform function customized for C and one specialized for Java.

```
CREATE TRANSFORM FOR POLYGON
   mystruct1 (FROM SQL WITH FUNCTION myxform_sqlstruct,
              TO SQL WITH FUNCTION myxform_structsql)
   myjava1   (FROM SQL WITH FUNCTION myxform_sqljava,
              TO SQL WITH FUNCTION myxform_javasql )
```