CHAPTER 9

SQL Statements

About this chapter

This chapter presents detailed descriptions of the SQL statements that are available to users of Adaptive Server Anywhere.

This chapter contains an alphabetical listing of SQL statements, including some that can only be used from Embedded SQL or Interactive SQL.

Contents

The chapter includes an alphabetical list of SQL statements.

Using the SQL statement reference

This section describes some conventions used in documenting the SQL statements.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords are shown in UPPER CASE. However, SQL keywords are case insensitive, so you can enter keywords in any case you wish; SELECT is the same as Select, which is the same as select.
- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown in *italics*.
- **Continuation** Lines beginning with an ellipsis (...) are a continuation from the previous line.
- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots). One or more list elements are allowed. If more than one is specified, they must be separated by commas.
- **Optional portions** Optional portions of a statement are enclosed by square brackets. For example, ...

  ```
  RELEASE SAVEPOINT [ savepoint-name ]
  ```

  ... indicates that the *savepoint-name* is optional. The square brackets should not be typed.
- **Options** When none or only one of a list of items must be chosen, the items are separated by vertical bars and the list enclosed in square brackets.

  For example, ...

  ```
  [ ASC | DESC ]
  ```

  ... indicates that you can choose one of ASC, DESC, or neither. The square brackets should not be typed.
- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces. For example ...

  ```
  [ QUOTES { ON | OFF } ]
  ```

  ... indicates that if the QUOTES option is chosen, one of ON or OFF must be provided. The braces should not be typed.

- **One or more options** If you choose more than one, separate your choices with commas. For example

  ```
  { CONNECT, DBA, RESOURCE }
  ```

Statement applicability indicators

Some statement titles are followed by an indicator in square brackets that indicate where the statement can be used. These indicators are as follows:

- **[ESQL]** The statement is for use in Embedded SQL.
- **[ISQL]** The statement can be used only in Interactive SQL.
- **[SP]** The statement is for use in stored procedures, triggers, or batches.
- **[TSQL]** The statement is implemented for compatibility with Adaptive Server Enterprise. In some cases, the statement cannot be used in stored procedures that are not in Transact-SQL format. In other cases, there an alternative statement closer to the SQL/92 standard is recommended unless Transact-SQL compatibility is an issue.

If two sets of brackets are used, the statement can be used in both environments. For example, [ESQL][SP] means a statement can be used either in Embedded SQL or in stored procedures.

ALLOCATE DESCRIPTOR statement [ESQL]

Function To allocate space for a SQL descriptor area (SQLDA).

Syntax **ALLOCATE DESCRIPTOR** *descriptor-name*
... [**WITH MAX** { *integer* | *hostvar* }]

Permissions None.

Side effects None.

See also "DEALLOCATE DESCRIPTOR statement" on page 433
"The SQL descriptor area (SQLDA)" on page 45 of the book *Adaptive Server Anywhere Programming Interfaces Guide*

Description Allocates space for a descriptor area (SQLDA). You must declare the following in your C code prior to using this statement:

```
struct sqlda * descriptor_name
```

The WITH MAX clause allows you to specify the number of variables within the descriptor area. The default size is one.

You must still call **fill_sqlda** to allocate space for the actual data items before doing a fetch or any statement that accesses the data within a descriptor area.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Open Client/Open Server.

Example The following sample program includes an example of ALLOCATE DESCRIPTOR statement usage.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

#include <sqldef.h>

EXEC SQL BEGIN DECLARE SECTION;
int      x;
short        type;
int      numcols;
char         string[100];
a_sql_statement_number  stmt = 0;
EXEC SQL END DECLARE SECTION;

int main(int argc, char * argv[])
{
   struct sqlda *      sqlda1;

   if( !db_init( &sqlca ) ) {
      return 1;
   }
   db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=d:\\asa6\\sample.db");

   EXEC SQL ALLOCATE DESCRIPTOR sqlda1 WITH MAX 25;

   EXEC SQL PREPARE :stmt FROM 'select * from employee';
   EXEC SQL DECLARE curs CURSOR FOR :stmt;
   EXEC SQL OPEN curs;

   EXEC SQL DESCRIBE :stmt into sqlda1;
   EXEC SQL GET DESCRIPTOR sqlda1 :numcols=COUNT;
      // how many columns?
   if( numcols > 25 ) {
      // reallocate if necessary
      EXEC SQL DEALLOCATE DESCRIPTOR sqlda1;
      EXEC SQL ALLOCATE DESCRIPTOR sqlda1
         WITH MAX :numcols;
   }
   type = DT_STRING;   // change the type to string
   EXEC SQL SET DESCRIPTOR sqlda1 VALUE 2 TYPE = :type;
   fill_sqlda( sqlda1 );  // now we allocate space for
the variables

   EXEC SQL FETCH ABSOLUTE 1 curs USING DESCRIPTOR
sqlda1;
   EXEC SQL GET DESCRIPTOR sqlda1 VALUE 2 :string =
DATA;

   printf("name = %s", string );

   EXEC SQL DEALLOCATE DESCRIPTOR sqlda1;
   EXEC SQL CLOSE curs;
   EXEC SQL DROP STATEMENT :stmt;

   db_string_disconnect( &sqlca, "" );
   db_fini( &sqlca );

   return 0;
}
```

ALTER DATABASE statement

Function

To upgrade a database created with previous versions of the software.

Syntax

ALTER DATABASE UPGRADE
... **[JAVA { ON | OFF }]**
... **[JCONNECT { ON | OFF }]**

Permissions

Must have DBA authority.

Not supported on Windows CE.

Side effects

Automatic commit

See also

"CREATE DATABASE statement" on page 385
"The Upgrade utility" on page 116

Description

You can use the ALTER DATABASE statement as an alternative to the Upgrade utility to upgrade a database.

The JAVA clause adds the entries for the Sybase runtime Java classes to the system tables (JAVA ON) or does not (JAVA OFF). By default, the classes are added during the upgrade.

If you wish to use the Sybase jConnect JDBC driver to access system catalog information, you need to install jConnect support. If you wish to exclude the jConnect system objects, specify JCONNECT OFF. You can still use JDBC, as long as you do not access system information. The default is to include jConnect support.

Do not use with Version 6 databases
You cannot use ALTER DATABASE against a Version 6 database to add Java or jConnect features. In order to add these features to a Version 6 database, you must run a script using Interactive SQL.

For information on adding Java support, see "Java-enabling a Version 6 database" on page 472 of the book *Adaptive Server Anywhere User's Guide*. For information on adding jConnect support, see "Installing jConnect system objects into a database" on page 524 of the book *Adaptive Server Anywhere User's Guide*.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- Upgrade a Version 5 database to enable Java operations:

```
ALTER DATABASE UPGRADE
JAVA ON
```

ALTER DBSPACE statement

Function

To modify the characteristics of the main database file or an extra dbspace. To preallocate space for a database or for the transaction log.

Syntax

ALTER DBSPACE { *dbspace-name* | **TRANSLOG** }
...
 ADD *number*
| **RENAME** *filename*

Permissions

Must have DBA authority. Must be the only connection to the database.

Side effects

Automatic commit.

See also

"CREATE DBSPACE statement" on page 389

Description

Each database is held in one or more files. A **dbspace** is an internal name that is associated with each database file. ALTER DBSPACE modifies the main database file (also called the root file) or an extra dbspace. The dbspace names for a database are held in the SYSFILE system table. The default dbspace name for the root file of a database is SYSTEM.

An ALTER DBSPACE with the ADD clause is used to preallocate disk space to a dbspace. It extends the size of a dbspace by the number of pages given by *number*. The page size of a database is defined when the database is created.

The ALTER DBSPACE statement with the ADD clause allows database files to be extended in large amounts before the space is required, rather than the normal 32 pages at a time when the space is needed. This can improve performance for loading large amounts of data and also serves to keep the dbspace files more contiguous within the file system.

The ALTER DBSPACE statement with the special dbspace name TRANSLOG preallocates disk space for the transaction log. Preallocation improves performance if the transaction log is expected to grow quickly. You may want to use this feature if, for example, you are handling many binary large objects (blobs), such as bitmaps.

The preallocation is carried out by altering the special dbspace name TRANSLOG, as follows:

```
ALTER DBSPACE TRANSLOG ADD number
```

This extends the size of the transaction log by the number of pages specified.

If you move a database file other than the root file to a different filename, directory, or device, use the ALTER DBSPACE statement with RENAME to ensure that Adaptive Server Anywhere can find the file when the database is started.

When a multi-file database is started, the start line or ODBC data source description tells Adaptive Server Anywhere where to find the root database file. The root database file (which has the default dbspace name SYSTEM) holds the system tables. Adaptive Server Anywhere looks in these system tables to find the location of the other dbspaces, and Adaptive Server Anywhere then opens each of the other dbspaces.

Using ALTER DBSPACE with RENAME on a root file has no effect.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not supported by Adaptive Server Enterprise.

Examples

- Increase the size of the SYSTEM dbspace by 200 pages.

```
ALTER DBSPACE system
ADD 200
```

- Rename the file for dbspace SYSTEM_2 to dbspace2.

```
ALTER DBSPACE system_2
RENAME 'e:\db\dbspace2.db'
```

ALTER PROCEDURE statement

Function

To replace a procedure with a modified version. You must include the entire new procedure in the ALTER PROCEDURE statement, and reassign user permissions on the procedure.

Also, to enable and disable a procedure for replication with Sybase Replication Server.

Syntax 1

ALTER PROCEDURE [*owner*.]*procedure-name* ([*parameter* , ...])
... [**RESULT** (*result-column* , ...)]
... [**ON EXCEPTION RESUME**]
... *compound-statement*

Syntax 2

ALTER PROCEDURE [*owner*.]*procedure-name*
... **REPLICATE** { **ON** | **OFF** }

Parameters

parameter:
parameter_mode parameter-name data-type
... [**DEFAULT** *expression*] | **SQLCODE** | **SQLSTATE**

parameter_mode:
IN | **OUT** | **INOUT**

result-column:
column-name data-type

Permissions

Must be the owner of the procedure or be DBA.

Side effects

Automatic commit.

See also

"CREATE PROCEDURE statement" on page 403

Description

Syntax 1 The ALTER PROCEDURE statement is identical in syntax to the CREATE PROCEDURE statement except for the first word. The ALTER PROCEDURE statement replaces the entire contents of the CREATE PROCEDURE statement with the contents of the ALTER PROCEDURE statement. Existing permissions on the procedure are maintained, and do not have to be reassigned. If a DROP PROCEDURE and CREATE PROCEDURE were carried out, execute permissions would have to be reassigned.

Syntax 2 If a procedure is to be replicated to other sites, you must set REPLICATE ON for the procedure.

Syntax 2 of the ALTER PROCEDURE statement has the same effect as the **sp_setreplicate** or **sp_setrepproc 'table'** Adaptive Server Enterprise system procedures.

You cannot combine Syntax 2 with Syntax 1.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not supported by Adaptive Server Enterprise.

ALTER SERVER statement

Function

To modify the attributes of a remote server.

Syntax

ALTER SERVER *server-name*
  [**CLASS** '*server-class*']
  [**USING** '*connection-info*']
  [**CAPABILITY** '*cap-name*' { **ON** | **OFF** }]

Parameters

server-class:
  { **ASAJDBC**
  | **ASEJDBC**
  | **ASAODBC**
  | **ASEODBC**
  | **DB2ODBC**
  | **MSSODBC**
  | **ORAODBC**
  | **ODBC** }

connection-info:
  { *machine-name:port-number* | *data-source-name* }

cap-name:

Permissions

Must have RESOURCE authority.

Supported on Windows 95 and Windows NT only.

Side effects

Automatic commit.

See also

"CREATE SERVER statement" on page 413
"Server Classes for Remote Data Access" on page 761 of the book *Adaptive Server Anywhere User's Guide*
"Troubleshooting remote data access" on page 759 of the book *Adaptive Server Anywhere User's Guide*

Description

The ALTER SERVER statement modifies the attributes of a server. These changes do not take effect until the next connection to the remote server.

CLASS clause The CLASS clause is specified to change the server's class.

USING clause The USING clause is specified to change the server's connection information.

CAPABILITY clause The CAPABILITY clause turns a server capability ON or OFF. Server capabilities are stored in the system table syscapability. The names of these capabilities are stored in the system table syscapabilityname. The syscapability table contains no entries for a remote server until the first connection is made to that server. At the first connection, Adaptive Server Anywhere interrogates the server about its capabilities and then populates the syscapability table. For subsequent connections, the server's capabilities are obtained from this table.

In general, you do not need to alter a server's capabilities. It may be necessary to alter capabilities of a generic server of class ODBC.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Open Client/Open Server.

Examples

- Change the server class of the Adaptive Server named ase_prod so its connection to Adaptive Server Anywhere is ODBC-based. Its Data Source Name is ase_prod.

```
ALTER SERVER ase_prod
CLASS 'aseodbc'
USING 'ase_prod'
```

- Change the capability of server infodc:

```
ALTER SERVER infodc
CAPABILITY 'insert select' OFF
```

ALTER TABLE statement

Function

To modify a table definition.

Also, to enable a table to take part in a Replication Server replication.

Syntax 1

ALTER TABLE [*owner*.]*table-name*
 ...
 ADD *column-definition [column-constraint ...]*
 | **ADD** *table-constraint*
 | **MODIFY** *column-definition*
 | **MODIFY** *column-name* **DEFAULT** *default-value*
 | **MODIFY** *column-name* [**NOT**] **NULL**
 | **MODIFY** *column-name* **CHECK NULL**
 | **MODIFY** *column-name* **CHECK** (*condition*)
 | { **DELETE** | **DROP** } *column-name*
 | { **DELETE** | **DROP** } **CHECK**
 | { **DELETE** | **DROP** } **UNIQUE** (*column-name*, ...)
 | { **DELETE** | **DROP** } **PRIMARY KEY**
 | { **DELETE** | **DROP** } **FOREIGN KEY** *role-name*
 | **RENAME** *new-table-name*
 | **RENAME** *column-name* **TO** *new-column-name*

Syntax 2

ALTER TABLE [*owner*.]*table-name*
 ... **REPLICATE** { **ON** | **OFF** }

Parameters

column-definition:
 column-name data-type [**NOT NULL**] [**DEFAULT** *default-value*]

column-constraint:
 UNIQUE
 | **PRIMARY KEY**
 | **REFERENCES** *table-name* [(*column-name*)] [*action*s]
 | **CHECK** (*condition*)
 | **COMPUTE** (*expression*)

default-value:
 string
 | *global variable*
 | *number*
 | **AUTOINCREMENT**
 | **CURRENT DATE**
 | **CURRENT TIME**
 | **CURRENT TIMESTAMP**
 | **NULL**
 | **USER**

table-constraint:
 UNIQUE (*column-name*, ...)
 | **PRIMARY KEY** (*column-name*, ...)
 | **CHECK** (*condition*)
 | *foreign-key-constraint*

foreign-key-constraint:
    [**NOT NULL**] **FOREIGN KEY** [*role-name*] [(*column-name*, ...)]
    ... **REFERENCES** *table-name* [(*column-name*, ...)]
    ... [*actions*] [**CHECK ON COMMIT**]

actions:
    [**ON UPDATE** *action*] [**ON DELETE** *action*]

action:
    CASCADE
    | **SET NULL**
    | **SET DEFAULT**
    | **RESTRICT**

Permissions

Must be one of the following:

- The owner of the table
- A user with DBA authority.
- A user granted ALTER permission on the table.

ALTER TABLE requires exclusive access to the table.

Global temporary tables cannot be altered unless all users that have referenced the temporary table have disconnected.

Side effects

Automatic commit.

The MODIFY and DELETE (DROP) options close all cursors for the current connection.

See also

"CREATE TABLE statement" on page 415
"DROP statement" on page 451
"SQL Data Types" on page 219

Description

Syntax 2 When a table has REPLICATE ON, all changes to the table are sent to Replication Server for replication. The replication definitions in Replication Server are used to decide which table changes are sent to other sites. The remainder of this section describes syntax 1.

Syntax 1 The ALTER TABLE statement changes table attributes (column definitions, constraints) in a table that was previously created. Note that the syntax allows a list of alter clauses; however, only one table-constraint or column-constraint can be added, modified or deleted in one ALTER TABLE statement.

You cannot use ALTER TABLE on a local temporary table.

ALTER TABLE is prevented whenever the statement affects a table that is currently being used by another connection. ALTER TABLE can be time-consuming, and the server will not process requests referencing the table while the statement is being processed.

Before version 5.0, all table and column constraints were held in a single table constraint. Consequently, for these databases individual constraints on columns cannot be deleted using the MODIFY column-name CHECK NULL clause or replaced using the MODIFY column-name CHECK (condition) clause. To use these statements, the entire table constraint should be deleted and the constraints added back using the MODIFY column-name CHECK (condition) clause. At this point you can use MODIFY CHECK.

ADD column-definition Add a new column to the table. The table must be empty to specify NOT NULL.

NULL values
Adaptive Server Anywhere optimizes the creation of columns which are allowed to contain NULL. The first column allowed to contain NULL allocates room for eight such columns, and initializes all eight to be NULL. (This requires no extra storage.) Thus, the next seven columns added require no changes to the rows of the table.

Adding a ninth column then allocates room for another eight such columns and modifies each row of the table to allocate the extra space. Consequently, seven out of eight column additions run quickly.

ADD table-constraint Add a constraint to the table. See "CREATE TABLE statement" on page 415 for a full explanation of table constraints.

If PRIMARY KEY is specified, the table must not already have a primary key that was created by the CREATE TABLE statement or another ALTER TABLE statement.

MODIFY column-definition Change the length or data type of an existing column in a table. If NOT NULL is specified, a NOT NULL constraint is added to the named column. Otherwise, the NOT NULL constraint for the column will not be changed. If necessary, the data in the modified column will be converted to the new data type. If a conversion error occurs, the operation will fail and the table will be left unchanged.

You cannot modify a column to make it a computed column. Computed columns can only be added or dropped.

Deleting an index, constraint, or key
If the column is contained in a uniqueness constraint, a foreign key, or a primary key, then the constraint or key must be deleted before the column can be modified. If a primary key is deleted, all foreign keys referencing the table will also be deleted.

You cannot MODIFY a table or column constraint. To change a constraint, you must DELETE the old constraint and ADD the new constraint.

MODIFY column-name DEFAULT default-value Change the default value of an existing column in a table. To remove a default value for a column, specify DEFAULT NULL.

MODIFY column-name [NOT] NULL Change the NOT NULL constraint on the column to allow or disallow NULL values in the column.

MODIFY column-name CHECK NULL Delete the check constraint for the column. This statement cannot be used on databases created before version 5.0.

MODIFY column-name CHECK (condition) Replace the existing CHECK condition for the column with the one specified. This statement cannot be used on databases created before version 5.0.

DELETE column-name Delete the column from the table. If the column is contained in any index, uniqueness constraint, foreign key, or primary key then the index, constraint or key must be deleted before the column can be deleted. This does not delete CHECK constraints that refer to the column.

DELETE CHECK Delete all check constraints for the table. This includes both table check constraints and column check constraints.

DELETE UNIQUE (column-name,...) Delete a uniqueness constraint for this table. Any foreign keys referencing this uniqueness constraint (rather than the primary key) will also be deleted.

DELETE PRIMARY KEY Delete the primary key constraint for this table. All foreign keys referencing the primary key for this table will also be deleted.

DELETE FOREIGN KEY role-name Delete the foreign key constraint for this table with the given role name.

RENAME new-table-name Change the name of the table to the *new-table-name*. Note that any applications using the old table name will need to be modified. Also, any foreign keys which were automatically assigned the same name as the old table name will not change names.

RENAME column-name TO new-column-name Change the name of the column to the *new-column-name*. Note that any applications using the old column name will need to be modified.

Standards and compatibility

- **SQL/92** Intermediate level feature.
- **Sybase** Supported by Adaptive Server Enterprise.

Examples

- Add a new column to the employees table showing which office they work in.

```
ALTER TABLE employee
ADD office CHAR(20) DEFAULT 'Boston'
```

- Drop the office column from the employees table.

```
ALTER TABLE employee
DELETE office
```

- The address column in the customer table can currently hold up to 35 characters. Allow it to hold up to 50 characters.

```
ALTER TABLE customer
MODIFY address CHAR(50)
```

- Add a column to the customer table assigning each customer a sales contact.

```
ALTER TABLE customer
ADD sales_contact INTEGER
REFERENCES employee (emp_id)
ON UPDATE CASCADE
ON DELETE SET NULL
```

  This foreign key is constructed with a cascading updates and is set null on deletes. If an employee has their employee ID changed, the column is updated to reflect this change. If an employee leaves the company and has their employee ID deleted, the column is set to NULL.

ALTER TRIGGER statement

Function

To replace a trigger definition with a modified version.

You must include the entire new trigger definition in the ALTER TRIGGER statement.

Syntax

```
ALTER TRIGGER trigger-name trigger-time trigger-event [, trigger-event,..]
      ... [ ORDER integer ] ON table-name
      ... [ REFERENCING [ OLD AS old-name ]
                                          [ NEW AS new-name ] ]
                                          [ REMOTE AS remote-name ] ]
      ... [ FOR EACH { ROW | STATEMENT } ]
      ... [ WHEN ( search-condition ) ]
      ... [ IF UPDATE ( column-name ) THEN
      ... [ { AND | OR } UPDATE ( column-name ) ] ... ]
                  ... compound-statement
      ... [ ELSEIF UPDATE ( column-name ) THEN
      ... [ { AND | OR } UPDATE ( column-name ) ] ...
                  ...  compound-statement
      ...  END IF ] ]
```

Parameters

trigger-time:
BEFORE | **AFTER** | **RESOLVE**

trigger-event:
DELETE | **INSERT** | **UPDATE** | **UPDATE OF** *column-list*

Permissions

Must be the owner of the table on which the trigger is defined, or be DBA, or have ALTER permissions on the table.

Side effects

Automatic commit.

See also

"CREATE TRIGGER statement" on page 424
"DROP statement" on page 451

Description

The ALTER TRIGGER statement is identical in syntax to the CREATE TRIGGER statement except for the first word. The ALTER TRIGGER statement replaces the entire contents of the CREATE TRIGGER statement with the contents of the ALTER TRIGGER statement.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not supported by Adaptive Server Enterprise.

ALTER VIEW statement

Function

To replace a view definition with a modified version. You must include the entire new view definition in the ALTER VIEW statement, and reassign permissions on the view

Syntax

ALTER VIEW
... [*owner*.]*view-name* [(*column-name*, ...)]
... **AS** *select-without-order-by*
... [**WITH CHECK OPTION**]

Permissions

Must be owner of the view or have DBA authority.

Side effects

Automatic commit.

See also

"CREATE VIEW statement" on page 430
"DROP statement" on page 451

Description

The ALTER VIEW statement is identical in syntax to the CREATE VIEW statement except for the first word. The ALTER VIEW statement replaces the entire contents of the CREATE VIEW statement with the contents of the ALTER VIEW statement. Existing permissions on the view are maintained, and do not have to be reassigned. If a DROP VIEW and CREATE VIEW were carried out, permissions on the view would have to be reassigned.

Standards and compatibility

- ♦ **SQL/92** Vendor extension
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

ALTER WRITEFILE statement

Function

To modify the configuration of a write file.

Syntax

ALTER WRITEFILE *write-file-name*
... **REFERENCES** *db-file-name*

Permissions

Must have DBA authority.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

"CREATE WRITEFILE statement" on page 432
"The Write File utility" on page 121
"Working with write files" on page 618 of the book *Adaptive Server Anywhere User's Guide*

Description

The ALTER WRITEFILE statement changes the read-only database file to which the file refers. If you move the database file from one directory to another, you can use this statement to point the write file to the new location.

The path name of the database file is relative to the database server.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- The following statement changes the existing write file *c:\readwrite.wrt* to point to the database file *h:\readonly.db*.

```
ALTER WRITEFILE 'c:\\readwrite.wrt'
REFERENCES 'h:\\readonly.db'
```

BACKUP statement

Function

To back up a database and transaction log.

Syntax 1 (image backup)

```
BACKUP DATABASE
     DIRECTORY backup_directory
     [ DBFILE ONLY ]
     [ TRANSACTION LOG ONLY ]
     [ TRANSACTION LOG RENAME ]
     [ TRANSACTION LOG TRUNCATE ]
```

Syntax 2 (archive backup)

```
BACKUP DATABASE TO archive_root
     [ CRC { ON | OFF } ]
     [ ATTENDED { ON | OFF } ]
     [ WITH COMMENT comment string ]
```

Parameters

archive_root: *string*

nnnn: *integer*

comment-string: *string*

Permissions

Must have DBA authority.

Side effects

None.

See also

"RESTORE statement" on page 532

Description

Image backup (syntax 1) creates copies of each of the database files, in the same way that the *dbbackup* utility does. In the case of the SQL statement, however, the backup is made on the server, while the Backup utility makes the backup from a client machine.

Optionally, only the database file(s) or transaction log can be saved. The log may also be renamed or truncated after the backup has completed.

To restore from an image backup, copy the saved files back to their original locations and reapply transaction logs as described in "Backing up your database" on page 565 of the book *Adaptive Server Anywhere User's Guide*.

You can use archive backup to create a single file that hold all the required backup information. The destination can be either a file name or a tape drive.

Each BACKUP operation, whether image or archive, updates a history file called *backup.syb*. This file is stored in the same directory as the database server executable.

backup_directory The target location on disk for those files, relative to the server's current directory at startup. If the directory does not already exist, it is created.

Archive backup (syntax 2) creates an **archive** of the database on disk or tape containing all the database files. Archive backups are only supported on NT and Unix platforms.

To restore a database from an archive backup, use the RESTORE statement.

archive_root The file name or tape drive for the archive file. The archive format saves all the backup data inside a single file.

To back up to tape, you must specify the device name of the tape drive. For example, on NT the first tape drive is:

```
\\.\tape0
```

The '\' is an escape character in SQL strings, so each backslash must be doubled.

Example

- Back up the current database and the transaction log to a file, truncating and renaming the existing transaction log.

  ```
  BACKUP DATABASE
  DIRECTORY 'd:\\temp\\backup'
  TRANSACTION LOG RENAME
  ```

  The option to rename the transaction log is useful especially in replication environments, where the old transaction log is still required.

BEGIN... END statement

Function

Groups SQL statements together.

Syntax

```
[ statement-label : ]
   ... BEGIN [ [ NOT ] ATOMIC ]
   ...   [ local-declaration ; ... ]
   ...   statement-list
   ...   [ EXCEPTION [ exception-case ... ] ]
   ... END [ statement-label ]
```

Parameters

```
local-declaration:
    variable-declaration
   | cursor-declaration
   | exception-declaration
   | temporary-table-declaration

variable-declaration:
   DECLARE variable-name data-type

exception-declaration:
   DECLARE exception-name EXCEPTION
   FOR SQLSTATE [ VALUE ] string

exception-case:
    WHEN exception-name [ ,... ] THEN statement-list
   | WHEN OTHERS THEN statement-list
```

Permissions

None.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436
"DECLARE LOCAL TEMPORARY TABLE statement" on page 441
"LEAVE statement" on page 502
"SIGNAL statement" on page 560
"RESIGNAL statement" on page 531
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in control statements within a procedure or trigger.

A compound statement allows one or more SQL statements to be grouped together and treated as a unit. A compound statement starts with the keyword BEGIN and ends with the keyword END. Immediately following the BEGIN, a compound statement can have local declarations that only exist within the compound statement. A compound statement can have a local declaration for a variable, a cursor, a temporary table, or an exception. Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures that are called from within a compound statement.

If the ending *statement-label* is specified, it must match the beginning *statement-label*. The LEAVE statement can be used to resume execution at the first statement after the compound statement. The compound statement that is the body of a procedure or trigger has an implicit label that is the same as the name of the procedure or trigger.

For a complete description of compound statements and exception handling, see "Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Supported by Adaptive Server Enterprise. This does not mean that all statements inside a compound statement are supported.

  The BEGIN and END keywords are not required in Transact-SQL.

  BEGIN and END are used in Transact-SQL to group a set of statements into a single compound statement, so that control statements such as IF ... ELSE , which only affect the performance of a single SQL statement, can affect the performance of the whole group. The ATOMIC keyword is not supported by Adaptive Server Enterprise.

  In Transact-SQL. DECLARE statements need not immediately follow a BEGIN keyword, and the cursor or variable that is declared exists for the duration of the compound statement. You should declare variables at the beginning of the compound statement for compatibility.

Example

- The body of a procedure or trigger is a compound statement.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany
CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION FOR
      SQLSTATE '02000' ;
   DECLARE curThisCust CURSOR FOR
      SELECT company_name, CAST(
            sum(sales_order_items.quantity *
            product.unit_price) AS INTEGER) VALUE
      FROM customer
            LEFT OUTER JOIN sales_order
            LEFT OUTER JOIN sales_order_items
            LEFT OUTER JOIN product
      GROUP BY company_name ;
   DECLARE ThisValue INT ;
   DECLARE ThisCompany CHAR(35) ;
   SET TopValue = 0 ;
   OPEN curThisCust ;

   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
         INTO ThisCompany, ThisValue ;
      IF SQLSTATE = err_notfound THEN
         LEAVE CustomerLoop ;
      END IF ;
      IF ThisValue > TopValue THEN
         SET TopValue = ThisValue ;
         SET TopCompany = ThisCompany ;
      END IF ;
   END LOOP CustomerLoop ;

CLOSE curThisCust ;
END
```

BEGIN TRANSACTION statement

Function To begin a user-defined transaction.

Syntax **BEGIN TRAN[SACTION]** [*transaction-name*]

Permissions None.

Side effects None.

See also "COMMIT TRANSACTION statement" on page 379
"SAVEPOINT statement" on page 541

Description The optional parameter *transaction-name* is the name assigned to this transaction. It must be a valid identifier. Use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

When executed inside a transaction, the BEGIN TRANSACTION statement increases the nesting level of transactions by one. The nesting level is decreased by a COMMIT statement. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

Chained and unchained modes

Both Adaptive Server Enterprise and Adaptive Server Anywhere have two transaction modes:

The default Adaptive Server Enterprise transaction mode, called **unchained mode**, commits each statement individually, unless an explicit BEGIN TRANSACTION statement is executed to start a transaction. In contrast, the ISO SQL/92 compatible **chained mode** only commits a transaction when an explicit COMMIT is executed or when a statement that carries out an autocommit (such as data definition statements) is executed.

You can control the mode by setting the CHAINED database option. The default setting for ODBC and Embedded SQL connection in Adaptive Server Anywhere is ON, in which case Adaptive Server Anywhere runs in chained mode. The default for TDS connections is OFF.

In unchained mode, a transaction is implicitly started before any data retrieval or modification statement. These statements include: DELETE, INSERT, OPEN, FETCH, SELECT, and UPDATE. You must still explicitly end the transaction with a COMMIT or ROLLBACK statement.

You cannot alter the CHAINED option within a transaction.

Caution
When calling a stored procedure, you should ensure that it operates correctly under the required transaction mode.

For more information about the CHAINED option and the chained mode, see "CHAINED option" on page 146.

The current nesting level is held in the global variable **@@trancount**. The **@@trancount** variable has a value of zero before the first BEGIN TRANSACTION statement is executed, and only a COMMIT executed when **@@trancount** is equal to one makes changes to the database permanent.

A ROLLBACK statement without a transaction or savepoint name always rolls back statements to the outermost BEGIN TRANSACTION (explicit or implicit) statement, and cancels the entire transaction.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Supported by Adaptive Server Enterprise.

Example

- The following batch reports successive values of **@@trancount** as 0, 1, 2, 1, 0. The values are printed on the server window.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT
PRINT @@trancount
COMMIT
PRINT @@trancount
```

@@trancount in Adaptive Server Enterprise and Adaptive Server Anywhere

You should not rely on the value of**@@trancount** for more than keeping track of the number of explicit BEGIN TRANSACTION statements that have been issued.

When Adaptive Server Enterprise starts a transaction implicitly, the @@trancount variable is set to 1. Adaptive Server Anywhere does not set the @@trancount value to 1 when a transaction is started implicitly. Consequently, the Adaptive Server Anywhere **@@trancount** variable has a value of zero before any BEGIN TRANSACTION statement (even though there is a current transaction), while in Adaptive Server Enterprise (in **chained** mode) it has a value of 1.

For transactions starting with a BEGIN TRANSACTION statement, **@@trancount** has a value of 1 in both Adaptive Server Anywhere and Adaptive Server Enterprise after the first BEGIN TRANSACTION statement. If a transaction is implicitly started with a different statement, and a BEGIN TRANSACTION statement is then executed, **@@trancount** has a value of 1 in Adaptive Server Anywhere, and a value of 2 in Adaptive Server Enterprise after the BEGIN TRANSACTION statement.

CALL statement

Function

To invoke a procedure.

Syntax

[*variable* =] **CALL** *procedure-name* ([*expression* ,...])

[*variable* =] **CALL** *procedure-name* ([*parameter-name* = *expression* ,...])

Permissions

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

"CREATE PROCEDURE statement" on page 403
"GRANT statement" on page 484
"EXECUTE statement" on page 462
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The CALL statement invokes a procedure that has been previously created with a CREATE PROCEDURE statement. When the procedure completes, any INOUT or OUT parameter values will be copied back.

The argument list can be specified by position or by using keyword format. By position, the arguments will match up with the corresponding parameter in the parameter list for the procedure. By keyword, the arguments are matched up with the named parameters.

All arguments are optional: procedure arguments can be assigned default values in the CREATE PROCEDURE statement, and missing parameters are assigned the default value or, if no default is set, NULL.

Inside a procedure, a CALL statement can be used in a DECLARE statement when the procedure returns result sets (see "Returning results from procedures" on page 246 of the book *Adaptive Server Anywhere User's Guide*).

Procedures can return a value (as a status indicator, say) using the RETURN statement. You can save this return value in a variable using the equality sign as an assignment operator:

```
CREATE VARIABLE returnval INT ;
returnval = CALL proc_integer ( arg1 = val1, ... )
```

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Not supported by Adaptive Server Enterprise. For an alternative that is supported, see "EXECUTE statement" on page 462.

Examples

- Call the **sp_customer_list** procedure. This procedure has no parameters, and returns a result set.

```
CALL sp_customer_list()
```

♦ The following Interactive SQL example creates a procedure to return the number of orders placed by the customer whose ID is supplied, creates a variable to hold the result, calls the procedure, and displays the result.

```
-- Set the statement delimiter to create the
procedure
SET OPTION COMMAND_DELIMITER = ';;'
-- Create the procedure
CREATE PROCEDURE OrderCount (IN customer_ID INT, OUT
Orders INT)
BEGIN
SELECT COUNT("DBA".sales_order.id)
INTO Orders
FROM "DBA".customer
KEY LEFT OUTER JOIN "DBA".sales_order
WHERE "DBA".customer.id = customer_ID ;
END ;;
-- Reset the statement delimiter to semicolon.
SET OPTION COMMAND_DELIMITER = ';'
-- Create a variable to hold the result
CREATE VARIABLE Orders INT ;
-- Call the procedure, FOR customer 101
-- -----------------------------
CALL OrderCount ( 101, Orders) ;
--------------------------------
--  Display the result
SELECT Orders FROM DUMMY ;
```

CASE statement

Function

Select execution path based on multiple cases.

Syntax

CASE *value-expression*
... **WHEN** [*constant* | **NULL**] **THEN** *statement-list* ...
... [**WHEN** [*constant* | **NULL**] **THEN** *statement-list*] ...
... **ELSE** *statement-list*
... **END CASE**

Permissions

None.

Side effects

None.

See also

"BEGIN... END statement" on page 361
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The CASE statement is a control statement that allows you to choose a list of SQL statements to execute based on the value of an expression. If a WHEN clause exists for the value of *value-expression*, the *statement-list* in the WHEN clause is executed. If no appropriate WHEN clause exists, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed. Execution resumes at the first statement after the END CASE.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following procedure using a case statement classifies the products listed in the product table of the sample database into one of shirt, hat, shorts, or unknown.

The following procedure uses a case statement to classify the results of a query.

```
CREATE PROCEDURE ProductType (IN product_id INT, OUT
type CHAR(10))
   BEGIN
   DECLARE prod_name CHAR(20) ;
   SELECT name INTO prod_name FROM "DBA"."product"
   WHERE id = product_id;
   CASE prod_name
   WHEN 'Tee Shirt' THEN
      SET type = 'Shirt'
   WHEN 'Sweatshirt' THEN
      SET type = 'Shirt'
   WHEN 'Baseball Cap' THEN
      SET type = 'Hat'
   WHEN 'Visor' THEN
      SET type = 'Hat'
```

```
WHEN 'Shorts' THEN
   SET type = 'Shorts'
ELSE
   SET type = 'UNKNOWN'
END CASE ;
END
```

CHECKPOINT statement

Function

To **checkpoint** the database.

Syntax

CHECKPOINT

Permissions

DBA authority is required to checkpoint the network database server.

No permissions are required to checkpoint the personal database server.

Side effects

None.

Description

The CHECKPOINT statement checkpoints the database. Checkpoints are also performed automatically by the database server. It is not normally required for an application to ever issue the CHECKPOINT statement.

For a full description of checkpoints, see "Backup and Data Recovery" on page 553 of the book *Adaptive Server Anywhere User's Guide*.

Standards and compatibility

- ♦ **SQL/92** Vendor extension
- ♦ **Sybase** Supported by Adaptive Server Enterprise.

CLEAR statement [ISQL]

Function	To clear the Interactive SQL Data window.
Syntax	**CLEAR**
Permissions	None.
Side effects	Closes the cursor associated with the data being cleared.
Description	The CLEAR statement is used to clear the Interactive SQL Data window.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not applicable

CLOSE statement [ESQL] [SP]

Function

To close a cursor.

Syntax

CLOSE *cursor-name*

Parameters

cursor-name:	*identifier*
cursor-name:	{ *identifier* | *host-variable* }

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

"OPEN statement" on page 511
"DECLARE CURSOR statement" on page 436
"PREPARE statement" on page 519

Description

This statement closes the named cursor.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Adaptive Server Enterprise.

Examples

The following examples close cursors in Embedded SQL.

```
EXEC SQL CLOSE employee_cursor;
```

```
EXEC SQL CLOSE :cursor_var;
```

The following procedure uses a cursor.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35),
OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
      FOR SQLSTATE '02000' ;
   DECLARE curThisCust CURSOR FOR
   SELECT company_name, CAST(
   sum(sales_order_items.quantity *
   product.unit_price) AS INTEGER) VALUE
   FROM customer
   LEFT OUTER JOIN sales_order
   LEFT OUTER JOIN sales_order_items
   LEFT OUTER JOIN product
   GROUP BY company_name ;

   DECLARE ThisValue INT ;
   DECLARE ThisCompany CHAR(35) ;
   SET TopValue = 0 ;
   OPEN curThisCust ;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
      INTO ThisCompany, ThisValue ;
         IF SQLSTATE = err_notfound THEN
            LEAVE CustomerLoop ;
         END IF ;
         IF ThisValue > TopValue THEN
            SET TopValue = ThisValue ;
            SET TopCompany = ThisCompany ;
         END IF ;
      END LOOP CustomerLoop ;
   CLOSE curThisCust ;
END
```

COMMENT statement

Function

To store a comment in the system tables for a database object.

Syntax

COMMENT ON
  {
    COLUMN [*owner*.]*table-name.column-name*
    | **FOREIGN KEY** [*owner*.]*table-name.role-name*
    | **INDEX** [*owner*.]*index-name*
    | **LOGIN** *integrated_login_id*
    | **PUBLICATION** [*owner*.]*publication-name*
    | **PROCEDURE** [*owner*.]*procedure-name*
    | **SUBSCRIPTION** [*owner*.]*subscription-name*
    | **TABLE** [*owner*.]*table-name*
    | **USER** *userid*
    | **TRIGGER** [*owner*.]*trigger-name*
    | **VIEW** [*owner*.]*view-name*
  }
  IS *comment*

Parameters

comment:
  { *string* | **NULL** }

Permissions

Must either be the owner of the database object being commented, or have DBA authority.

Side effects

Automatic commit.

Description

Several system tables have a column named Remarks that allows you to associate a comment with a database item (SYSUSERPERM, SYSTABLE, SYSCOLUMN, SYSINDEX, SYSLOGIN, SYSFOREIGNKEY, SYSPROCEDURE, SYSTRIGGER). The COMMENT ON statement allows you to set the Remarks column in these system tables. A comment can be removed by setting it to NULL.

For a comment on an index or trigger, the owner of the comment is the owner of the table on which the index or trigger is defined.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Examples

The following examples show how to add and remove a comment.

- Add a comment to the employee table.

```
COMMENT
ON TABLE employee
IS "Employee information"
```

- Remove the comment from the employee table.

```
COMMENT
```

```
ON TABLE employee
IS NULL
```

COMMIT statement

Function

To make any changes to the database permanent.

Syntax

COMMIT [**WORK**]

Permissions

Must be connected to the database.

Side effects

Closes all cursors except those opened WITH HOLD.

See also

"ROLLBACK statement" on page 538
"PREPARE TO COMMIT statement" on page 522
"CONNECT statement" on page 381
"SET CONNECTION statement" on page 551
"DISCONNECT statement" on page 450

Description

The COMMIT statement ends a logical unit of work (transaction) and makes all changes made during this transaction permanent in the database. A transaction is defined as the database work done between successful COMMIT and ROLLBACK statements on a single database connection.

The COMMIT statement is also used as the second phase of a two-phase commit operation. For more information, see "Typical inconsistencies" on page 378 of the book *Adaptive Server Anywhere User's Guide*, and "PREPARE TO COMMIT statement" on page 522.

The changes committed are those made by the data manipulation statements: INSERT, UPDATE, and DELETE, as well as the Interactive SQL load statement INPUT.

Data definition statements all do an automatic commit. They are:

- ALTER
- COMMENT
- CREATE
- DROP
- GRANT
- REVOKE
- SET OPTION

The COMMIT statement fails if the database server detects any invalid foreign keys. This makes it impossible to end a transaction with any invalid foreign keys. Usually, foreign key integrity is checked on each data manipulation operation. However, if either the database option WAIT_FOR_COMMIT is set ON or a particular foreign key was defined with a CHECK ON COMMIT clause, the database server will not check integrity until the COMMIT statement is executed. For a two-phase commit operation, these errors will be reported on the first phase (PREPARE TO COMMIT), not on the second phase (COMMIT).

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Adaptive Server Enterprise.

Examples

- The following statement commits the current transaction:

```
COMMIT
```

COMMIT TRANSACTION statement [T-SQL]

Function

To terminate a user-defined transaction or make changes to the database permanent.

Syntax

COMMIT TRAN[SACTION] [*transaction-name*]

Authorization

None.

Side effects

None.

See also

"BEGIN TRANSACTION statement" on page 364

Description

The optional parameter *transaction-name* is the name assigned to this transaction. It must be a valid identifier. You should use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

When executed inside a transaction, the COMMIT TRANSACTION statement decreases the nesting level of transactions by one. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

For a discussion of transaction nesting in Adaptive Server Enterprise and Adaptive Server Anywhere, see "BEGIN TRANSACTION statement" on page 364.

Savepoints and the ROLLBACK statement are discussed in "SQL Language Elements" on page 179.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

- The following Transact-SQL batch reports successive values of **@@trancount** as 0, 1, 2, 1, 0.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
go
```

CONFIGURE statement [ISQL]

Function To activate the Interactive SQL configuration window.

Syntax **CONFIGURE**

Permissions None.

Side effects None.

See also "SET OPTION statement" on page 553

Description The CONFIGURE statement activates the Interactive SQL configuration window. This window displays the current settings of all Interactive SQL options. It does not display or allow you to modify database options.

If you press ENTER and you have selected Save Options to Database the options will be written to the SYSOPTION table in the database and the database server will perform an automatic COMMIT. If you do not select Save Options to Database, the options are set temporarily and remain in effect for the current database connection only.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

CONNECT statement [ESQL] [ISQL]

Function

To establish a connection to a database.

Syntax 1

CONNECT
... [**TO** *engine-name*]
... [**DATABASE** *database-name*]
... [**AS** *connection-name*]
... [**USER**] *userid* [**IDENTIFIED BY** *password*]

Syntax 2

CONNECT USING *connect-string*

Parameters

engine-name: { *identifier* | *string* | *host-variable* }

database-name: { *identifier* | *string* | *host-variable* }

connection-name: { *identifier* | *string* | *host-variable* }

userid: { *identifier* | *string* | *host-variable* }

password: { *identifier* | *string* | *host-variable* }

connect-string: { *connection string* | *host variable* }

Permissions

None.

Side effects

None.

See also

"GRANT statement" on page 484
"DISCONNECT statement" on page 450
"SET CONNECTION statement" on page 551

Description

The CONNECT statement establishes a connection to the database identified by *database-name* running on the server identified by *engine-name*.

Embedded SQL behavior In Embedded SQL, if no *engine-name* is specified, the default local database server will be assumed (the first database server started). If no *database-name* is specified, the first database on the given server will be assumed.

Interactive SQL behavior If no database or server is specified in the CONNECT statement, Interactive SQL remains connected to the current database, rather than to the default server and database. If a database name is specified without a server name, Interactive SQL will try to connect to the specified database on the current server. If a server name is specified without a database name, Interactive SQL will connect to the default database on the specified server. For example, in the following batch, the two tables are created in the same database.

```
CREATE TABLE t1( c1 int );
CONNECT DBA IDENTIFIED BY SQL;
CREATE TABLE t2 (c1 int );
```

No other database statements are allowed until a successful CONNECT statement has been executed.

(In Embedded SQL, WHENEVER, SET SQLCA and some DECLARE statements do not generate code and thus may appear before the CONNECT statement in the source file.)

The user ID and password are used for checking the permissions on SQL statements. If the password or the user ID and password are not specified, the user will be prompted to type the missing information.

In Embedded SQL, the user ID and password are used for permission checks on all dynamic SQL statements.

If you are connected to a user ID with DBA authority, you can connect to another user ID without specifying a password. (The output of DBTRAN requires this capability.) For example, if you are connected to a database as DBA, you can connect without a password with the statement:

```
connect other_user_id
```

In Embedded SQL, you can connect without a password by using a host variable for the password and setting the value of the host variable to be the null pointer.

A connection can optionally be named by specifying the AS clause. This allows multiple connections to the same database, or multiple connections to the same or different database servers, all simultaneously. Each connection has its own associated transaction. You may even get locking conflicts between your transactions if, for example, you try to modify the same record in the same database from two different connections.

Multiple connections are managed through the concept of a current connection. After a successful connect statement, the new connection becomes the current one. To switch to a different connection, use the SET CONNECTION statement. The DISCONNECT statement is used to drop connections.

For Syntax 2, a **connect string** is a list of parameter settings of the form **keyword**=*value*. For a description of valid settings for the connection string, see "Connection parameters" on page 46 of the book *Adaptive Server Anywhere User's Guide*. The connect string must be enclosed in single quotes.

Standards and compatibility

- ♦ **SQL/92** Syntax 1 is a full SQL feature; syntax 2 is a vendor extension.
- ♦ **Sybase** Open Client Embedded SQL supports a different syntax for the CONNECT statement.

Examples

- ♦ The following are examples of CONNECT usage within Embedded SQL.

```
EXEC SQL CONNECT AS :conn_name
USER :userid IDENTIFIED BY :password;
```

```
EXEC SQL CONNECT USER "dba" IDENTIFIED BY "sql";
```

- Connect to a database from Interactive SQL. Interactive SQL prompts for a user ID and a password.

```
CONNECT
```

- Connect to the default database as DBA, from Interactive SQL. Interactive SQL promptS for a password.

```
CONNECT USER "DBA"
```

- Connect to the sample database as the DBA, from Interactive SQL.

```
CONNECT
TO asademo
USER DBA
IDENTIFIED BY sql
```

- Connect to the sample database using a connect string, from Interactive SQL.

```
CONNECT
USING 'UID=DBA;PWD=sql;DBN=asademo'
```

CREATE COMPRESSED DATABASE statement

Function To create a compressed database from an existing database file, or to expand a compressed database.

Syntax **CREATE** [**COMPRESSED** | **EXPANDED**] **DATABASE** *new-db-file-name*
... **FROM** *old-db-file-name*

Parameters *new-db-file-name* | *old-db-file-name:* '*file-name*'

Permissions

- The permissions required to execute this statement are set on the server command line, using the `-gu` command-line option. The default setting is to require DBA authority.
- The account under which the server is running must have write permissions on the directories where files are created.
- The source database file must not be currently loaded on a server.
- Not supported on Windows CE.

Side effects An operating system file is created.

See also "The Compression utility" on page 75
"The Uncompression utility" on page 108

Description Creates a compressed database file from an uncompressed database file, or an uncompressed database file from a compressed one.

Any relative path is taken relative to the current working directory of the server.

You cannot use this statement on files other than the primary database file.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- The following statement creates a compressed database file named *compress.db* in the *C:\* directory from a database file named *full.db* in the current working directory of the server.

```
CREATE COMPRESSED DATABASE 'C:\\compress.db'
FROM 'full.db'
```

- The following statement creates an uncompressed database file named *full.db* in the *C:\* directory from a compressed database file named *compress.db* in the current working directory of the server.

```
CREATE EXPANDED DATABASE 'C:\\full.db'
FROM 'compress.db'
```

CREATE DATABASE statement

Function To create a database. The database is an operating system file.

Syntax

```
CREATE  DATABASE  db-file-name
      ... [
      ...   [ [ TRANSACTION ] LOG OFF |
                    [ TRANSACTION ] LOG ON [ log-file-name ]
                    [ MIRROR mirror-file-name ]
            ]
      ...   [ CASE { RESPECT | IGNORE } ]
      ...   [ PAGE SIZE page-size ]
      ...   [ COLLATION collation-label ]
      ...   [ ENCRYPTED { ON | OFF } ]
      ...   [ BLANK PADDING { ON | OFF } ]
      ...   [ ASE [ COMPATIBLE ] ]
      ...   [ JAVA { ON | OFF } ]
      ...   [ JCONNECT { ON | OFF } ]
      ]
```

Parameters

db-file-name | *log-file-name* | *mirror-file-name* :
'*file-name*'

page-size :
1024 | **2048** | **4096**

collation-label: string

Permissions The permissions required to execute this statement are set on the server command line, using the `-gu` command-line option. The default setting is to require DBA authority.

The account under which the server is running must have write permissions on the directories where files are created.

Not supported on Windows CE.

Side effects An operating system file is created.

See also "ALTER DATABASE statement" on page 344
"DROP DATABASE statement" on page 453
"The Initialization utility" on page 84

Description Creates a database file with the supplied name and attributes.

File name The file names (*db-file-name*, *log-file-name*, *mirror-file-name*) are strings containing operating system file names. As literal strings, they must be enclosed in single quotes.

- If you specify a path, any backslash characters (\) must be doubled if they are followed by an n or an x. This prevents them being interpreted as a new line character (\n) or as a hexadecimal number (\x), according to the rules for strings in SQL.

  It is safer to always escape the backslash character. For example:

  ```
  CREATE DATABASE 'c:\\sybase\\my_db.db'
  LOG ON 'e:\\logdrive\\my_db.log'
  ```

- If you specify no path, or a relative path, the database file is created relative to the working directory of the server. If you specify no path for a log file, the file is created in the same directory as the database file.
- If you provide no file extension, a file is created with extension *.db* for databases or *.log* for the transaction log.

TRANSACTION LOG clause The transaction log is a file where the database server logs all changes made by all users no matter what application system is being used. The transaction log plays a key role in backup and recovery (see "The transaction log" on page 557 of the book *Adaptive Server Anywhere User's Guide*), and in data replication. If the filename has no path, it is placed in the same directory as the database file.

MIRROR clause A transaction log mirror is an identical copy of a transaction log, usually maintained on a separate device, for greater protection of your data. By default, Adaptive Server Anywhere does not use a mirrored transaction log. If you do wish to use a transaction log mirror, this option allows you to provide a filename.

CASE clause For databases created with this option, all values are considered to be case sensitive in comparisons and string operations.

This option is provided for compatibility with the ISO/ANSI SQL standard. The default is that all comparisons are case insensitive.

User ID and password
All databases are created with at least one user ID, **DBA,** with password **sql**. If you create a database requiring case-sensitive comparisons, the **DBA** user ID and its password must be entered in upper case.

PAGE SIZE clause The page size for a database can be 512, 1024, 2048 or 4096 bytes, with 1024 being the default. Other values for the size will be changed to the next larger size. Large databases usually benefit from a larger page size.

For example:

```
CREATE DATABASE 'c:\\sybase\\my_db.db'
PAGE SIZE 4096
```

Page size limit
The page size cannot be larger than the page size used by the current server. The server page size is taken from the first database loaded or is set on the server command line using the `-gp` command-line option.

COLLATION clause The collation sequence used for all string comparisons in the database.

For more information on custom collating sequences, see "Database Collations and International Languages" on page 289 of the book *Adaptive Server Anywhere User's Guide*.

ENCRYPTED clause Encryption makes it more difficult for someone to decipher the data in your database by using a disk utility to look at the file. File compaction utilities are not able to compress encrypted database files as much as unencrypted ones.

BLANK PADDING clause If you specify blank padding, trailing blanks are ignored for comparison purposes, and Embedded SQL programs pad strings fetched into character arrays. For example, the two strings

```
'Smith'
'Smith   '
```

would be treated as equal in a database created with trailing blanks ignored.

This option is provided for compatibility with the ISO/ANSI SQL standard, which is to ignore trailing blanks in comparisons. The default is that blanks are significant for comparisons.

JCONNECT clause If you wish to use the Sybase jConnect JDBC driver to access system catalog information, you need to install jConnect support. Specify JCONNECT OFF if you wish to exclude the jConnect system objects. You can still use JDBC, as long as you do not access system information.

JAVA clause If you wish to use Java in your database, you must install entries for the Sybase runtime Java classes into the system tables. By default, these entries are installed. You can specify JAVA OFF if you are sure you will not be using Java, to avoid installing these entries.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Adaptive Server Enterprise provides a CREATE DATABASE statement, but with different options.

Example

- The following statement creates a database file named *mydb.db* in the *C:\* directory.

  ```
  CREATE DATABASE 'C:\\mydb'
  TRANSACTION LOG ON
  CASE IGNORE
  PAGE SIZE 1024
  COLLATION '437'
  ENCRYPTED OFF
  BLANK PADDING OFF
  JAVA ON
  JCONNECT OFF
  ```

- The following statement creates a database with no Sybase runtime Java classes. All database operations will execute normally, except for those involving Java classes or objects.

  ```
  CREATE DATABASE 'C:\\nojava'
  JAVA OFF
  ```

CREATE DBSPACE statement

Function

To create a new database file.

Syntax

CREATE DBSPACE *dbspace-name*
  AS *filename*

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

"DROP statement" on page 451
"Using additional dbspaces" on page 615 of the book *Adaptive Server Anywhere User's Guide*

Description

The CREATE DBSPACE statement creates a new database file. When a database is initialized, it is composed of one file. All tables and indexes created are placed in that file. CREATE DBSPACE adds a new file to the database. This file can be on a different disk drive than the root file, which means that the database can be larger than one physical device.

The *dbspace-name* parameter is an internal name for the database file. The *filename* parameter is the actual name of the database file, with a path where necessary.

For each database, there is a limit of twelve dbspaces, including the root file.

A *filename* without an explicit directory is created in the same directory as the main database file. Any relative directory is relative to the main database file. The *filename* is a filename on the server machine. When you are using the database server for NetWare, the *filename* should use a volume name (not a drive letter) when an absolute directory is specified.

Each table, including its associated indexes, is contained entirely within one database file. The IN clause of the CREATE TABLE statement specifies the dbspace into which a table is placed. Tables are put into the root database file by default.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- Create a dbspace called **library** to hold the **LibraryBooks** table and its indexes.

```
CREATE DBSPACE library
AS 'e:\dbfiles\library.db' ;

CREATE TABLE LibraryBooks (
title char(100),
author char(50),
isbn char(30),
) IN library ;
```

CREATE DOMAIN statement

Function

To create a user-defined data type in the database.

Syntax

CREATE { **DOMAIN** | **DATATYPE** } [**AS**] *domain-name data-type*
... [[**NOT**] **NULL**]
... [**DEFAULT** *default-value*]
... [**CHECK** (*condition*)]

Parameters

domain-name:	*identifier*
data-type:	*built-in data type, with precision and scale*

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

"DROP statement" on page 451
"SQL Data Types" on page 219

Description

User-defined data types are aliases for built-in data types, including precision and scale values where applicable. They improve convenience and encourage consistency in the database.

It is recommended that you use CREATE DOMAIN, rather than CREATE DATATYPE, because CREATE DOMAIN is the ANSI/ISO SQL3 term.

User-defined data types can have CHECK conditions and DEFAULT conditions associated with them, and you can indicate whether the data type permits NULL values or not. These conditions are inherited by any column defined on the data type. Any conditions explicitly specified on the column override the data type conditions.

The user who creates a data type is automatically made the owner of that data type. No owner can be specified in the CREATE DATATYPE statement. The user-defined data type name must be unique, and all users can access the data type without using the owner as prefix.

User-defined data types are objects within the database. Their names must conform to the rules for identifiers. User-defined data type names are always case insensitive, as are built-in data type names.

By default, user-defined data types allow NULLs unless the **allow_nulls_by_default** option is set to OFF. In this case, new user-defined data types by default do not allow NULLs. Any column created on a user-defined data type either allows or does not allow NULLs depending on the setting of the user-defined data type at the time the column was created, not on the current setting of the **allow_nulls_by_default** option. Any explicit setting of NULL or NOT NULL in the column definition overrides the user-defined data type setting.

When creating a CHECK condition, you can use a variable name prefixed with the @ sign in the condition. When the data type is used in the definition of a column, such a variable is replaced by the column name. This allows CHECK conditions to be defined on data types and used by columns of any name.

To drop the data type from the database, use the DROP statement. You must be either the owner of the data type or have DBA authority in order to drop a user-defined data type.

Standards and compatibility

- **SQL/92** Intermediate level feature.
- **Sybase** Not supported by Adaptive Server Enterprise. Transact-SQL provides similar functionality using the CREATE DEFAULT and CREATE RULE statements.

Example

- The following statement creates a data type named **address**, which holds a 35-character string, and which may be NULL.

```
CREATE DOMAIN address CHAR( 35 ) NULL
```

- The following statement creates a data type named **id**, which does not allow NULLS, and which is autoincremented by default.

```
CREATE DOMAIN id INT
NOT NULL
DEFAULT AUTOINCREMENT
```

CREATE EXISTING TABLE statement

Function

To create a new proxy table representing an existing object on a remote server.

Syntax

CREATE EXISTING TABLE [*owner*.]*table_name*
  [(*column-definition*, ...)]
  AT *'location-string'*

Parameters

column-definition:
  column-name data-type [**NOT NULL**]

location-string:
   remote-server-name.[*db-name*].[*owner*].*object-name*
  | *remote-server-name*;[*db-name*];[*owner*];*object-name*

Permissions

Must have RESOURCE authority. To create a table for another user, you must have DBA authority.

Supported on Windows 95 and Windows NT only.

Side effects

Automatic commit.

See also

CREATE TABLE statement

Description

The CREATE EXISTING TABLE statement creates a new local, proxy table that maps to a table at an external location. The CREATE EXISTING TABLE statement is a variant of the CREATE TABLE statement. The EXISTING keyword is used with CREATE TABLE to specify that a table already exists remotely and that its metadata is to be imported into Adaptive Server Anywhere. This establishes the remote table as a visible entity to Adaptive Server Anywhere users. Adaptive Server Anywhere verifies that the table exists at the external location before it creates the table.

If the object does not exist (either host data file or remote server object), the statement is rejected with an error message.

Index information from the host data file or remote server table is extracted and used to create rows for the system table sysindexes. This defines indexes and keys in server terms and enables the query optimizer to consider any indexes that may exist on this table.

Referential constraints are passed to the remote location when appropriate.

If column-definitions are not specified, Adaptive Server Anywhere derives the column list from the metadata it obtains from the remote table. If column-definitions are specified, Adaptive Server Anywhere verifies the column-definitions. Column names, data types, lengths, identity property, and null properties are checked for the following:

♦ Column names must match identically (although case is ignored).

- Data types in the CREATE EXISTING TABLE statement must match or be convertible to the data types of the column on the remote location. For example, a local column data type is defined as money, while the remote column data type is numeric.
- Each column's NULL property is checked. If the local column's NULL property is not identical to the remote column's NULL property, a warning message is issued, but the statement is not aborted.
- Each column's length is checked. If the length of char, varchar, binary, varbinary, decimal and numeric columns do not match, a warning message is issued, but the command is not aborted. You may choose to include only a subset of the actual remote column list in your CREATE EXISTING statement.
- AT clause   The AT clause specifies the location of the remote object. The AT clause supports the semicolon (;) as a delimiter. If a semicolon is present anywhere in the location-string string, the semicolon is the field delimiter. If no semicolon is present, a period is the field delimiter. This allows filenames and extensions to be used in the database and owner fields. For example, the following statement maps the table a1 to the MS Access file *mydbfile.mdb*:

  ```
  CREATE EXISTING TABLE a1
  AT 'access;d:\mydbfile.mdb;;a1'
  ```

Standards and compatibility

- **SQL/92**   Entry-level feature.
- **Sybase**   Supported by Open Client/Open Server.

Examples

- Create a proxy table named **blurbs** for the **blurbs** table at the remote server **server_a**.

  ```
  CREATE EXISTING TABLE blurbs
  ( author_id id not null,
  copy text not null)
  AT 'server_a.db1.joe.blurbs'
  ```

- Create a proxy table named **blurbs** for the **blurbs** table at the remote server **server_a**. Adaptive Server Anywhere derives the column list from the metadata it obtains from the remote table.

  ```
  CREATE EXISTING TABLE blurbs
  AT 'server_a.db1.joe.blurbs'
  ```

- Create a proxy table named **rda_employee** for the **employee** table at the Adaptive Server Anywhere remote server **asademo**.

  ```
  CREATE EXISTING TABLE rda_employee
  AT 'asademo..dba.employee'
  ```

CREATE EXTERNLOGIN statement

Function

To assign an alternate login name and password to be used when communicating with a remote server.

Syntax

CREATE EXTERNLOGIN *login-name*
  TO *remote-server*
  REMOTE LOGIN *remote-user*
  [**IDENTIFIED BY** *remote-password*]

Permissions

Only the login-name and the DBA account can add or modify an external login for login-name.

Supported on Windows 95 and Windows NT only.

Side effects

Automatic commit.

See also

"DROP EXTERNLOGIN statement" on page 455

Description

By default, Adaptive Server Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. CREATE EXTERNLOGIN assigns an alternate login name and password to be used when communicating with a remote server. It stores the password internally in encrypted form. The remote_server must be known to the local server by an entry in the sysservers table. For more information, see CREATE SERVER Statement.

Sites with automatic password expiration should plan for periodic updates of passwords for external logins.

CREATE EXTERNLOGIN cannot be used from within a transaction.

login-name specifies the local user login name.

TO clause The TO clause specifies the name of the remote server.

REMOTE LOGIN clause The REMOTE LOGIN clause specifies the corresponding user account on remote-server for the local user login-name.

IDENTIFIED BY clause The IDENTIFIED BY clause specifies remote-password is the password for remote-user

remote-user and *remote-password* must be a valid combination on the node where the remote-server runs.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Open Client/Open Server.

Examples

- Map the local user named **dba** to the user **sa** when connecting to the server **sybase1**.

```
CREATE EXTERNLOGIN dba
TO sybase1
REMOTE LOGIN sa
IDENTIFIED BY monkey
```

CREATE FUNCTION statement

Function

To create a new function in the database.

Syntax

CREATE FUNCTION [*owner*.]*function-name* ([*parameter* , ...])
... **RETURNS** *data-type*
... { **EXTERNAL NAME** *library-call* |
... [**ON EXCEPTION RESUME**]
... *compound-statement* }

Parameters

parameter:
IN *parameter-name data-type*

library-call:
'[*operating-system*:]*function-name@library.dll*; ...'

operating-system:
OS2
| **Windows3X**
| **Windows95**
| **WindowsNT**
| **NetWare**

Permissions

Must have RESOURCE authority.

For external functions, must have DBA authority.

Side effects

Automatic commit.

See also

"DROP statement" on page 451
"BEGIN... END statement" on page 361
"CREATE PROCEDURE statement" on page 403
"RETURN statement" on page 534
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The CREATE FUNCTION statement creates (stores) a user-defined function in the database. A function can be created for another user by specifying an owner name. Subject to permissions, a user-defined function can be used in exactly the same way as other nonaggregate functions.

Parameter names must conform to the rules for other database identifiers such as column names. They must have a valid SQL data type (see "SQL Data Types" on page 219), and must be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the procedure.

A function using the EXTERNAL NAME clause is a wrapper around a call to an external dynamic link library, and is called an external stored procedure. An external stored procedure can have no clauses other than the EXTERNAL NAME clause following the RETURNS clause. For a description of external procedures, see "Calling external libraries from procedures" on page 271 of the book *Adaptive Server Anywhere User's Guide*.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- The following function concatenates a **firstname** string and a **lastname** string.

```
CREATE FUNCTION fullname ( firstname CHAR(30),
                            lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
   DECLARE name CHAR(61) ;
   SET name = firstname || ' ' || lastname ;
   RETURN (name) ;
END
```

The following examples illustrate the use of the **fullname** function.

- Return a full name from two supplied strings:

```
SELECT fullname ('joe','smith')
```

fullname('joe','smith')
joe smith

- List the names of all employees:

```
SELECT fullname (emp_fname, emp_lname)
FROM employee
```

fullname (emp_fname, emp_lname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
Robert Breault
...

CREATE INDEX statement

Function

To create an index on a specified table. Indexes are used to improve database performance.

Syntax

CREATE [**UNIQUE**] **INDEX** *index-name*
  ... **ON** [*owner*.]*table-name*
    ... (*column-name* [**ASC** | **DESC**], ...)
  ... [{ **IN** | **ON** } *dbspace-name*]

Permissions

Must be the owner of the table or have DBA authority.

Side effects

Automatic commit.

See also

"DROP statement" on page 451

Description

The CREATE INDEX statement creates a sorted index on the specified columns of the named table. Indexes are automatically used to improve the performance of queries issued to the database, and to sort queries with an ORDER BY clause. Once an index is created, it is never referenced again except to delete it using the DROP INDEX statement.

UNIQUE constraint The UNIQUE constraint ensures that there will not be two rows in the table with identical values in all the columns in the index.

Ascending or descending sorting Columns are sorted in ascending (increasing) order unless descending (DESC) is explicitly specified. An index will be used for both an ascending and a descending ORDER BY, whether the index was ascending or descending. However, if an ORDER BY is performed with mixed ascending and descending attributes, an index will be used only if the index was created with the same ascending and descending attributes.

Index placement By default, the index is placed in the same database file as its table. You can place the index in a separate database file by specifying a dbspace name in which to put the index. This feature is useful mainly for large databases, to circumvent the limit, on operating systems other than Windows NT, of 2 GB per table.

Notes

- **Index ownership** There is no way of specifying the index owner in the CREATE INDEX statement. Indexes are automatically owned by the owner of the table on which they are defined. The index name must be unique for each owner.
- **No indexes on views** Indexes cannot be created for views.

- **Index name space** The name of each index must be unique for a given table. For databases created previous to version 5.5.01, the condition was more restrictive: that each index name must be unique for a given user ID.
- **Exclusive table use** CREATE INDEX is prevented whenever the statement affects a table currently being used by another connection. CREATE INDEX can be time consuming and the server will not process requests referencing the same table while the statement is being processed.
- **Automatically created indexes** Adaptive Server Anywhere automatically creates indexes for primary keys and for unique constraints. These automatically-created indexes are held in the same database file as the table.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Adaptive Server Enterprise has a more complex CREATE INDEX statement than Adaptive Server Anywhere. While the Adaptive Server Enterprise syntax is permitted in Adaptive Server Anywhere, some clauses and keywords are ignored.

The full syntax for Adaptive Server Enterprise 11.5 is as follows:

```
CREATE  [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]
    ... INDEX index-name
    ... ON [ [ database.]owner.]table_name
                    (column_name [, column_name]...)
    ... [ WITH {
            ... {  FILLFACTOR | MAX_ROWS_PER_PAGE } = x,
               CONSUMERS = x,
            ... IGNORE_DUP_KEY,
            ... SORTED_DATA,
               [ IGNORE_DUP_ROW | ALLOW_DUP_ROW  ]
            } ]
    ... [ ON segment_name ]
```

Adaptive Server Enterprise indexes can be either **clustered** or **nonclustered**. A clustered index almost always retrieves data faster than a nonclustered index. Only one clustered index is permitted per table.

Adaptive Server Anywhere does not support clustered indexes. The CLUSTERED and NONCLUSTERED keywords are allowed by Adaptive Server Anywhere, but no action is taken.

Adaptive Server Anywhere also allows, by ignoring, the following keywords:

- FILLFACTOR
- IGNORE_DUP_KEY

- SORTED_DATA
- IGNORE_DUP_ROW
- ALLOW_DUP_ROW

Physical placement of an index is carried out differently in Adaptive Server Enterprise and Adaptive Server Anywhere. The *ON segment-name* clause is supported in Adaptive Server Anywhere, but *segment-name* refers to a dbspace.

Index names must be unique on a given table for both Adaptive Server Anywhere and Enterprise.

Examples

- Create a two-column index on the employee table.

```
CREATE INDEX employee_name_index
ON employee
( emp_lname, emp_fname )
```

- Create an index on the sales_order_items table for the product ID column.

```
CREATE INDEX item_prod
ON sales_order_items
( prod_id )
```

CREATE MESSAGE statement [T-SQL]

Function

To add a user-defined message to the SYSUSERMESSAGES system table for use by PRINT and RAISERROR calls.

Syntax

CREATE MESSAGE *message-num*
    ... **AS** *'message-text'*

Permissions

Must have resource authority

Side effects

Automatic commit.

See also

"PRINT statement" on page 523
"RAISERROR statement" on page 526

Description

CREATE MESSAGE is provided in Adaptive Server Anywhere as an alternative to the **sp_addmessage** system procedure used in Adaptive Server Enterprise.

The replaceable text in the syntax is as follows:

♦ **message_num** The message number of the message to add. The message number for a user-defined message must be 20000 or greater.

♦ **message_text** The text of the message to add. The maximum length is 255 bytes. PRINT and RAISERROR recognize placeholders in the message text. A single message can contain up to 20 unique placeholders in any order. These placeholders are replaced with the formatted contents of any arguments that follow the message when the text of the message is sent to the client.

  The placeholders are numbered to allow reordering of the arguments when translating a message to a language with a different grammatical structure. A placeholder for an argument appears as "%nn!", a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation mark (!). The integer represents the argument number in the string in the argument list. "%1!" is the first argument in the original version, "%2!" is the second argument, and so on.

There is no parameter corresponding to the language argument for **sp_addmessage**.

Standards and compatibility

♦ **SQL/92** Vendor extension.

♦ **Sybase** The functionality of CREATE MESSAGE is provided by the **sp_addmessage** procedure in Adaptive Server Enterprise.

CREATE PROCEDURE statement

Function To create a procedure in the database.

Syntax

```
CREATE PROCEDURE [ owner.]procedure-name ( [ parameter , ... ] )
    ... { [ RESULT ( result-column , ... ) ]
          [ ON EXCEPTION RESUME ]
           ... compound-statement
        |    EXTERNAL NAME library-call
        | [ DYNAMIC RESULT SETS integer-expression  ]
             EXTERNAL NAME java-call LANGUAGE JAVA
        }
```

Parameters

```
parameter:
     parameter_mode parameter-name data-type [ DEFAULT expression ]
    | SQLCODE
    | SQLSTATE

parameter_mode:
    IN | OUT | INOUT

result-column:
    column-name data-type

library-call:
    '[operating-system:]function-name@library.dll; ...'

operating-system:
     OS2
    | Windows3X
    | Windows95
    | WindowsNT
    | NetWare

java-call:
    '[package-name.]ClassName.methodName method-signature'
```

Permissions Must have RESOURCE authority.

For external procedures, must have DBA authority.

Side effects Automatic commit.

See also "DROP statement" on page 451
"CALL statement" on page 367
"BEGIN... END statement" on page 361
"GRANT statement" on page 484
"CREATE FUNCTION statement" on page 397
"EXECUTE IMMEDIATE statement" on page 464
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The CREATE PROCEDURE statement creates (stores) a procedure in the database. A procedure can be created for another user by specifying a **owner.** A procedure is invoked with a CALL statement

The body of a procedure consists of a compound statement. For information about compound statements, see "BEGIN... END statement" on page 361.

For information about error handling in stored procedures, see "Errors and warnings in procedures and triggers" on page 256 of the book *Adaptive Server Anywhere User's Guide*.

Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type (see "SQL Data Types" on page 219), and must be prefixed by one of the keywords IN, OUT or INOUT. The keywords have the following meanings:

♦ **IN** The parameter is an expression that provides a value to the procedure.

♦ **OUT** The parameter is a variable that could be given a value by the procedure.

♦ **INOUT** The parameter is a variable that provides a value to the procedure, and could be given a new value by the procedure.

When procedures are executed using the CALL statement, not all parameters need to be specified. If a default value is provided in the CREATE PROCEDURE statement, missing parameters are assigned the default values. If no default value is supplied, the parameter is NULL.

SQLSTATE and SQLCODE are special parameters that output the SQLSTATE or SQLCODE value when the procedure ends (they are OUT parameters). Whether or not a SQLSTATE and SQLCODE parameter is specified, the SQLSTATE and SQLCODE special constants can always be checked immediately after a procedure call to test the return status of the procedure.

The SQLSTATE and SQLCODE special constant values are modified by the next SQL statement. Providing SQLSTATE or SQLCODE as procedure arguments allows the return code to be stored in a variable.

Result sets

A procedure that returns result sets ("Returning results from procedures" on page 246 of the book *Adaptive Server Anywhere User's Guide*) may have a RESULT clause. The parenthesized list following the RESULT keyword defines the number of result columns and name and type. This information is returned by the Embedded SQL DESCRIBE or by ODBC **SQLDescribeCol** when a CALL statement is being described. Allowable data types are listed in "SQL Data Types" on page 219.

Some procedures can return different result sets, with different numbers of columns, depending on how they are executed. For example, the following procedure returns two columns under some circumstances, and one in others.

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
   IF formal = 'n' THEN
      SELECT emp_fname
      FROM employee
   ELSE
      SELECT emp_lname,emp_fname
      FROM employee
   END IF
END
```

Procedures with variable result sets must be written without a RESULT clause, or in Transact-SQL. Their use is subject to the following limitations:

- **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, in order to get the proper shape of result set.
- **ODBC** Variable result-set procedures can be used by ODBC applications. The proper description of the variable result sets is carried out by the ODBC driver.
- **Open Client applications** Variable result-set procedures can be used by Open Client applications.

If your procedure does not return variable result sets, you should use a RESULT clause. The presence of this clause prevents ODBC and Open Client applications from re-describing the result set after a cursor is open.

In order to handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the stored procedure definition. To avoid this problem, you can use column aliases in your procedure result set cursor.

External procedures

A procedure using the EXTERNAL NAME clause is a wrapper around a call to an external dynamic link library, and is called an external stored procedure. An external stored procedure can have no clauses other than the EXTERNAL NAME clause following the parameter list. For a description of external procedures, see "Calling external libraries from procedures" on page 271 of the book *Adaptive Server Anywhere User's Guide*.

Java procedures

A procedure that uses EXTERNAL NAME with a LANGUAGE JAVA clause is a wrapper around a Java method.

If the DYNAMIC RESULT SETS clause is not provided, it is assumed that no result sets are being returned from the method.

If the number of parameters is less than the number indicated in the method-signature then the difference must equal the number specified in DYNAMIC RESULT SETS, and each parameter in the method signature in excess of those in the procedure parameter list must have a method signature of [Ljava.sql.ResultSet.

Java method signatures

A Java method signature is compact character representation of the types of the parameters and the type of the return value. The format is...

```
'( [type,...] ) type'
```

... where *type* is as follows:

- **'Z'** boolean
- **'B'** byte
- **'S'** short
- **'I'** int
- **'J'** long
- **'F'** float
- **'D'** double
- **'C'** char
- **'V'** void
- **[type** array of type
- **Lclass-name** object class

For example,

```
double some_method(
  boolean a,
  int b,
  java.math.BigDecimal c,
  byte [][] d,
  java.sql.ResultSet[] d ) {
}
```

would have the following signature:

```
'(ZILjava/math/BigDecimal;[[B[Ljava/sql/ResultSet;)D'
```

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** The Transact-SQL CREATE PROCEDURE statement is different.
- **SQLJ** The syntax extensions for Java result sets are as specified in the proposed SQLJ1 standard.

Example

- The following procedure uses a case statement to classify the results of a query.

```
CREATE PROCEDURE ProductType (IN product_id INT, OUT
type CHAR(10))
BEGIN
   DECLARE prod_name CHAR(20) ;
   SELECT name INTO prod_name FROM "DBA"."product"
   WHERE id = product_id;
   CASE prod_name
   WHEN 'Tee Shirt' THEN
      SET type = 'Shirt'
   WHEN 'Sweatshirt' THEN
      SET type = 'Shirt'
   WHEN 'Baseball Cap' THEN
      SET type = 'Hat'
   WHEN 'Visor' THEN
      SET type = 'Hat'
   WHEN 'Shorts' THEN
      SET type = 'Shorts'
   ELSE
      SET type = 'UNKNOWN'
   END CASE ;
END
```

- The following procedure uses a cursor and loops over the rows of the cursor to return a single value.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany
CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
   FOR SQLSTATE '02000' ;
   DECLARE curThisCust CURSOR FOR
   SELECT company_name, CAST(
   sum(sales_order_items.quantity *
   product.unit_price) AS INTEGER) VALUE
   FROM customer
   LEFT OUTER JOIN sales_order
   LEFT OUTER JOIN sales_order_items
   LEFT OUTER JOIN product
   GROUP BY company_name ;

   DECLARE ThisValue INT ;
   DECLARE ThisCompany CHAR(35) ;
   SET TopValue = 0 ;
   OPEN curThisCust ;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
      INTO ThisCompany, ThisValue ;
      IF SQLSTATE = err_notfound THEN
```

```
            LEAVE CustomerLoop ;
         END IF ;
         IF ThisValue > TopValue THEN
            SET TopValue = ThisValue ;
            SET TopCompany = ThisCompany ;
            END IF ;
      END LOOP CustomerLoop ;
      CLOSE curThisCust ;
   END
```

CREATE PROCEDURE statement [T-SQL]

Function

To create a new procedure in the database in a manner compatible with Adaptive Server Enterprise.

Syntax

The following subset of the Transact-SQL CREATE PROCEDURE statement is supported in Adaptive Server Anywhere.

```
CREATE PROCEDURE [owner.]procedure_name
    ...[ [ ( ] @parameter_name data-type [ = default ] [ OUTPUT ], ... [ ) ]
    ]
    ...[ WITH RECOMPILE ]
    ...AS
    ... statement-list
```

Parameters

procedure-name:	identifier
parameter_name:	identifier

Description

- If the Transact-SQL WITH RECOMPILE optional clause is supplied, it is ignored. Adaptive Server Anywhere always recompiles procedures the first time they are executed after a database is started, and stores the compiled procedure until the database is stopped.
- Groups of procedures are not supported.

The following differences between Transact-SQL and Adaptive Server Anywhere statements (Watcom-SQL) are listed to help those writing in both dialects.

- **Variable names prefixed by @** The "@" sign denotes a Transact-SQL variable name, while Watcom-SQL variables can be any valid identifier, and the @ prefix is optional.
- **Input and output parameters** Watcom-SQL procedure parameters are specified as IN, OUT, or INOUT, while Transact-SQL procedure parameters are INPUT parameters by default or can be specified as OUTPUT. Those parameters that would be declared as INOUT or as OUT in Adaptive Server Anywhere should be declared with OUTPUT in Transact-SQL.
- **Parameter default values** Watcom-SQL procedure parameters are given a default value using the keyword DEFAULT, while Transact-SQL uses an equality sign (=) to provide the default value.
- **Returning result sets** Watcom-SQL uses a RESULT clause to specify returned result sets. In Transact-SQL procedures, the column names or alias names of the first query are returned to the calling environment.

  The following Transact-SQL procedure illustrates how result sets are returned from Transact-SQL stored procedures:

```
CREATE PROCEDURE showdept @deptname varchar(30)
AS
   SELECT employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = @deptname
   AND department.dept_id = employee.dept_id
```

The following is the corresponding Watcom-SQL procedure:

```
CREATE PROCEDURE showdept(in deptname
         varchar(30) )
RESULT ( lastname char(20), firstname char(20))
ON EXCEPTION RESUME
BEGIN
   SELECT employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = deptname
   AND department.dept_id = employee.dept_id
END
```

♦ **Procedure body** The body of a Transact-SQL procedure is a list of Transact-SQL statements prefixed by the AS keyword. The body of a Watcom-SQL procedure is a compound statement, bracketed by BEGIN and END keywords.

Standards and compatibility

♦ **SQL/92** Transact-SQL extension.

♦ **Sybase** Anywhere supports a subset of the Adaptive Server Enterprise CREATE PROCEDURE statement syntax.

CREATE SCHEMA statement

Function

Creates a collection of tables, views, and permissions for a database user.

Syntax

```
CREATE SCHEMA AUTHORIZATION userid
    ... [
        create-table-statement
        | create-view-statement
        | grant-statement
      ],...
```

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

"CREATE TABLE statement" on page 415
"CREATE VIEW statement" on page 430
"GRANT statement" on page 484

Description

The CREATE SCHEMA statement creates a schema. A schema is a collection of tables, views, and their associated permissions.

The *userid* must be the user ID of the current connection. You cannot create a schema for another user.

If any statement contained in the CREATE SCHEMA statement fails, the entire CREATE SCHEMA statement is rolled back.

The CREATE SCHEMA statement is simply a way of collecting together individual CREATE and GRANT statements into one operation. There is no SCHEMA database object created in the database, and to drop the objects you must use individual DROP TABLE or DROP VIEW statements. To revoke permissions, you must use a REVOKE statement for each permission granted.

The individual CREATE or GRANT statements are not separated by statement delimiters. The statement delimiter marks the end of the CREATE SCHEMA statement itself.

The individual CREATE or GRANT statements must be ordered such that the objects are created before permissions are granted on them.

Although you can currently create more than one schema for a user, this is not recommended, and may not be supported in future releases.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Adaptive Server Anywhere does not support the use of REVOKE statements within the CREATE SCHEMA statement, and does not allow its use within Transact-SQL batches or procedures.

Examples

- The following CREATE SCHEMA statement creates a schema consisting of two tables. The statement must be executed by the user ID **sample_user**, who must have RESOURCE authority. If the statement creating table **t2** fails, neither table is created.

```
CREATE SCHEMA AUTHORIZATION sample_user
CREATE TABLE t1 ( id1 INT PRIMARY KEY )
CREATE TABLE t2 ( id2 INT PRIMARY KEY ) ;
```

- The statement delimiter in the following CREATE SCHEMA statement is placed after the first CREATE TABLE statement. As the statement delimiter marks the end of the CREATE SCHEMA statement, the example is interpreted as a two statement batch by the database server. Consequently, if the statement creating table **t2** fails, the table **t1** is still created.

```
CREATE SCHEMA AUTHORIZATION sample_user
CREATE TABLE t1 ( id1 INT PRIMARY KEY ) ;
CREATE TABLE t2 ( id2 INT PRIMARY KEY ) ;
```

CREATE SERVER statement

Function

To add a server to the sysservers table.

Syntax

CREATE SERVER *server-name*
  CLASS '*server-class*'
  USING '*connection-info*'
  [**READ ONLY**]

Parameters

server-class:
  { **asajdbc**
  | **asejdbc**
  | **asaodbc**
  | **aseodbc**
  | **db2odbc**
  | **mssodbc**
  | **oraodbc**
  | **odbc** }

connection-info:
  { *machine-name:port-number*[*/dbname*] | *data-source-name* }

Permissions

Must have RESOURCE authority.

Supported on Windows 95 and Windows NT only.

Side effects

Automatic commit.

See also

"ALTER SERVER statement" on page 349
"DROP SERVER statement" on page 457
"Server Classes for Remote Data Access" on page 761 of the book *Adaptive Server Anywhere User's Guide*

Description

The CREATE SERVER statement defines a remote server from the Adaptive Server Anywhere catalogs.

For more information on server classes and how to configure a server, see "Server Classes for Remote Data Access" on page 761 of the book *Adaptive Server Anywhere User's Guide*.

USING clause If a JDBC-based server class is used, the USING clause is the machine-name:port-number. If an ODBC-based server class is used, the USING clause is the data-source-name. The data-source-name is the user Data Source Name in the Configuration Manager.

READ ONLY The READ ONLY clause specifies that the remote server is a read-only data source. Any update request is rejected by Adaptive Server Anywhere.

connection-info The machine name or IP number is required. By default. Adaptive Server Anywhere uses port 2638. The database name */dbname* is useful when creating a remote server that is of class **asajdbc**. If you do not use this, then the default database is used.

The */dbname* parameter is optional for **asejdbc**. If you don't specify it uses **master**, or you can specify another database by some other means (for example, in the FORWARD TO statement).

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Open Client/Open Server.

Examples

- Create an entry in the sysservers table for the JDBC-based Adaptive Server named ase_prod. Its machine name is banana and port number is 3025.

```
CREATE SERVER ase_prod
CLASS 'asejdbc'
USING 'banana:3025'
```

- Create an entry in the sysservers table for the Oracle server named oracle723. Its user Data Source Name in the Configuration Manager is also oracle723.

```
CREATE SERVER oracle723
CLASS 'oraodbc'
USING 'oracle723'
```

CREATE TABLE statement

Function

To create a new table in the database, and (optionally) to create a table on a remote server.

Syntax

CREATE [**GLOBAL TEMPORARY**] **TABLE** [*owner*.]*table-name*
... ({ *column-definition* [*column-constraint* ...] | *table-constraint* }, ...)
... [{ **IN** | **ON** } *dbspace-name*]
... [**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**]
[**AT** *location-string*]

Parameters

column-definition:
column-name data-type [**NOT NULL**] [**DEFAULT** *default-value*]

column-constraint:
UNIQUE
| **PRIMARY KEY**
| **REFERENCES** *table-name* [(*column-name*)] [*actions*]
| **CHECK** (*condition*)
| **COMPUTE** (*expression*)

default-value:
string
| *global variable*
| *number*
| **AUTOINCREMENT**
| **CURRENT DATE**
| **CURRENT TIME**
| **CURRENT TIMESTAMP**
| **NULL**
| **USER**
| (*constant-expression*)

table-constraint:
UNIQUE (*column-name*, ...)
| **PRIMARY KEY** (*column-name*, ...)
| **CHECK** (*condition*)
| *foreign-key-constraint*

foreign-key-constraint:
[**NOT NULL**] **FOREIGN KEY** [*role-name*] [(*column-name*, ...)]
... **REFERENCES** *table-name* [(*column-name*, ...)]
... [*actions*] [**CHECK ON COMMIT**]

action:
ON { **UPDATE** | **DELETE** }
...{ **CASCADE** | **SET NULL** | **SET DEFAULT** | **RESTRICT** }

location-string:
remote-server-name.[*db-name*].[*owner*].*object-name*
| *remote-server-name*;[*db-name*];[*owner*];*object-name*

Permissions

Must have RESOURCE authority. To create a table for another user, you must have DBA authority.

The AT clause to create proxy tables is supported on Windows 95 and Windows NT only.

Side effects

Automatic commit.

See also

"DROP statement" on page 451
"ALTER TABLE statement" on page 351
"CREATE DBSPACE statement" on page 389
"CREATE EXISTING TABLE statement" on page 393
"SQL Data Types" on page 219
"Creating tables" on page 71 of the book *Adaptive Server Anywhere User's Guide*

Description

The CREATE TABLE statement creates a new table. A table can be created for another user by specifying an owner name. If GLOBAL TEMPORARY is specified, the table is a **temporary table**. Otherwise, the table is a **base table.**

A created temporary table exists in the database like a base table and remains in the database until it is explicitly removed by a DROP TABLE statement. The rows in a temporary table are visible only to the connection that inserted the rows. Multiple connections from the same or different applications can use the same temporary table at the same time, and each connection will see only its own rows. The rows of a temporary table are deleted when the connection ends.

IN clause The IN clause specifies in which database file (dbspace) the table is to be created. If the table is a GLOBAL TEMPORARY table, the IN clause is ignored.

For more information about dbspaces, see "CREATE DBSPACE statement" on page 389.

ON COMMIT clause The ON COMMIT clause is allowed only for temporary tables. By default, the rows of a temporary table are deleted on COMMIT.

The parenthesized list following the CREATE TABLE statement can contain the following clauses in any order:

AT clause The AT clause is used to create a table at the remote location specified by *location-string*. The local table that is created is a proxy table that maps to the remote location. The AT clause supports the semicolon (;) as a delimiter. If a semicolon is present anywhere in the *location-string* string, the semicolon is the field delimiter. If no semicolon is present, a period is the field delimiter. This allows filenames and extensions to be used in the database and owner fields.

For example, the following statement maps the table a1 to the MS Access file *mydbfile.mdb*:

```
CREATE TABLE a1
AT 'access;d:\mydbfile.mdb;;a1'
```

column-name data-type [NOT NULL] [DEFAULT default-value] [column-constraint] Define a column in the table. Allowable data types are described in "SQL Data Types" on page 219. Two columns in the same table cannot have the same name.

If NOT NULL is specified, or if the column is in a UNIQUE or PRIMARY KEY constraint, the column cannot contain any NULL values. If a DEFAULT value is specified, it will be used as the value for the column in any INSERT statement that does not specify a value for the column. If no DEFAULT is specified, it is equivalent to DEFAULT NULL.

When using DEFAULT AUTOINCREMENT, the **data type** must be one of the integer data types, or FLOAT, or DOUBLE. On INSERTs into the table, if a value is not specified for the autoincrement column, a unique value is generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent INSERTs.

Deleting rows does not decrement the autoincrement counter. Gaps created by deleting rows can only be filled by explicit assignment when using an insert. After an explicit insert of a row number less then the maximum, subsequent rows without explicit assignment are still autoincremented with a value of one greater than the previous maximum.

The next value to be used for each column is stored as a long integer (4 bytes). Using values greater than (2**31 - 1), that is, large double or numeric values, may cause wraparound to negative values, and AUTOINCREMENT should not be used in such cases.

For performance reasons, it is highly recommended that DEFAULT AUTOINCREMENT be used only with columns defined as a PRIMARY KEY, with a UNIQUE constraint; or the first column of an index. This allows the maximum value to be found at startup time without scanning the entire table.

Constant expressions that do not reference database objects are allowed in a DEFAULT clause, so functions such as **getdate** or **dateadd** can be used. If the expression is not a function or simple value, it must be enclosed in parentheses.

Column constraints are abbreviations for the corresponding table constraints. For example, the following are equivalent:

```
CREATE TABLE Product (
   product_num integer UNIQUE
)

CREATE TABLE Product (
   product_num integer,
   UNIQUE ( product_num )
)
```

Column constraints are normally used unless the constraint references more than one column in the table. In these cases, a table constraint must be used.

table-constraint Table constraints help ensure the integrity of data in the database. There are four types of integrity constraints:

- **UNIQUE constraint** Identifies one or more columns that uniquely identify each row in the table.
- **PRIMARY KEY constraint** This is the same as a unique constraint, except that a table can have only one primary key constraint. The primary key usually identifies the best identifier for a row. For example, the customer number might be the primary key for the customer table.
- **FOREIGN KEY constraint** This restricts the values for a set of columns to match the values in a primary key or uniqueness constraint of another table. For example, a foreign key constraint could be used to ensure that a customer number in an invoice table corresponds to a customer number in the customer table.
- **CHECK constraint** This allows arbitrary conditions to be verified. For example, a check constraint could be used to ensure that a column called **Sex** only contains the values M or F.

  No row is allowed to fail the condition. If an INSERT or UPDATE statement would cause a row to fail the condition, the operation is not permitted and the effects of the statement are undone.

  When is the change rejected?
  The change is rejected only if the condition is FALSE; in particular, the change is allowed if the condition is UNKNOWN. (See "NULL value" on page 213 and "Search conditions" on page 194 for more information about TRUE, FALSE, and UNKNOWN conditions.)

♦ **COMPUTE clause** When a column is created using a COMPUTE clause, its value in any row is the value of the supplied expression. Columns created with this constraint are read-only columns.

  Any UPDATE statement that attempts to change the value of a computed column does, however, fire any triggers associated with the column.

  The Compute constraint is particularly useful when designing databases using Java class data types. For more information, see "Using computed columns with Java classes" on page 498 of the book *Adaptive Server Anywhere User's Guide*.

If a statement would cause changes to the database that would violate an integrity constraint, the statement is effectively not executed and an error is reported. (*Effectively* means that any changes made by the statement before the error was detected are undone.)

Integrity constraints

column-definition UNIQUE or UNIQUE (column-name, ...) No two rows in the table can have the same values in all the named column(s). A table may have more than one unique constraint.

Unique constraint versus unique index
There is a difference between a **unique constraint** and a **unique index**. Columns in a unique index are allowed to be NULL, while columns in a unique constraint are not. Also, the column referenced by a foreign key can be either a primary key or a column with a unique constraint. Unique indexes cannot be referenced, because they can include multiple NULLs.

column-definition PRIMARY KEY, or PRIMARY KEY (column-name, ...) The primary key for the table will consist of the listed column(s), none of which can contain any NULL values. Each row in the table has a unique primary key value. A table can have only one PRIMARY KEY.

When PRIMARY KEY is followed by a list of columns, the primary key includes the columns in the order in which they are defined in the original CREATE TABLE statement, not the order in which they are listed.

column-definition REFERENCES primary-table-name [(primary-column-name)] The column is a **foreign key** for the primary key or a unique constraint in the primary table. Normally, a foreign key would be for a primary key rather than a unique constraint. If a primary column name is specified, it must match a column in the primary table which is subject to a unique constraint or primary key constraint, and that constraint must consist of only that one column. Otherwise the foreign key references the primary key of the second table.

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

[NOT NULL] FOREIGN KEY [role-name] [(...)] REFERENCES primary-table-name [(...)] The table contains a **foreign key** for the primary key or a unique constraint in another table. Normally, a foreign key would be for a primary key rather than a unique constraint. (In this description, this other table will be called the **primary table**.)

If the primary table column names are not specified, then the primary table columns will be the columns in the table's primary key. If foreign key column names are not specified then the foreign key columns will have the same names as the columns in the primary table. If foreign key column names are specified, then the primary key column names must be specified, and the column names are paired according to position in the lists.

Any foreign key column not explicitly defined will automatically be created with the same data type as the corresponding column in the primary table. These automatically created columns cannot be part of the primary key of the foreign table. Thus, a column used in both a primary key and foreign key of the same table must be explicitly created.

A NULL foreign key means that no row in the primary table corresponds to this row in the foreign table.

The **role name** is the name of the foreign key. The main function of the role name is to distinguish two foreign keys to the same table. If no role name is specified, the role name is assigned as follows:

1. If there is no foreign key with a role name the same as the table name, the table name is assigned as the role name.
2. If the table name is already taken, the role name is the table name concatenated with a zero-padded three-digit number unique to the table.

The referential integrity action defines the action to be taken to maintain foreign key relationships in the database. Whenever a primary key value is changed or deleted from a database table, there may be corresponding foreign key values in other tables that should be modified in some way. You can specify either an ON UPDATE clause, an ON DELETE clause, or both, followed by one of the following actions:

CASCADE When used with ON UPDATE, updates the corresponding foreign keys to match the new primary key value. When used with ON DELETE, deletes the rows from the foreign table that match the deleted primary key.

SET NULL Sets to NULL all the foreign key values that correspond to the updated or deleted primary key.

SET DEFAULT Sets foreign key values that match the updated or deleted primary key value to values specified on the DEFAULT clause of each foreign key column.

RESTRICT Generates an error if an attempt is made to update or delete a primary key value while there are corresponding foreign keys elsewhere in the database. This is the default action.

The CHECK ON COMMIT clause causes the database to wait for a COMMIT before checking the referential integrity of inserts and RESTRICT actions on this foreign key, overriding the setting of the WAIT_FOR_COMMIT database option. The CHECK ON COMMIT clause does not delay CASCADE, SET NULL, or SET DEFAULT actions.

If you use CHECK ON COMMIT with out specifying any actions, then RESTRICT is implied as an action for UPDATE and DELETE.

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

Remote tables

- Foreign key definitions are ignored on remote tables. Foreign key definitions on local tables that refer to remote tables are also ignored. Primary key definitions will be sent to the remote server if the server supports it.
- The COMPUTE clause is not supported for remote tables and will be ignored. You may however perform a CREATE EXISTING TABLE against remote tables that contain COMPUTE clause.

Standards and compatibility

- **SQL/92** Entry level feature.

  The following are vendor extensions:

  - The { **IN** | **ON** } *dbspace-name* clause.
  - The **ON COMMIT** clause
  - Some of the default values.
- **Sybase** Supported by Adaptive Server Enterprise, with some differences.
  - **Temporary tables** You can create a temporary table by preceding the table name in a CREATE TABLE statement with a pound sign (#). In Adaptive Server Anywhere, these are declared temporary tables, which are available only in the current connection. For information, see "DECLARE LOCAL TEMPORARY TABLE statement" on page 441.

- **Physical placement** Physical placement of a table is carried out differently in Adaptive Server Anywhere and in Adaptive Server Enterprise. The **ON** *segment-name* clause supported by Adaptive Server Enterprise is supported in Adaptive Server Anywhere, but *segment-name* refers to a dbspace name.
- **Constraints** Adaptive Server Anywhere does not support named constraints or named defaults, but does support user-defined data types which allow constraint and default definitions to be encapsulated in the data type definition. It also supports explicit defaults and CHECK conditions in the CREATE TABLE statement.
- **NULL default** By default, columns in Adaptive Server Enterprise default to NOT NULL, whereas in Adaptive Server Anywhere the default setting is NULL, to allow NULL values. This setting can be controlled using the **allow_nulls_by_default** option. For information on this option, see "ALLOW_NULLS_BY_DEFAULT option" on page 140. You should explicitly specify NULL or NOT NULL to make your data definition statements transferable.

Examples

- Create a table for a library database to hold book information.

```
CREATE TABLE library_books (
-- NOT NULL is assumed for primary key columns
isbn CHAR(20)      PRIMARY KEY,
copyright_date     DATE,
title              CHAR(100),
author             CHAR(50),
-- column(s) corresponding to primary key of room
-- will be created
FOREIGN KEY location REFERENCES room
)
```

- Create a table for a library database to hold information on borrowed books.

```
CREATE TABLE borrowed_book (
-- Default on insert is that book is borrowed today
date_borrowed DATE NOT NULL DEFAULT CURRENT DATE,
-- date_returned will be NULL until the book is
returned
date_returned       DATE,
book                CHAR(20)
                    REFERENCES library_books (isbn),
-- The check condition is UNKNOWN until
-- the book is returned, which is allowed
CHECK( date_returned >= date_borrowed )
)
```

- Create tables for a sales database to hold order and order item information.

```
CREATE TABLE Orders (
   order_num INTEGER NOT NULL PRIMARY KEY,
   date_ordered DATE,
   name CHAR(80)
) ;
```

```
CREATE TABLE Order_item (
   order_num        INTEGER NOT NULL,
   item_num         SMALLINT NOT NULL,
   PRIMARY KEY (order_num, item_num),
   -- When an order is deleted, delete all of its
   -- items.
   FOREIGN KEY (order_num)
   REFERENCES Orders (order_num)
   ON DELETE CASCADE
)
```

♦ Creates a table named t1 at the remote server SERVER_A and creates a proxy table named t1 that is mapped to the remote table.

```
CREATE TABLE t1
( a  INT,
  b  CHAR(10))
AT 'SERVER_A.db1.joe.t1'
```

CREATE TRIGGER statement

Function

To create a new trigger in the database.

Syntax

```
CREATE TRIGGER trigger-name trigger-time trigger-event [, trigger-event,..]
   ... [ ORDER integer ] ON table-name
   ... [ REFERENCING [ OLD AS old-name ]
                     [ NEW AS new-name ] ]
                     [ REMOTE AS remote-name ] ]
   ... [ FOR EACH { ROW | STATEMENT } ]
   ... [ WHEN ( search-condition ) ]
   ... [ IF UPDATE ( column-name ) THEN
   ... [ { AND | OR } UPDATE ( column-name ) ] ... ]
              ... compound-statement
   ... [ ELSEIF UPDATE ( column-name ) THEN
   ... [ { AND | OR } UPDATE ( column-name ) ] ...
              ...  compound-statement
   ...  END IF ] ]
```

Parameters

trigger-time:
   BEFORE | **AFTER** | **RESOLVE**

trigger-event:
   DELETE | **INSERT** | **UPDATE** | **UPDATE OF** *column-list*

Permissions

Must have RESOURCE authority and have ALTER permissions on the table, or must have DBA authority. CREATE TRIGGER puts a table lock on the table, and thus requires exclusive use of the table.

Side effects

Automatic commit.

See also

"BEGIN... END statement" on page 361
"CREATE PROCEDURE statement" on page 403
"DROP statement" on page 451
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The CREATE TRIGGER statement creates a trigger associated with a table in the database, and stores the trigger in the database.

Triggers can be triggered by one or more of the following events:

- **INSERT** Invoked whenever a new row is inserted into the table associated with the trigger.
- **DELETE** Invoked whenever a row of the associated table is deleted.
- **UPDATE** Invoked whenever a row of the associated table is updated.
- **UPDATE OF column-list** Invoked whenever a row of the associated table is updated and a column in the *column-list* has been modified.

Row and statement-level triggers

The trigger is declared as either a row-level trigger, in which case it executes before or after each row is modified, or as a statement-level trigger, in which case it executes after the entire triggering statement is completed.

Row-level triggers can be defined to execute BEFORE or AFTER the insert, update, or delete. Statement-level triggers execute AFTER the statement. The RESOLVE trigger time is for use with SQL Remote; it fires before row-level UPDATE or UPDATE OF column-lists only.

To declare a trigger as a row-level trigger, use the FOR EACH ROW clause. To declare a trigger as a statement-level trigger, you can either use a FOR EACH STATEMENT clause or omit the FOR EACH clause. For clarity, it is recommended that you enter the FOR EACH STATEMENT clause if declaring a statement-level trigger.

The **WHEN (***search-condition***)** clause can only be used with row level triggers.

Order of firing

Triggers of the same type (insert, update, or delete) that fire at the same time (before, after, or resolve) can use the ORDER clause to determine the order that the triggers are fired.

Referencing deleted and inserted values

The REFERENCING OLD and REFERENCING NEW clauses allow you to refer to the deleted and inserted rows. For the purposes of this clause, an UPDATE is treated as a delete followed by an insert.

The REFERENCING REMOTE clause is for use with SQL Remote. It allows you to refer to the values in the VERIFY clause of an UPDATE statement. It should be used only with RESOLVE UPDATE or RESOLVE UPDATE OF column-list triggers.

The meaning of REFERENCING OLD and REFERENCING NEW differs, depending on whether the trigger is a row-level or a statement-level trigger. For row-level triggers, the REFERENCING OLD clause allows you to refer to the values in a row prior to an update or delete, and the REFERENCING NEW clause allows you to refer to the inserted or updated values. The OLD and NEW rows can be referenced in BEFORE and AFTER triggers. The REFERENCING NEW clause allows you to modify the new row in a BEFORE trigger before the insert or update operation takes place.

For statement-level triggers, the REFERENCING OLD and REFERENCING NEW clauses refer to declared temporary tables holding the old and new values of the rows. The default names for these tables are **deleted** and **inserted**.

The WHEN clause causes the trigger to fire only for rows where the search-condition evaluates to true. The WHEN clause can be used only with row level triggers.

Updating values with the same value

BEFORE UPDATE triggers fire any time an UPDATE occurs on a row, whether or not the new value differs from the old value. AFTER UPDATE triggers fire only if the new value is different from the old value.

Standards and compatibility

- **SQL/92** Persistent stored module feature. Some clauses are vendor extensions.
- **Sybase** This syntax is different to that supported by Adaptive Server Enterprise.

Example

- When a new department head is appointed, update the **manager_id** column for employees in that department.

```
CREATE TRIGGER
tr_manager BEFORE UPDATE OF dept_head_id ON
department
REFERENCING OLD AS old_dept
NEW AS new_dept
FOR EACH ROW
BEGIN
   UPDATE employee
   SET employee.manager_id=new_dept.dept_head_id
   WHERE employee.dept_id=old_dept.dept_id
END
```

CREATE TRIGGER statement [T-SQL]

Function

To create a new procedure in the database in a manner compatible with Adaptive Server Enterprise.

Syntax 1

```
CREATE TRIGGER [owner.]trigger_name
   ... ON [owner.]table_name
   ... FOR { INSERT , UPDATE , DELETE }
   ... AS
   ...        statement-list
```

Syntax 2

```
CREATE TRIGGER [owner.]trigger_name
   ...  ON [owner.]table_name
   ...  FOR {INSERT , UPDATE}
   ...  AS
   ...  [ IF UPDATE ( column_name )
   ...  [ { AND | OR} UPDATE ( column_name ) ] ... ]
   ...        statement-list
   ...  [ IF UPDATE ( column_name )
   ...  [ { AND | OR} UPDATE ( column_name ) ] ... ]
   ...   statement-list
```

Description

- The rows deleted or inserted are held in two declared temporary tables given the default names **deleted**, and **inserted**, as in Adaptive Server Enterprise.
- In Adaptive Server Enterprise, trigger names must be unique in the database. In Adaptive Server Anywhere, trigger names must be unique for a given owner. For compatible databases, you should make trigger names unique in the database.
- Transact-SQL triggers are executed after the triggering statement.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Anywhere supports a subset of the Adaptive Server Enterprise syntax.

CREATE VARIABLE statement

Function To create a SQL variable.

Syntax **CREATE VARIABLE** *identifier data-type*

Permissions None.

Side effects None.

See also "BEGIN... END statement" on page 361
"SQL Data Types" on page 219
"DROP VARIABLE statement" on page 459
"SET statement" on page 546

Description The CREATE VARIABLE statement creates a new variable of the specified data type. The variable contains the NULL value until it is assigned a different value by the SET VARIABLE statement.

A variable can be used in a SQL statement anywhere a column name is allowed. If column name matches the identifier, Adaptive Server Anywhere checks to see if there is a variable that matches and uses its value.

Variables belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE statement. Variables are not visible to other connections. Variables are not affected by COMMIT or ROLLBACK statements.

Variables are useful for creating large text or binary objects for INSERT or UPDATE statements from Embedded SQL programs.

Variables created by CREATE VARIABLE can be used in any SQL statement or in any procedure or trigger.

Local variables in procedures and triggers are declared within a compound statement (see "Using compound statements" on page 239 of the book *Adaptive Server Anywhere User's Guide*).

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- The following code fragment could be used to insert a large text value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
char buffer[5000];
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE VARIABLE hold_blob LONG VARCHAR;
EXEC SQL SET hold_blob = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( buffer, 1, 5000, fp );
   if( size <= 0 ) break;
   /* add data to blob using concatenation
   Note that concatenation works for binary
   data too! */
   EXEC SQL SET hold_blob = hold_blob || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES ( 1,
hold_blob );
EXEC SQL DROP VARIABLE hold_blob;
```

CREATE VIEW statement

Function

To create a view on the database. Views are used to give a different perspective on the data, even though it is not stored that way.

Syntax

```
CREATE VIEW
    ... [ owner.]view-name [( column-name, ... )]
    ... AS select-without-order-by
    ... [ WITH CHECK OPTION ]
```

Permissions

Must have RESOURCE authority and SELECT permission on the tables in the view definition.

Side effects

Automatic commit.

See also

"DROP statement" on page 451
"CREATE TABLE statement" on page 415

Description

The CREATE VIEW statement creates a view with the given name. You can create a view owned by another user by specifying the **owner**. You must have DBA authority to create a view for another user.

A view name can be used in place of a table name in SELECT, DELETE, UPDATE, and INSERT statements. Views, however, do not physically exist in the database as tables. They are derived each time they are used. The view is derived as the result of the SELECT statement specified in the CREATE VIEW statement. Table names used in a view should be qualified by the user ID of the table owner. Otherwise, a different user ID might not be able to find the table or might get the wrong table.

The columns in the view are given the names specified in the column name list. If the column name list is not specified, the view columns are given names from the select list items. In order to use the names from the select list items, each item must be a simple column name or have an alias-name specified (see "SELECT statement" on page 542).

Views can be updated unless the SELECT statement defining the view contains a GROUP BY clause, an aggregate function, or involves a UNION operation. An update to the view causes the underlying table(s) to be updated.

The WITH CHECK OPTION clause rejects any updates and inserts to the view that do not meet the criteria of the views as defined by its SELECT statement.

The SELECT statement must not have an ORDER BY clause on it. It may have a GROUP BY clause and may be a UNION.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Adaptive Server Enterprise.

Examples

- Create a view showing information for male employees only. This view has the same column names as the base table.

```
CREATE VIEW male_employee
AS SELECT *
FROM Employee
WHERE Sex = 'M'
```

- Create a view showing employees and the departments they belong to.

```
CREATE VIEW emp_dept
AS SELECT emp_lname, emp_fname, dept_name
FROM Employee JOIN Department
ON Employee.dept_id = Department.dept_id
```

CREATE WRITEFILE statement

Function

To create a write file for a database.

Syntax

CREATE WRITEFILE *write-file-name*
  ... **FOR DATABASE** *db-file-name*
  ... [**LOG OFF** | **LOG ON** [*log-file-name* [**MIRROR** *mirrorfile-name*]]]

Parameters

write-file-name | *db-file-name* | *log-file-name* | *mirror-file-name* :
  '*file-name*'

Permissions

The permissions required to execute this statement are set on the server command line, using the `-gu` command-line option. The default setting is to require DBA authority.

The account under which the server is running must have write permissions on the directories where files are created.

Not supported on Windows CE.

Side effects

An operating system file is created.

See also

"CREATE DATABASE statement" on page 385
"The Write File utility" on page 121
"Working with write files" on page 618 of the book *Adaptive Server Anywhere User's Guide*

Description

Creates a database write file with the supplied name and attributes.

The file names (*write-file-name*, *db-file-name*, *log-file-name*, *mirror-file-name*) are strings containing operating system file names. As literal strings, they must be enclosed in single quotes. If you specify a path, any backslash characters (\) must be doubled according to the rules for strings in SQL.

If you specify no path, or a relative path, the file is created relative to the current working directory of the server.

You cannot create a write file for a database that is currently loaded.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Adaptive Server Enterprise provides a CREATE DATABASE statement, but with different options.

Example

- The following statement creates a write file..

```
CREATE WRITEFILE 'c:\\sybase\\my_db.wrt'
FOR DATABASE 'c:\\sybase\\my_db.db'
LOG ON 'e:\\logdrive\\my_db.log'
```

DEALLOCATE DESCRIPTOR statement [ESQL]

Function

Frees memory associated with a SQL descriptor area.

Syntax

DEALLOCATE DESCRIPTOR *descriptor-name*

Permissions

None.

Side effects

None.

See also

"ALLOCATE DESCRIPTOR statement" on page 342
"The SQL descriptor area (SQLDA)" on page 45 of the book *Adaptive Server Anywhere Programming Interfaces Guide*
"SET DESCRIPTOR statement" on page 552

Description

Frees all memory associated with a descriptor area, including the data items, indicator variables, and the structure itself.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Open Client/Open Server.

Example

- For an example, see "ALLOCATE DESCRIPTOR statement" on page 342.

Declaration section [ESQL]

Function

To declare host variables in an Embedded SQL program. Host variables are used to exchange data with the database.

Syntax

EXEC SQL BEGIN DECLARE SECTION;
...C declarations
EXEC SQL END DECLARE SECTION;

Permissions

None.

See also

"BEGIN... END statement" on page 361

Description

A declaration section is simply a section of C variable declarations surrounded by the BEGIN DECLARE SECTION and END DECLARE SECTION statements. A declaration section makes the SQL preprocessor aware of C variables that will be used as host variables. Not all C declarations are valid inside a declaration section. See "Using host variables" on page 20 of the book *Adaptive Server Anywhere Programming Interfaces Guide* for more information.

Standards and compatibility

- **SQL/92**
- **Sybase**

Examples

```
EXEC SQL BEGIN DECLARE SECTION;
char *emp_lname, initials[5];
int dept;
EXEC SQL END DECLARE SECTION;
```

DECLARE statement

Function

To declare a SQL variable within a compound statement (BEGIN... END).

Syntax

DECLARE *variable_name data-type*

Description

Variables used in the body of a procedure or trigger can be declared using the DECLARE statement. The variable persists for the duration of the compound statement in which it is declared.

The body of a Watcom-SQL procedure or trigger is a compound statement, and variables must be declared immediately following BEGIN. In a Transact-SQL procedure or trigger, there is no such restriction.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Supported by Adaptive Server Enterprise.
    - To be compatible with Adaptive Server Enterprise, the variable name must be preceded by an @.
    - In Adaptive Server Enterprise, a variable that is declared in a procedure or trigger exists for the duration of the procedure or trigger. In Adaptive Server Anywhere, if a variable is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).

Examples

- The following batch illustrates the use of the DECLARE statement and prints a message on the server window:

```
BEGIN
  DECLARE varname CHAR(61) ;
  SET varname = 'Test name';
  MESSAGE name;
END
```

DECLARE CURSOR statement [ESQL] [SP]

Function To declare a cursor. Cursors are the primary means for manipulating the results of queries.

Syntax

```
DECLARE cursor-name
   ... [ UNIQUE
        | SCROLL
        | NO SCROLL
        | DYNAMIC SCROLL
        | INSENSITIVE ]
   ...  CURSOR FOR statement | CURSOR FOR statement-name
   ... [ FOR UPDATE | FOR READ ONLY ]
```

Parameters

cursor-name:	*identifier*
statement-name:	*identifier* or *host-variable*

Permissions None.

Side effects None.

See also
"PREPARE statement" on page 519
"OPEN statement" on page 511
"EXPLAIN statement" on page 466
"SELECT statement" on page 542
"CALL statement" on page 367

Description The DECLARE CURSOR statement declares a cursor with the specified name for a SELECT statement or a CALL statement.

When a cursor is declared UNIQUE, the query is forced to return all the columns required to uniquely identify each row. Often this will mean ensuring that all of the columns in the primary key or a uniqueness table constraint are returned. Any columns that are required but were not specified will be added.

A DESCRIBE done on a UNIQUE cursor sets the following additional flags in the indicator variables:

- ♦ DT_KEY_COLUMN The column is part of the key for the row
- ♦ DT_HIDDEN_COLUMN The column was added to the query, because it was required to uniquely identify the rows

A cursor declared FOR READ ONLY may not be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default.

A cursor declared NO SCROLL is restricted to FETCH NEXT and FETCH RELATIVE 0 seek operations. A cursor declared SCROLL or DYNAMIC SCROLL can use all formats of the FETCH statement. DYNAMIC SCROLL is the default.

SCROLL cursors behave differently from DYNAMIC SCROLL cursors when a row in the cursor is modified or deleted after the first time the row is read. SCROLL cursors have more predictable behavior when changes happen.

Each row fetched in a SCROLL cursor is remembered. If one of these rows is deleted, either by your program or by another program in a multiuser environment, it creates a "hole" in the cursor. If you fetch the row at this "hole" with a SCROLL cursor, Adaptive Server Anywhere returns the error SQLE_NO_CURRENT_ROW indicating that the row has been deleted, and leaves the cursor positioned on the "hole". (A DYNAMIC SCROLL cursor will just skip the "hole" and retrieve the next row.)

This allows your application to remember row positions within a cursor and be assured that these positions will not change. For example, an application could remember that Cobb is the second row in the cursor for *SELECT * FROM employee*. If the first employee (Whitney) is deleted while the SCROLL cursor is still open, FETCH ABSOLUTE 2 will still position on Cobb while FETCH ABSOLUTE 1 will return SQLE_NO_CURRENT_ROW. Similarly, if the cursor is on Cobb, FETCH PREVIOUS will return SQLE_NO_CURRENT_ROW.

In addition, a fetch on a SCROLL cursor will return the warning SQLE_ROW_UPDATED_WARNING if the row has changed since it was last read. (The warning only happens once; fetching the same row a third time will not produce the warning.) Similarly, an UPDATE (positioned) or DELETE (positioned) statement on a row that has been modified since it was last fetched will return the error SQLE_ROW_UPDATED_SINCE_READ and abort the statement. An application must FETCH the row again before the UPDATE or DELETE will be permitted.

Note that an update to any column will cause the warning/error, even if the column is not referenced by the cursor. For example, a cursor on Surname and Initials would report the update even if only the Birthdate column were modified. These update warning and error conditions will not occur in bulk operations mode (`-b` database server statement line switch) when row locking is disabled. See "Tuning bulk loading of data" on page 287 of the book *Adaptive Server Anywhere User's Guide*.

Adaptive Server Anywhere maintains more information about SCROLL cursors than DYNAMIC SCROLL cursors; thus, DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required. There is no extra overhead in Adaptive Server Anywhere for DYNAMIC SCROLL cursors versus NO SCROLL cursors.

A cursor declared INSENSITIVE has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. FETCHING from an INSENSITIVE cursor does not see the effect of any other INSERT, UPDATE, or DELETE statement, or any other PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on a different cursor. It does see the effect of PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on the same cursor.

INSENSITIVE cursors make it easier to write an application that deals with cursors, since you only have to worry about changes you make explicitly to the cursor; you do not have to worry about actions taken by other users or by other parts of your application.

INSENSITIVE cursors can be expensive if the cursor is on a lot of rows. Also, INSENSITIVE cursors are not affected by ROLLBACK or ROLLBACK TO SAVEPOINT; the ROLLBACK is not an operation on the cursor that changes the cursor contents.

INSENSITIVE cursors meet the ODBC requirements for static cursors.

Embedded SQL

Statements are named using the PREPARE statement. Cursors can be declared only for a prepared SELECT or CALL.

The DECLARE CURSOR statement does not generate any C code.

Cursor-name is a string supplied by the programmer.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Open Client/Open Server.

Examples

- The following example illustrates how to declare a scroll cursor in Embedded SQL:

```
EXEC SQL DECLARE cur_employee SCROLL CURSOR
FOR SELECT * FROM employee ;
```

- The following example illustrates how to declare a cursor for a prepared statement in Embedded SQL:

```
EXEC SQL PREPARE employee_statement
FROM 'SELECT emp_lname FROM employee' ;
EXEC SQL DECLARE cur_employee CURSOR
FOR employee_statement ;
```

- The following example illustrates the use of cursors in a stored procedure:

```
BEGIN
  DECLARE cur_employee CURSOR FOR
      SELECT emp_lname
      FROM employee ;
  DECLARE name CHAR(40) ;
  OPEN cur_employee;
  LOOP
    FETCH NEXT cur_employee INTO name ;
    ...
  END LOOP
  CLOSE cur_employee;
END
```

DECLARE CURSOR statement [T-SQL]

Function

To declare a cursor in a manner compatible with Adaptive Server Enterprise.

Syntax

DECLARE *cursor-name*
  ... **CURSOR FOR** *select-statement*
  ...[**FOR** { **READ ONLY** | **UPDATE** }]

Permissions

None.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436

Description

Adaptive Server Anywhere supports a DECLARE CURSOR syntax that is not supported in Enterprise. For information on the full DECLARE CURSOR syntax, see "DECLARE CURSOR statement" on page 436.

This section describes the overlap between the Adaptive Server Anywhere and Enterprise flavors of DECLARE CURSOR.

Standards and compatibility

- **SQL/92** Entry-level compliant. The FOR UPDATE and FOR READ ONLY options are Transact-SQL extensions.
- **Sybase** There are some features of the Adaptive Server Enterprise DECLARE CURSOR statement that are not supported in Adaptive Server Anywhere.
  - Adaptive Server Enterprise supports cursors opened for update of a list of columns from the tables specified in the *select-statement*. This is not supported in Adaptive Server Anywhere.
  - In the Watcom-SQL dialect, a DECLARE CURSOR statement in a procedure, trigger, or batch must immediately follow the BEGIN keyword. In the Transact-SQL dialect, there is no such restriction.
  - In Adaptive Server Enterprise, when a cursor is declared in a procedure, trigger, or batch, it exists for the duration of the procedure, trigger, or batch. In Adaptive Server Anywhere, if a cursor is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).
  - CURSOR type and CURSOR FOR statement name are not supported in Adaptive Server Anywhere.

DECLARE LOCAL TEMPORARY TABLE statement

Function

To declare a local temporary table.

Syntax

DECLARE LOCAL TEMPORARY TABLE *table-name*
... ({ *column-definition* [*column-constraint* ...] | *table-constraint* }, ...)
... [**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**]

Permissions

None.

Side effects

None.

See also

"CREATE TABLE statement" on page 415
"Using compound statements" on page 239 of the book *Adaptive Server Anywhere User's Guide*

Description

The DECLARE LOCAL TEMPORARY TABLE statement declares a temporary table. See "CREATE TABLE statement" on page 415 for definitions of **column-definition**, **column-constraint**, and **table-constraint**.

Declared local temporary tables within compound statements exist within the compound statement. (See "Using compound statements" on page 239 of the book *Adaptive Server Anywhere User's Guide*). Otherwise, the declared local temporary table exists until the end of the connection.

By default, the rows of a temporary table are deleted on COMMIT.

Standards and compatibility

♦ **SQL/92** Conforms to the SQL/92 standard.

♦ **Sybase** Adaptive Server Enterprise does not support DECLARE TEMPORARY TABLE.

Examples

♦ The following example illustrates how to declare a temporary table in Embedded SQL:

```
EXEC SQL DECLARE LOCAL TEMPORARY TABLE MyTable (
  number INT
   );
```

♦ The following example illustrates how to declare a temporary table in a stored procedure:

```
BEGIN
  DECLARE LOCAL TEMPORARY TABLE TempTab (
    number INT
  );
  ...
END
```

DELETE statement

Function

To delete rows from the database.

Syntax

DELETE [**FROM**] [*owner*.]*table-name*
... [**FROM** *table-list*]
... [**WHERE** *search-condition*]

Permissions

Must have DELETE permission on the table.

Side effects

None.

See also

"TRUNCATE TABLE statement" on page 568
"INSERT statement" on page 498
"INPUT statement" on page 494
"FROM clause" on page 476

Description

The DELETE statement deletes all the rows from the named table that satisfy the search condition. If no WHERE clause is specified, all rows from the named table are deleted.

The DELETE statement can be used on views, provided the SELECT statement defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

For a full description of the FROM clause and joins, see "FROM clause" on page 476.

The optional second FROM clause in the DELETE statement allows rows to be deleted based on joins. If the second FROM clause is present, the WHERE clause qualifies the rows of this second FROM clause. Rows are deleted from the table name given in the first FROM clause.

Correlation name resolution

The following statement illustrates a potential ambiguity in table names in DELETE statements with two FROM clauses that use correlation names:

```
DELETE
FROM table_1
FROM table_1 AS alias_1, table_2 AS alias_2
WHERE ...
```

The table **table_1** is identified without a correlation name in the first FROM clause, but with a correlation name in the second FROM clause. In this case, **table_1** in the first clause is identified with **alias_1** in the second clause—there is only one instance of **table_1** in this statement.

This is an exception to the general rule that where a table is identified with a correlation name and without a correlation name in the same statement, two instances of the table are considered.

Consider the following example:

```
DELETE
FROM table_1
FROM table_1 AS alias_1, table_1 AS alias_2
WHERE ...
```

In this case, there are two instances of **table_1** in the second FROM clause. In this case, there is no way of identifying which instance the first FROM clause should be identified with. The usual rules of correlation names apply, and **table_1** in the first FROM clause is identified with neither instance in the second clause: there are three instances of **table_1** in the statement.

Internally, PowerBuilder processes DELETE, INSERT, and UPDATE statements the same way. PowerBuilder inspects them for any PowerBuilder variable references and replaces all references with a constant that conforms to Adaptive Server Anywhere rules for the data type.

Standards and compatibility

- **SQL/92** Entry level compliant. The use of more than one table in the FROM clause is a vendor extension.
- **Sybase** Supported by Adaptive Server Enterprise, including the vendor extension.

Examples

- Remove employee 105 from the database.

  ```
  DELETE
  FROM employee
  WHERE emp_id = 105
  ```

- Remove all data prior to 1993 from the **fin_data** table.

  ```
  DELETE
  FROM fin_data
  WHERE year < 1993
  ```

- Remove all names from the **contact** table if they are already present in the **customer** table.

  ```
  DELETE
  FROM contact
  FROM contact, customer
  WHERE contact.last_name = customer.lname
  AND contact.first_name = customer.fname
  ```

DELETE (positioned) statement [ESQL] [SP]

Function To delete the data at the current location of a cursor.

Syntax

DELETE [**FROM** *table-spec*]
...**WHERE CURRENT OF** *cursor-name*

Parameters

cursor-name: *identifier* or *host-variable*

table-spec: [*owner*.]*correlation-name*

owner: *identifier*

Permissions Must have DELETE permission on tables used in the cursor.

Side effects None.

See also

"UPDATE statement" on page 572
"UPDATE (positioned) statement" on page 575
"INSERT statement" on page 498
"PUT statement" on page 524

Description This form of the DELETE statement deletes the current row of the specified cursor. The current row is defined to be the last row fetched from the cursor.

Name resolution

The table from which rows are deleted is determined as follows:

- If no FROM clause is included, the cursor must be on a single table only.
- If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.
- If a FROM clause is included, and no table owner is specified, the table-spec value is first matched against any correlation names.
  - If a correlation name exists, the *table-spec* value is identified with the correlation name.
  - If a correlation name does not exist, the *table-spec* value must be unambiguously identifiable as a table name in the cursor.
- If a FROM clause is included, and a table owner is specified, the *table-spec* value must be unambiguously identifiable as a table name in the cursor.
- The positioned DELETE statement can be used on a cursor open on a view as long as the view is updatable.

Standards and compatibility

- **SQL/92** Entry level feature.

- **Sybase** Embedded SQL use is supported by Open Client/Open Server, and procedure and trigger use is supported only in Adaptive Server Anywhere.

Example

- The following statement removes the current row from the database.

```
DELETE
WHERE CURRENT OF cur_employee
```

DESCRIBE statement [ESQL]

Function

To get information about the host variables required to store data retrieved from the database, or host variables used to pass data to the database.

Syntax

```
DESCRIBE
    ... [ USER TYPES ]
    ... [ ALL | BIND VARIABLES FOR |
          INPUT | OUTPUT | SELECT LIST FOR ]
    ... [ LONG NAMES [long-name-spec ] | WITH VARIABLE RESULT ]
    ... [ FOR ] { statement-name | CURSOR cursor-name }
    ... INTO sqlda-name
```

Parameters

long-name-spec:
OWNER.TABLE.COLUMN | TABLE.COLUMN | COLUMN

statement-name:	*identifier,* or *host-variable*
cursor-name:	*declared cursor*
sqlda-name:	*identifier*

Permissions

None.

Side effects

None.

See also

"ALLOCATE DESCRIPTOR statement" on page 342
"DECLARE CURSOR statement" on page 436
"OPEN statement" on page 511
"PREPARE statement" on page 519

Description

The DESCRIBE statement sets up the named SQLDA to describe either the OUTPUT (equivalently SELECT LIST) or the INPUT (BIND VARIABLES) for the named statement.

In the INPUT case, DESCRIBE BIND VARIABLES does not set up the data types in the SQLDA: this needs to be done by the application. The ALL keyword allows you to describe INPUT and OUTPUT in one SQLDA.

If you specify a statement name, the statement must have been previously prepared using the PREPARE statement with the same statement name and the SQLDA must have been previously allocated (see the "ALLOCATE DESCRIPTOR statement" on page 342).

If you specify a cursor name, the cursor must have been previously declared and opened. The default action is to describe the OUTPUT. Only SELECT statements and CALL statements have OUTPUT. A DESCRIBE OUTPUT on any other statement will indicate no output by setting the **sqld** field of the SQLDA to zero.

USER TYPES A DESCRIBE statement with the USER TYPES clause returns information about user-defined data types of a column. Typically, such a DESCRIBE will be done when a previous DESCRIBE returns an indicator of DT_HAS_USERTYPE_INFO.

The information returned is the same as for a DESCRIBE without the USER TYPES keywords, except that the **sqlname** field holds the name of the user-defined data type, instead of the name of the column.

If the DESCRIBE uses the LONG NAMES clause, the **sqldata** field holds this information.

SELECT The DESCRIBE OUTPUT statement fills in the data type and length for each select list item in the SQLDA. The name field is also filled in with a name for the select list item. If an alias is specified for a select list item, the name will be that alias. Otherwise, the name will be derived from the select list item: if the item is a simple column name, it will be used; otherwise, a substring of the expression will be used. DESCRIBE will also put the number of select list items in the **sqld** field of the SQLDA.

If the statement being described is a UNION of two or more SELECT statements, the column names returned for DESCRIBE OUTPUT are the same column names which would be returned for the first SELECT statement.

CALL The DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each INOUT or OUT parameter in the procedure. DESCRIBE OUTPUT will also put the number of INOUT or OUT parameters in the **sqld** field of the SQLDA.

CALL (result set) The DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each RESULT column in the procedure definition. DESCRIBE OUTPUT will also put the number of result columns in the **sqld** field of the SQLDA.

A **bind variable** is a value supplied by the application when the database executes the statements. Bind variables can be considered parameters to the statement. DESCRIBE INPUT will fill in the name fields in the SQLDA with the bind variable names. DESCRIBE INPUT will also put the number of bind variables in the **sqld** field of the SQLDA.

DESCRIBE uses the indicator variables in the SQLDA to provide additional information. DT_PROCEDURE_IN and DT_PROCEDURE_OUT are bits that are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns will have both bits clear. After a describe OUTPUT, these bits can be used to distinguish between statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE). DESCRIBE INPUT only sets DT_PROCEDURE_IN and DT_PROCEDURE_OUT appropriately when a bind variable is an argument to a CALL statement; bind variables within an expression that is an argument in a CALL statement will not set the bits.

DESCRIBE ALL allows you to describe INPUT and OUTPUT with one request to the database server. This has a performance benefit in a multi-user environment. The INPUT information will be filled in the SQLDA first, followed by the OUTPUT information. The **sqld** field contains the total number of INPUT and OUTPUT variables. The DT_DESCRIBE_INPUT bit in the indicator variable is set for INPUT variables and clear for OUTPUT variables.

Retrieving long column names

The LONG NAMES clause is provided to retrieve column names for a statement or cursor. Without this clause, there is a 29-character limit on the length of column names; with the clause, names of an arbitrary length are supported.

If LONG NAMES is used, the long names are placed into the SQLDATA field of the SQLDA, as if you were fetching from a cursor. None of the other fields (SQLLEN, SQLTYPE, and so on) are filled in. The SQLDA must be set up like a FETCH SQLDA: it must contain one entry for each column, and the entry must be a string type.

The default specification for the long names is TABLE.COLUMN.

Describing variable result sets

The WITH VARIABLE RESULT statement is used to describe procedures that may have more than one result set, with different numbers or types of columns.

If WITH VARIABLE RESULT is used, the database server sets the SQLCOUNT value after the DESCRIBE statement to one of the following values:

♦ **0** The result set may change. The procedure call should be described again following each OPEN statement.

♦ **1** The result set is fixed. No redescribing is required.

For more information on the use of the SQLDA structure, see "The SQL descriptor area (SQLDA)" on page 45 of the book *Adaptive Server Anywhere Programming Interfaces Guide.*

Standards and compatibility

- **SQL/92** Part of the SQL/92 standard. Some clauses are vendor extensions.
- **Sybase** Some clauses supported by Open Client/Open Server.

Example

- The following example shows how to use the DESCRIBE statement:

```
sqlda = alloc_sqlda( 3 );
EXEC SQL DESCRIBE OUTPUT
  FOR employee_statement
  INTO sqlda;
if( sqlda->sqld  >  sqlda->sqln ) {
  actual_size = sqlda->sqld;
  free_sqlda( sqlda );
  sqlda = alloc_sqlda( actual_size );
  EXEC SQL DESCRIBE OUTPUT
    FOR employee_statement
    INTO sqlda;
}
```

DISCONNECT statement [ESQL][ISQL]

Function

To drop a connection with the database.

Syntax

DISCONNECT
    connection-name
  | [**CURRENT**]
  | **ALL**

Parameters

connection-name:*identifier*, *string,* or *host-variable*.

Permissions

None.

Side effects

None.

See also

"CONNECT statement" on page 381
"SET CONNECTION statement" on page 551

Description

The DISCONNECT statement drops a connection with the database server and releases all resources used by it. If the connection to be dropped was named on the CONNECT statement, the name can be specified. Specifying ALL will drop all of the application's connections to all database environments. CURRENT is the default, and will drop the current connection.

An implicit ROLLBACK is executed on connections that are dropped.

Standards and compatibility

- **SQL/92** Intermediate level feature.
- **Sybase** Supported by Open Client/Open Server.

Examples

- The following statement shows how to use DISCONNECT in Embedded SQL:

```
EXEC SQL DISCONNECT :conn_name
```

- The following statement shows how to use DISCONNECT from Interactive SQL to disconnect all connections:

```
DISCONNECT ALL
```

DROP statement

Function

To remove objects from the database.

Syntax

DROP
  DATABASE *file-name*
 | { **DATATYPE** | **DOMAIN** } *datatype-name*
 | **DBSPACE** *dbspace-name*
 | **FUNCTION** [*owner*.]*function-name*
 | **INDEX** [[*owner*].*table-name*.]*index-name*
 | **PROCEDURE** [*owner*.]*procedure-name*
 | **TABLE** [*owner*.]*table-name*
 | **TRIGGER** [[*owner*.]*table-name*.]*trigger-name*
 | **VIEW** [*owner*.]*view-name*

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP statement.

For DROP DBSPACE, you must be the only connection to the database.

A user with ALTER permissions on the table can execute DROP TRIGGER.

A user with REFERENCES permissions on the table can execute DROP INDEX.

Global temporary tables cannot be dropped unless all users that have referenced the temporary table have disconnected.

Side effects

Automatic commit. Clears the Data window in Interactive SQL. DROP TABLE and DROP INDEX close all cursors for the current connection.

Local temporary tables is an exception; no commit is performed when one is dropped.

See also

"CREATE DATABASE statement" on page 385
"CREATE DOMAIN statement" on page 391
"CREATE INDEX statement" on page 399
"CREATE FUNCTION statement" on page 397
"CREATE PROCEDURE statement" on page 403
"CREATE TABLE statement" on page 415
"CREATE TRIGGER statement" on page 424
"CREATE VIEW statement" on page 430

Description

The DROP statement removes the definition of the indicated database structure. If the structure is a dbspace, all tables in that dbspace must be dropped prior to dropping the dbspace. If the structure is a table, all data in the table is automatically deleted as part of the dropping process. Also, all indexes and keys for the table are dropped by the DROP TABLE statement.

DROP TABLE, DROP INDEX, and DROP DBSPACE are prevented whenever the statement affects a table that is currently being used by another connection.

DROP PROCEDURE and DROP FUNCTION are prevented when the procedure or function is in use by another connection.

DROP DATATYPE is prevented if the data type is used in a table. You must change data types on all columns defined on the user-defined data type in order to drop the data type. It is recommended that you use DROP DOMAIN rather than DROP DATATYPE, as DROP DOMAIN is the syntax used in the ANSI/ISO SQL3 draft.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Adaptive Server Enterprise for those objects that exist in Adaptive Server Enterprise.

Examples

- Drop the department table from the database.

  ```
  DROP TABLE department
  ```

- Drop the **emp_dept** view from the database.

  ```
  DROP VIEW emp_dept
  ```

DROP DATABASE statement

Function

To delete a database file and the associated transaction log.

Syntax

DROP DATABASE *file-name*

Permissions

Required permissions are set using the database server `-gu` command-line option. The default setting is to require DBA authority.

The file must not be in use in order to be deleted.

Not supported on Windows CE.

Side effects

In addition to deleting the database file from disk, the associated transaction log file is deleted.

See also

"CREATE DATABASE statement" on page 385

Description

The DROP DATABASE statement physically deletes the database file from disk. The statement can be used in SQL scripts that create databases, to make sure there is no database file in the place where it is to be created.

If the file does not exist, an error is generated.

Standards and compatibility

♦ **SQL/92** Vendor extension.

♦ **Sybase** Not supported by Adaptive Server Enterprise.

Examples

♦ Drop the database *temp.db*, in the *C:\temp* directory..

```
DROP DATABASE 'c:\temp\temp.db'
```

DROP CONNECTION statement

Function To drop a connection to the database, belonging to any user.

Syntax **DROP CONNECTION** *connection-id*

Permissions Must have DBA authority.

Side effects None.

See also "CONNECT statement" on page 381

Description The DROP CONNECTION statement disconnects a user from the database by dropping the connection to the database.

You can obtain the *connection-id* by using the **connection_property** function to request the connection number. The following statement returns the connection ID of the current connection:

```
SELECT connection_property( 'number' )
```

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

- ♦ The following statement drops the connection with ID number 4.

```
DROP CONNECTION 4
```

DROP EXTERNLOGIN statement

Function

To drop an external login from the Adaptive Server Anywhere catalogs.

Syntax

DROP EXTERNLOGIN *login-name*
 TO *remote-server*

Permissions

Only the login-name and the DBA account can delete an external login for login-name.

Side effects

Automatic commit.

See also

"CREATE EXTERNLOGIN statement" on page 395

Description

DROP EXTERNLOGIN deletes an external login from the Adaptive Server Anywhere catalogs.

login-name specifies the local user login name

TO clause The TO clause specifies the name of the remote server. The local user's alternate login name and password for that server is the external login that is deleted.

Standards and compatibility

- SQL/92 Entry-level feature.
- Sybase Supported by Open Client/Open Server.

Example

```
DROP EXTERNLOGIN dba TO sybase1
```

DROP OPTIMIZER STATISTICS statement

Function
To reset optimizer statistics.

Syntax
DROP OPTIMIZER STATISTICS

Permissions
Must have DBA authority.

Side effects
None.

Description
The DROP OPTIMIZER STATISTICS statement resets optimizer statistics. The query optimizer maintains statistics as it evaluates queries, and uses these statistics to make better decisions when optimizing subsequent queries. These statistics are stored permanently in the database. The DROP OPTIMIZER STATISTICS statement resets these statistics to default values. This statement is most useful when first learning about the optimizer. It can help you understand the process used by the optimizer.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not supported by Adaptive Server Enterprise.

DROP SERVER statement

Function To drop a remote server from the Adaptive Server Anywhere catalogs.

Syntax **DROP SERVER** *server-name*

Permissions Only the DBA account can delete a remote server.

Not supported on Windows CE.

Side effects Automatic commit.

See also "CREATE SERVER statement" on page 413

Description DROP SERVER deletes a remote server from the Adaptive Server Anywhere catalogs. You must drop all the proxy tables that have been defined for the remote server before this statement will succeed.

Standards and compatibility

- SQL/92 Entry-level feature.
- Sybase Supported by Open Client/Open Server.

Example

```
DROP SERVER ase_prod
```

DROP STATEMENT statement [ESQL]

Function To free statement resources.

Syntax **DROP STATEMENT** [*owner*.]*statement-name*

Parameters *statement-name*: *identifier* or *host-variable*

Permissions Must have prepared the statement.

Side effects None.

See also "PREPARE statement" on page 519

Description The DROP STATEMENT statement frees resources used by the named prepared statement. These resources are allocated by a successful PREPARE statement, and are normally not freed until the database connection is released.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Open Client/Open Server

Example

- The following are examples of DROP STATEMENT use:

```
EXEC SQL DROP STATEMENT S1;

EXEC SQL DROP STATEMENT :stmt;
```

DROP VARIABLE statement

Function	To eliminate a SQL variable.
Syntax	**DROP VARIABLE** *identifier*
Permissions	None.
Side effects	None.
See also	"CREATE VARIABLE statement" on page 428 "SET statement" on page 546
Description	The DROP VARIABLE statement eliminates a SQL variable that was previously created using the CREATE VARIABLE statement. Variables will be automatically eliminated when the database connection is released. Because, variables are often used for large objects, eliminating them after use may free up significant resources (primarily disk space).
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **Sybase** Not supported in Adaptive Server Enterprise.

EXECUTE statement [ESQL]

Function

To execute a SQL statement.

Syntax 1

```
EXECUTE statement-name
    ...  [ USING DESCRIPTOR sqlda-name
         | USING host-variable-list ]

    ...  [ INTO DESCRIPTOR into-sqlda-name
         | INTO into-host-variable-list ]

    ...  [ ARRAY :nnn ]
```

Syntax 2

EXECUTE IMMEDIATE *statement*

Parameters

statement-name:	*identifier or host-variable*
sqlda-name:	*identifier*
into-sqlda-name:	*identifier*
statement:	*string* or *host-variable*

Permissions

Permissions are checked on the statement being executed.

Side effects

None.

See also

"PREPARE statement" on page 519
"DECLARE CURSOR statement" on page 436

Description

Format 1 executes the named dynamic statement, which was previously prepared. If the dynamic statement contains host variable place holders which supply information for the request (bind variables), either the **sqlda-name** must specify a C variable which is a pointer to an SQLDA containing enough descriptors for all of the bind variables occurring in the statement, or the bind variables must be supplied in the **host-variable-list**.

The optional ARRAY clause can be used with prepared INSERT statements to allow wide inserts, which insert more than one row at a time and which may improve performance. The value **nnn** is the number of rows to be inserted. The SQLDA must contain **nnn * (columns per row)** variables. The first row is placed in SQLDA variables 0 to **(columns per row)-**1, and so on.

OUTPUT from a SELECT statement or a CALL statement is put into either the variables in the variable list or the program data areas described by the named SQLDA. The correspondence is one-to-one from the OUTPUT (selection list or parameters) to either the host variable list or the SQLDA descriptor array.

If EXECUTE is used with an INSERT statement, the inserted row is returned in the second descriptor. For example, when using auto-increment primary keys or BEFORE INSERT triggers that generate primary key values, the EXECUTE statement provides a mechanism to re-fetch the row immediately and determine the primary key value that was assigned to the row. The same thing can be achieved by using @@identity with auto-increment keys.

Format 2 is a short form to PREPARE and EXECUTE a statement that does not contain bind variables or output. The SQL statement contained in the string or host-variable is immediately executed.

The EXECUTE statement can be used for any SQL statement that can be prepared. Cursors are used for SELECT statements or CALL statements that return many rows from the database (see "Cursors in Embedded SQL" on page 33 of the book *Adaptive Server Anywhere Programming Interfaces Guide*).

After successful execution of an INSERT, UPDATE or DELETE statement, the *sqlerrd[2]* field of the SQLCA (SQLCOUNT) is filled in with the number of rows affected by the operation.

Standards and compatibility

- **SQL/92** Intermediate level feature.
- **Sybase** Supported in Open Client/Open Server.

Examples

- Execute a DELETE.

```
EXEC SQL EXECUTE IMMEDIATE
'DELETE FROM employee WHERE emp_id = 105';
```

- Execute a prepared DELETE statement.

```
EXEC SQL PREPARE del_stmt FROM
'DELETE FROM employee WHERE emp_id = :a';
EXEC SQL EXECUTE del_stmt USING :employee_number;
```

- Execute a prepared query.

```
EXEC SQL PREPARE sel1 FROM
'SELECT emp_lname FROM employee WHERE emp_id = :a';
EXEC SQL EXECUTE sel1 USING :employee_number INTO
:emp_lname;
```

EXECUTE statement [T-SQL]

Function

To invoke a procedure, as an Adaptive Server Enterprise-compatible alternative to the CALL statement.

Syntax

EXECUTE [*@return_status* =] [*creator*.]*procedure_name*
... | [*@parameter-name* =] *expression*,
| [*@parameter-name* =] *@variable* [*output*] |, ...

Authorization

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

"CALL statement" on page 367

Description

The EXECUTE statement executes a stored procedure, optionally supplying procedure parameters and retrieving output values and return status information.

The EXECUTE statement is implemented for Transact-SQL compatibility, but can be used in either Transact-SQL or Watcom-SQL batches and procedures.

Example

♦ The following demonstration procedure is used to illustrate the EXECUTE statement.

```
CREATE PROCEDURE p1( @var INTEGER = 54 )
AS
PRINT 'on input @var = %1.', @var
DECLARE @intvar integer
SELECT @intvar=123
SELECT @var=@intvar
PRINT 'on exit @var = %1.', @var
```

♦ The following statement executes the procedure, supplying the input value of 23 for the parameter. If you are connected from an Open Client application, the PRINT messages are displayed on the client window. If you are connected from an ODBC or Embedded SQL application, the messages are displayed on the database server window.

```
EXECUTE p1 23
```

♦ The following is an alternative way of executing the procedure, which is useful if there are several parameters.

```
EXECUTE p1 @var = 23
```

♦ The following statement executes the procedure, using the default value for the parameter

```
EXECUTE p1
```

- The following statement executes the procedure, and stores the return value in a variable for checking return status.

```
EXECUTE @status = p1 23
```

EXECUTE IMMEDIATE statement [ESQL] [SP]

Function

To enable dynamically constructed statements to be executed from within a procedure.

Syntax 1

EXECUTE IMMEDIATE *string-expression*

Syntax 2

EXECUTE (*string-expression*)

Permissions

None. The statement is executed with the permissions of the owner of the procedure, not with the permissions of the user who calls the procedure.

Side effects

None. However, if the statement is a data definition statement with an automatic commit as a side effect, that commit does take place.

See also

"CREATE PROCEDURE statement" on page 403
"BEGIN... END statement" on page 361

Description

The EXECUTE IMMEDIATE statement extends the range of statements that can be executed from within procedures and triggers. It lets you execute dynamically prepared statements, such as statements that are constructed using the parameters passed in to a procedure.

Literal strings in the statement must be enclosed in single quotes, and the statement must be on a single line.

Standards and compatibility

- ♦ **SQL/92** Intermediate level feature.
- ♦ **Sybase** Supported in Open Client/Open Server.

Example

The following procedure creates a table, where the table name is supplied as a parameter to the procedure. The EXECUTE IMMEDIATE statement must all be on a single line.

```
CREATE PROCEDURE CreateTableProc(
                   IN tablename char(30)
                   )
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE ' || tablename ||
' ( column1 INT PRIMARY KEY)'
END
```

To call the procedure and create a table **mytable**:

```
CALL CreateTableProc( 'mytable' )
```

EXIT statement [ISQL]

Function	To leave Interactive SQL.
Syntax	**EXIT** | **QUIT** | **BYE**
Permissions	None.
Side effects	Will do a commit if option COMMIT_ON_EXIT is ON (default); otherwise will do a rollback.
See also	"SET OPTION statement" on page 553
Description	Leave the Interactive SQL environment and return to the operating system. This will close your connection with the database. Before doing so, Interactive SQL will perform a COMMIT operation if the COMMIT_ON_EXIT option is ON. If the option is OFF, Interactive SQL will perform a ROLLBACK. The default action is to COMMIT any changes you have made to the database.
Standards and compatibility	♦ **SQL/92** Vendor extension ♦ **Sybase** Not applicable in Adaptive Server Enterprise.

EXPLAIN statement [ESQL]

Function

To retrieve a text specification of the optimization strategy used for a particular cursor.

Syntax

EXPLAIN PLAN FOR CURSOR *cursor-name* **INTO** *host-variable*
    ...   **INTO** *host-variable*
        | **USING DESCRIPTOR** *sqlda-name*

Parameters

cursor-name:	*identifier* or *host-variable*
sqlda-name:	*identifier*

Permissions

Must have opened the named cursor.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436
"PREPARE statement" on page 519
"FETCH statement" on page 468
"CLOSE statement" on page 373
"OPEN statement" on page 511
"Cursors in Embedded SQL" on page 33 of the book *Adaptive Server Anywhere Programming Interfaces Guide*
"The SQL Communication Area" on page 27 of the book *Adaptive Server Anywhere Programming Interfaces Guide*

Description

The EXPLAIN statement retrieves a text representation of the optimization strategy for the named cursor. The cursor must be previously declared and opened.

The **host-variable** or SQLDA variable must be of string type. The optimization string specifies in what order the tables are searched, and also which indexes are being used for the searches if any. This string can be quite long, but most optimization strings will fit into 300 characters.

The format of this string is, in general:

```
table (index), table (index), ...
```

If a table has been given a correlation name, the correlation name will appear instead of the table name. The order that the table names appear in the list is the order in which they will be accessed by the database server. After each table is a parenthesized index name. This is the index that will be used to access the table. If no index will be used (the table will be scanned sequentially) the letters "seq" will appear for the index name. If a particular SQL SELECT statement involves subqueries, a colon (:) will separate each subquery's optimization string. These subquery sections will appear in the order that the database server executes the queries.

After successful execution of the EXPLAIN statement, the **sqlerrd[3]** field of the SQLCA (SQLIOESTIMATE) will be filled in with an estimate of the number of input/output operations required to fetch all rows of the query.

A discussion with quite a few examples of the optimization string can be found in "Monitoring and Improving Performance" on page 623 of the book *Adaptive Server Anywhere User's Guide*.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

Example

- The following example illustrates use of EXPLAIN:

```
EXEC SQL BEGIN DECLARE SECTION;
char plan[300];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE employee_cursor CURSOR FOR
   SELECT empnum, empname
   FROM employee
   WHERE name like :pattern;
EXEC SQL OPEN employee_cursor;
EXEC SQL EXPLAIN PLAN FOR CURSOR employee_cursor
INTO :plan;
printf( "Optimization Strategy: '%s'.n", plan );
```

FETCH statement [ESQL] [SP]

Function

To reposition a cursor, and then get data from it.

Syntax

```
FETCH
   {
     NEXT
   | PRIOR
   | FIRST
   | LAST
   | ABSOLUTE row-count
   | RELATIVE row-count
   }
   ... cursor-name
   ... [  | INTO host-variable-list     | ]
                | USING DESCRIPTOR sqlda-name|
                | INTO variable-list|
      ...       [ PURGE ] [ BLOCK n ]
      ...       [ FOR UPDATE ] [ ARRAY fetch-count ]
      ...       INTO variable-list [ FOR UPDATE ]
```

Parameters

row-count:	*number* or *host variable*
cursor-name:	*identifier* or *host-variable*
host-variable-list:	may contain indicator variables
sqlda-name:	*identifier*
fetch-count:	*integer* or *host variable*

Permissions

The cursor must be opened, and the user must have SELECT permission on the tables referenced in the declaration of the cursor.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436
"PREPARE statement" on page 519
"OPEN statement" on page 511
"Cursors in Embedded SQL" on page 33 of the book *Adaptive Server Anywhere Programming Interfaces Guide*
"Using cursors in procedures and triggers" on page 251 of the book *Adaptive Server Anywhere User's Guide*
FETCH in *PowerScript Reference*

Description

The FETCH statement retrieves one row from the named cursor.

The ARRAY clause allows so-called **wide fetches**, which retrieve more than one row at a time, and which may improve performance.

The cursor must have been previously opened.

One row from the result of the SELECT statement is put into the variables in the variable list. The correspondence is one-to-one from the select list to the host variable list.

One or more rows from the result of the SELECT statement are put into either the variables in the variable list or the program data areas described by the named SQLDA. In either case, the correspondence is one-to-one from the select list to either the host variable list or the SQLDA descriptor array.

The INTO clause is optional. If it is not specified, the FETCH statement positions the cursor only (see the following paragraphs).

An optional positional parameter allows the cursor to be moved before a row is fetched. The default is NEXT, which causes the cursor to be advanced one row before the row is fetched. PRIOR causes the cursor to be backed up one row before fetching.

RELATIVE positioning is used to move the cursor by a specified number of rows in either direction before fetching. A positive number indicates moving forward and a negative number indicates moving backwards. Thus, a NEXT is equivalent to RELATIVE 1 and PRIOR is equivalent to RELATIVE -1. RELATIVE 0 retrieves the same row as the last fetch statement on this cursor.

The ABSOLUTE positioning parameter is used to go to a particular row. A zero indicates the position before the first row (see "Using cursors in procedures and triggers" on page 251 of the book *Adaptive Server Anywhere User's Guide*).

A one (1) indicates the first row, and so on. Negative numbers are used to specify an absolute position from the end of the cursor. A negative one (-1) indicates the last row of the cursor. FIRST is a short form for ABSOLUTE 1. LAST is a short form for ABSOLUTE -1.

The OPEN statement initially positions the cursor before the first row.

If the fetch includes a positioning parameter and the position is outside the allowable cursor positions, the SQLE_NOTFOUND warning is issued.

Cursor positioning problems
Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database server will not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row will not appear at all until the cursor is closed and opened again.

This occurs if a temporary table had to be created to open the cursor (see "Temporary tables used in query processing" on page 640 of the book *Adaptive Server Anywhere User's Guide* for a description).

The UPDATE statement may cause a row to move in the cursor. This will happen if the cursor has an ORDER BY that uses an existing index (a temporary table is not created).

The FOR UPDATE clause indicates that the fetched row will subsequently be updated with an UPDATE WHERE CURRENT OF CURSOR statement. This clause causes the database server to put a write lock on the row. The lock will be held until the end of the current transaction. See "How locking works" on page 382 of the book *Adaptive Server Anywhere User's Guide*.

Using the FETCH statement in Embedded SQL

The following clauses are for use in Embedded SQL only:

- ♦ USING DESCRIPTOR *sqlda-name*
- ♦ INTO *host-variable-list*
- ♦ PURGE
- ♦ BLOCK n
- ♦ ARRAY fetch-count
- ♦ Use of *host-variable* in cursor-name and row-count.

The DECLARE CURSOR statement must appear before the FETCH statement in the C source code, and the OPEN statement must be executed before the FETCH statement. If a host variable is being used for the cursor name, the DECLARE statement actually generates code and thus must be executed before the FETCH statement.

In the multi-user environment, rows may be fetched by the client more than one at a time. Note that in UNIX, the client is linked into the application so this will always happen by default. This is referred to as block fetching or multi-row fetching. The first fetch causes several rows to be sent back from the server. The client buffers these rows, and subsequent fetches are retrieved from these buffers without a new request to the server.

The BLOCK clause gives the client and server a hint as to how many rows may be fetched by the application. The special value of 0 means the request will be sent to the server and a single row will be returned (no row blocking).

The PURGE clause causes the client to flush its buffers of all rows, and then send the fetch request to the server. Note that this fetch request may return a block of rows.

If the SQLSTATE_NOTFOUND warning is returned on the fetch, the *sqlerrd[2]* field of the SQLCA (SQLCOUNT) will contain the number of rows by which the attempted fetch exceeded the allowable cursor positions. (A cursor can be on a row, before the first row or after the last row.) The value is 0 if the row was not found but the position is valid, for example, executing FETCH RELATIVE 1 when positioned on the last row of a cursor. The value will be positive if the attempted fetch was further beyond the end of the cursor, and negative if the attempted fetch was further before the beginning of the cursor.

After successful execution of the fetch statement, the *sqlerrd[1]* field of the SQLCA (SQLIOCOUNT) will be incremented by the number of input/output operations required to perform the fetch. This field is actually incremented on every database statement.

To use wide fetches in Embedded SQL, include the fetch statement in your code as follows:

```
EXEC SQL FETCH . . . ARRAY nnn
```

where *ARRAY nnn* is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The SQLDA must contain **nnn * (columns per row)** variables. The first row is placed in SQLDA variables 0 **to (columns per row)-**1, and so on.

The server returns in SQLCOUNT the number of records fetched, and always returns a SQLCOUNT greater than zero unless there is an error. Older versions of the only server return a single row and set the SQLCOUNT to zero. A SQLCOUNT of zero with no error condition indicates that one valid row has been fetched.

Standards and compatibility

- **SQL/92** Entry level feature. Use in procedures is a Persistent Stored Module feature.
- **Sybase** Supported in Adaptive Server Enterprise.

Example

- The following is an Embedded SQL example.

```
EXEC SQL DECLARE cur_employee CURSOR FOR
SELECT emp_id, emp_lname FROM employee ;
EXEC SQL OPEN cur_employee;
EXEC SQL FETCH cur_employee
INTO :emp_number, :emp_name:indicator;
```

For a detailed example of using wide fetches, see the section "Fetching more than one row at a time" on page 41 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

♦ The following is a procedure example:

```
BEGIN
   DECLARE cur_employee CURSOR FOR
      SELECT emp_lname
      FROM employee ;
   DECLARE name CHAR(40) ;
   OPEN cur_employee;
   LOOP
      FETCH NEXT cur_employee into name ;
       ...
   END LOOP
   CLOSE cur_employee;
END
```

FOR statement

Function

Repeat the execution of a statement list once for each row in a cursor.

Syntax

```
[ statement-label : ]
   . . .FOR for-loop-name AS cursor-name
      . . . CURSOR FOR statement
      . . .[ FOR UPDATE | FOR READ ONLY ]
      . . . DO statement-list
   . . . END FOR [ statement-label ]
```

Permissions

None.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436
"FETCH statement" on page 468
"LEAVE statement" on page 502
"LOOP statement" on page 508

Description

The FOR statement is a control statement that allows you to execute a list of SQL statements once for each row in a cursor. The FOR statement is equivalent to a compound statement with a DECLARE for the cursor and a DECLARE of a variable for each column in the result set of the cursor followed by a loop that fetches one row from the cursor into the local variables and executes *statement-list* once for each row in the cursor.

The name and data type of each local variables is derived from the *statement* used in the cursor. With a SELECT statement, the data types will be the data types of the expressions in the select list. The names will be the select list item aliases, if they exist; otherwise, they will be the name of the columns. Any select list item that is not a simple column reference must have an alias. With a CALL statement, the names and data types will be taken from the RESULT clause in the procedure definition.

The LEAVE statement can be used to resume execution at the first statement after the END FOR. If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Standards and compatibility

♦ **SQL/92** Persistent Stored Module feature.

♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

♦ The following fragment illustrates the use of the FOR loop.

```
FOR names AS curs CURSOR FOR
SELECT emp_lname
FROM employee
DO
   CALL search_for_name( emp_lname );
END FOR;
```

FORWARD TO statement

Function To send native syntax to a remote server.

Syntax 1 **FORWARD TO** *server-name* { *sql-statement* }

Syntax 2 **FORWARD TO** [*server-name*]

Permissions None

Side effects The remote connection is set to AUTOCOMMIT (unchained) mode for the duration of the FORWARD TO session. Any work that was pending prior to the FORWARD TO statement is automatically committed.

Description The FORWARD TO statement enables users to specify the server to which a passthrough connection is required. The statement can be used in two ways:

- ♦ To send a statement to a remote server (syntax 1)
- ♦ To place Adaptive Server Anywhere into passthrough mode for sending a series of statements to a remote server (syntax 2)

When establishing a connection to server-name on behalf of the user, the server uses:

- ♦ A remote login alias set using CREATE EXTERNLOGIN, or
- ♦ If a remote login alias is not set up, the name and password used to communicate with the Adaptive Server Anywhere

If the connection cannot be made to the server specified, the reason is contained in a message returned to the user.

After statements are passed to the requested server, any results are converted into a form that can be recognized by the client program.

server-name is the name of the remote server.

sql-statement is a command in the remote server's native syntax. The command or group of commands is enclosed in curly brackets ({}).

When you specify a server_name, but do not specify a statement in the FORWARD TO query, your session enters passthrough mode, and all subsequent queries are passed directly to the remote server. To turn passthrough mode off, issue FORWARD TO without a server_name specification.

Standards and compatibility

- ♦ **SQL/92** Entry-level feature.
- ♦ **Sybase** Supported by Open Client/Open Server.

Example

- ♦ The following example shows a passthrough session with the remote server **ase_prod**:

```
FORWARD TO aseprod
SELECT * from titles
SELECT * from authors
FORWARD TO
```

FROM clause

Function

To specify the database tables or views involved in a SELECT or UPDATE statement.

Syntax

... **FROM** *table-expression*, ...

Parameters

```
table-expression:
     table-spec
   | table-expression join-type table-spec [ ON condition ]
   | ( table-expression, ... )

table-spec:
     [userid.]table-name [ [AS] correlation-name ]
   |  select-statement  [ AS correlation-name ( column-name, ... )  ]

join-type:
     CROSS JOIN
   | [ NATURAL | KEY ] JOIN
   | [ NATURAL | KEY ] INNER JOIN
   | [ NATURAL | KEY ] LEFT OUTER JOIN
   | [ NATURAL | KEY ] RIGHT OUTER JOIN
```

Permissions

Must be connected to the database.

Side effects

None.

See also

"SELECT statement" on page 542
"UPDATE statement" on page 572
"Joins: Retrieving Data from Several Tables" on page 129 of the book *Adaptive Server Anywhere User's Guide*

Description

The SELECT and UPDATE statements require a table list, to specify which tables will be used by the statement.

Views
Although this description refers to tables, it applies to views unless otherwise noted.

The FROM table list creates a result set consisting of all the columns from all the tables specified. Initially, all combinations of rows in the component tables are in the result set, and the number of combinations is usually reduced by **join** conditions and/or WHERE conditions.

Tables owned by a different user can be qualified by specifying the **user ID**. Tables owned by groups to which the current user belongs will be found by default without specifying the user ID (see "Referring to tables owned by groups" on page 588 of the book *Adaptive Server Anywhere User's Guide*).

The **correlation name** is used to give a temporary name to the table for this SQL statement only. This is useful when referencing columns from a table with a long name. The correlation name is also necessary to distinguish between table instances if you reference the same table more than once in the same query. If no correlation name is specified, the table name is used as the correlation name for the current statement.

If the same correlation name is used twice for the same table in a table expression, that table is treated as if it were listed only once. For example, in:

```
SELECT *
FROM sales_order
KEY JOIN sales_order_items,
sales_order
KEY JOIN employee
```

the two instances of the **sales_order** table are treated as one instance, this is equivalent to:

```
SELECT *
FROM sales_order_items
KEY JOIN sales_order
KEY JOIN employee
```

Whereas:

```
SELECT *
FROM Person HUSBAND, Person WIFE
```

would be treated as two instances of the Person table, with different correlation names HUSBAND and WIFE.

You can supply SELECT statements instead of table or view names in the FROM clause. This allows you to use groups on groups, or joins with groups, without creating a view. The tables that you create in this way are **derived tables**.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** The JOIN clause is not supported in Adaptive Server Enterprise. Instead, you must use the WHERE clause to build joins.

Examples

- The following are valid FROM clauses:

```
...
FROM employee
...

...
FROM employee NATURAL JOIN department
...

...
FROM customer
```

```
KEY JOIN sales_order
KEY JOIN sales_order_items
KEY JOIN product
...
```

- The following query illustrates how to use derived tables in a query:

```
SELECT lname, fname, number_of_orders
FROM customer JOIN
     ( SELECT cust_id, count(*)
       FROM sales_order
        GROUP BY cust_id )
     AS sales_order_counts ( cust_id,
                             number_of_orders )
ON ( customer.id - sales_order_counts.cust_id )
WHERE number_of_orders > 3
```

GET DATA statement [ESQL]

Function

To get string or binary data for one column on the current row of a cursor. GET DATA is usually used to fetch LONG BINARY or LONG VARCHAR fields. See "SET statement" on page 546.

Syntax

GET DATA *cursor-name* **COLUMN** *column-num* **OFFSET** *start-offset*
... [**WITH TEXTPTR**]
... **USING DESCRIPTOR** *sqlda-name*
| **INTO** *host-variable* [, ...] |

Parameters

cursor-name:	*identifier*, or *host-variable*
column-num:	*integer* or *host-variable*
start-offset:	*integer* or *host-variable*
sqlda-name:	*identifier*

Permissions

The cursor must be opened and positioned on a row, using FETCH.

Side effects

None.

See also

"FETCH statement" on page 468
"READTEXT statement" on page 528

Description

Get a piece of one column value from the row at the current cursor position. The value of *column-num* starts at one, and identifies which column's data is to be fetched. That column must be of a string or binary type.

The *start-offset* indicates the number of bytes to skip over in the field value. Normally, this would be the number of bytes previously fetched. The number of bytes fetched on this GET DATA statement is determined by the length of the target host variable.

The indicator value for the target host variable is a short integer, so it cannot always contain the number of bytes truncated. Instead, it contains a negative value if the field contains the NULL value, a positive value (NOT necessarily the number of bytes truncated) if the value is truncated, and zero if a non-NULL value is not truncated.

If the WITH TEXTPTR clause is given, a text pointer is retrieved into a second indicator variable or into the second field in the SQLDA. This text pointer can be used with the Transact-SQL READ TEXT and WRITE TEXT statements.

The total length of the data is returned in the SQLCOUNT field of the SQLCA structure.

Standards and compatibility

- **SQL/92** Vendor extension.

- **Sybase** Not supported by Open Client/Open Server. An alternative is the Transact-SQL READTEXT statement.

Example

- The following example uses GET DATA to fetch a **binary large object** (often called a **blob**).

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY(1000) piece;
short ind;
long offset;
EXEC SQL END DECLARE SECTION;
int size;
/* Open a cursor on a long varchar field */
EXEC SQL DECLARE big_cursor CURSOR FOR
SELECT long_data FROM some_table
WHERE key_id = 2;
EXEC SQL OPEN big_cursor;
EXEC SQL FETCH big_cursor INTO :piece;
for( offset = 0; ; offset += piece.len ) {
   EXEC SQL GET DATA big_cursor COLUMN 1
   OFFSET :offset INTO :piece:ind;
   /* Done if the NULL value */
   if( ind < 0 ) break;
   write_out_piece( piece );
   /* Done when the piece was not truncated */
   if( ind == 0 ) break;
}
EXEC SQL CLOSE big_cursor;
```

GET DESCRIPTOR statement [ESQL]

Function

Retrieves information about a variable within a descriptor area, or retrieves its value.

Syntax

GET DESCRIPTOR *descriptor-name*
...{ *hostvar* = **COUNT** | **VALUE** *n* *assignment* [,...] }

Parameters

assignment:
hostvar = { **TYPE** | **LENGTH** | **PRECISION** | **SCALE** | **DATA** | **INDICATOR** | **NAME** | **NULLABLE** | **RETURNED_LENGTH** }

Permissions

None.

Side effects

None.

See also

"ALLOCATE DESCRIPTOR statement" on page 342
"DEALLOCATE DESCRIPTOR statement" on page 433
"SET DESCRIPTOR statement" on page 552
"The SQL descriptor area (SQLDA)" on page 45 of the book *Adaptive Server Anywhere Programming Interfaces Guide*

Description

The GET DESCRIPTOR statement is used to retrieve information about a variable within a descriptor area, or to retrieve its value.

The value *n* specifies the variable in the descriptor area about which the information will be retrieved. Type checking is performed when doing GET ... DATA to ensure that the host variable and the descriptor variable have the same data type.

If an error occurs, it is returned in the SQLCA.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Open Client/Open Server.

Example

- For an example, see "ALLOCATE DESCRIPTOR statement" on page 342.

GET OPTION statement [ESQL]

Function

To find the current setting of an option. This statement is deprecated in favor of system functions.

Syntax

GET OPTION [*userid*.]*option-name*
    ... | **INTO** *host-variable*
         | **USING DESCRIPTOR** *sqlda-name*

Parameters

userid:	*identifier, string,* or *host-variable*
option-name:	*identifier, string,* or *host-variable*
host-variable:	indicator variable allowed
sqlda-name:	*identifier*

Permissions

None required.

Side effects

None.

See also

"SET OPTION statement" on page 553
"System and catalog stored procedures" on page 753

Description

The GET OPTION statement is provided for compatibility with older versions of the software. The recommended way to get the values of options is to use the **connection_property** system function.

The GET OPTION statement gets the option setting of the option *option-name* for the user *userid* or for the connected user if *userid* is not specified. This will be either the user's personal setting or the **PUBLIC** setting if there is no setting for the connected user. If the option specified is a database option and the user has a temporary setting for that option, then the temporary setting is retrieved.

If *option-name* does not exist, GET OPTION returns the warning SQLE_NOTFOUND.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

- The following statement illustrates use of GET OPTION.

```
EXEC SQL GET OPTION 'date_format' INTO :datefmt;
```

GOTO statement [T-SQL]

Function

To branch to a labeled statement.

Syntax

label:
    GOTO *label*

Authorization

None.

Side effects

None.

Description

Any statement in a Transact-SQL procedure, trigger, or batch can be labeled. The label name is a valid identifier followed by a colon. In the GOTO statement, the colon is not used.

Standards and compatibility

- ♦ **SQL/92** Persistent Stored Module feature.
- ♦ **Sybase** Adaptive Server Enterprise supports the GOTO statement.

Example

- ♦ The following Transact-SQL batch prints the message "yes" on the server window four times:

```
declare @count smallint
select @count = 1
restart:
   print 'yes'
   select @count = @count + 1
   while @count <=4
    goto restart
```

GRANT statement

Function

To give permissions to specific users and to create new user IDs.

Syntax 1

GRANT CONNECT TO *userid*,... **IDENTIFIED BY** *password*,...

Syntax 2

```
GRANT {
      DBA ,
      GROUP ,
      MEMBERSHIP IN GROUP userid,...,
      [ RESOURCE  | ALL ]
    }
    ... TO userid,...
```

Syntax 3

```
GRANT {
      ALL [ PRIVILEGES ] ,
      ALTER  ,
      DELETE  ,
      INSERT  ,
      REFERENCES  [ ( column-name,... ) ] ,
      SELECT [ ( column-name,... ) ] ,
      UPDATE [ ( column-name,... ) ] ,
    }
    ... ON [ owner.]table-name
    ... TO userid, ...
      [ WITH GRANT OPTION ]
```

Syntax 4

GRANT EXECUTE ON [*owner.*]*procedure-name* **TO** *userid*,...

Syntax 5

GRANT INTEGRATED LOGIN TO *user_profile_name*,... **AS USER** *userid*

Permissions

For Syntax 1 or 2 one of the following conditions must be met.

- ♦ You are changing your own password using GRANT CONNECT.
- ♦ You have DBA authority.

If you are changing another user's password (with DBA authority), the other user must not be connected to the database.

For Syntax 3, one of the following conditions must be met:

- ♦ You created the table
- ♦ You have been granted permissions on the table with GRANT OPTION.
- ♦ You have DBA authority

For Syntax 4, one of the following conditions must be met:

- ♦ You created the procedure
- ♦ You have DBA authority

For Syntax 5, the following condition must be met:

- ♦ You have DBA authority

Side effects

Automatic commit.

See also

"REVOKE statement" on page 536

Description

The GRANT statement is used to grant database permissions to individual user IDs and groups. It is also used to create and delete users and groups.

Special privileges

Syntax 1 and 2 of the GRANT statement are used for granting special privileges to users as follows:

CONNECT TO userid,... Creates a new user. GRANT CONNECT can also be used by any user to change their own password. To create a user with the empty string as the password, type:

```
GRANT CONNECT TO userid IDENTIFIED BY ""
```

To create a user with no password, type:

```
GRANT CONNECT TO userid
```

A user with no password cannot connect to the database. This is useful if you are creating a group and do not want anyone to connect to the database using the group user ID. The password must be a valid identifier, as described in "Statement elements" on page 180.

DBA Database Administrator authority gives a user permission to do anything. This is usually reserved for the person in the organization who is looking after the database.

GROUP Allows the user(s) to have members.

For more information, see "Managing groups" on page 586 of the book *Adaptive Server Anywhere User's Guide*.

MEMBERSHIP IN GROUP This allows the user(s) to inherit table permissions from a group and to reference tables created by the group without qualifying the table name.

For more information, see "Managing groups" on page 586 of the book *Adaptive Server Anywhere User's Guide*.

Syntax 3 of the GRANT statement is used to grant permission on individual tables or views. The table permissions can be specified individually, or by specifying ALL grants all six permissions at once.

RESOURCE Allows the user to create tables and views. In syntax 2, **ALL** is a synonym for RESOURCE that is compatible with Sybase Adaptive Server Enterprise.

ALL In Syntax 3, this grants all of the permissions outlined below.

Permissions

The permissions have the following meaning:

ALTER The users will be allowed to alter this table with the ALTER TABLE statement. This permission is not allowed for views.

DELETE The users will be allowed to delete rows from this table or view.

INSERT The users will be allowed to insert rows into the named table or view.

REFERENCES [(column-name,...)] The users will be allowed to create indexes on the named tables, and foreign keys which reference the named tables. If column names are specified, the users will be allowed to reference only those columns. REFERENCES permissions on columns cannot be granted for views, only for tables.

INDEX is a synonym for REFERENCES.

SELECT [(column-name,...)] The users will be allowed to look at information in this view or table. If column names are specified, the users will be allowed to look at only those columns. SELECT permissions on columns cannot be granted for views, only for tables.

UPDATE [(column-name,...)] The users will be allowed to update rows in this view or table. If column names are specified, the users will be allowed to update only those columns. UPDATE permissions on columns cannot be granted for views, only for tables.

Other notes

If WITH GRANT OPTION is specified, then the named user ID is also given permission to GRANT the same permissions to other user IDs.

Syntax 4 of the GRANT statement is used to grant permission to execute a procedure.

Syntax 5 of the GRANT statement creates an explicit integrated login mapping between one or more Windows NT user profiles and an existing database user ID, allowing users who successfully log in to their local machine to connect to a database without having to provide a user ID or password.

☞ For more information on integrated logins see "Using integrated logins" on page 58 of the book *Adaptive Server Anywhere User's Guide*.

Standards and compatibility

- **SQL/92** Syntax 3 is an entry-level feature. Syntax 4 is a Persistent Stored Module feature. Other syntaxes are vendor extensions.
- **Sybase** Syntaxes 2 and 3 are supported in Adaptive Server Enterprise. The security model is different in Adaptive Server Enterprise and Adaptive Server Anywhere, so other syntaxes differ.

Examples

♦ Make two new users for the database.

```
GRANT
CONNECT TO Laurel, Hardy
IDENTIFIED BY Stan, Ollie
```

♦ Grant permissions on the employee table to user Laurel.

```
GRANT
SELECT, UPDATE ( street )
ON employee
TO Laurel
```

More than one permission can be granted in a single statement. The permissions are separated by commas.

♦ Allow the user Hardy to execute the **Calculate_Report** procedure.

```
GRANT
EXECUTE ON Calculate_Report
TO Hardy
```

HELP statement [ISQL]

Function To receive help in the Interactive SQL environment.

Syntax **HELP** [*topic*]

Permissions None.

Side effects None.

Description The HELP statement is used to enter the Interactive SQL interactive help facility. The *topic* for help can be optionally specified. If *topic* is not specified, the help system is entered at the index.

Standards and compatibility

♦ **SQL/92** Vendor extension

♦ **Sybase** Not applicable

IF statement

Function

Provide conditional execution of SQL statements.

Syntax

IF *search-condition* **THEN** *statement-list*
... [**ELSEIF** *search-condition* **THEN** *statement-list*] ...
... [**ELSE** *statement-list*]
... **END IF**

Permissions

None.

Side effects

None.

See also

"BEGIN... END statement" on page 361
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The IF statement is a control statement that allows you to conditionally execute the first list of SQL statements whose *search-condition* evaluates to TRUE. If no *search-condition* evaluates to TRUE, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed.

Execution resumes at the first statement after the END IF.

IF statement is different from IF expression
Do not confuse the syntax of the IF statement with that of the IF expression.

For information on the IF expression, see "IF expressions" on page 188.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** The Transact-SQL IF statement has a slightly different syntax.

Example

- The following procedure illustrates the use of the IF statement:

```
CREATE PROCEDURE TopCustomer (OUT TopCompany
CHAR(35),                              OUT
TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
   FOR SQLSTATE '02000' ;
   DECLARE curThisCust CURSOR FOR
   SELECT company_name, CAST(
   sum(sales_order_items.quantity *
   product.unit_price) AS INTEGER) VALUE
   FROM customer
   LEFT OUTER JOIN sales_order
   LEFT OUTER JOIN sales_order_items
   LEFT OUTER JOIN product
   GROUP BY company_name ;

   DECLARE ThisValue INT ;
   DECLARE ThisCompany CHAR(35) ;
   SET TopValue = 0 ;
   OPEN curThisCust ;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
      INTO ThisCompany, ThisValue ;
      IF SQLSTATE = err_notfound THEN
         LEAVE CustomerLoop ;
      END IF ;
      IF ThisValue > TopValue THEN
         SET TopValue = ThisValue ;
         SET TopCompany = ThisCompany ;
      END IF ;
   END LOOP CustomerLoop ;
   CLOSE curThisCust ;
END
```

IF statement [T-SQL]

Function

To provide conditional execution of a SQL statement, as an alternative to the Watcom-SQL IF statement.

Syntax

IF *expression*
... *statement*
... [**ELSE**
... [**IF** *expression*]
... *statement*]

Authorization

None.

Side effects

None.

Description

The Transact-SQL IF conditional and the ELSE conditional each control the performance of only a single SQL statement or compound statement (between the keywords BEGIN and END).

In comparison to the Watcom-SQL IF statement, there is no THEN in the Transact-SQL IF statement. The Transact-SQL version also has no ELSEIF or END IF keywords.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Adaptive Server Enterprise supports the Transact-SQL IF statement.

Example

- The following example illustrates the use of the Transact-SQL IF statement:

```
IF (SELECT max(id) FROM sysobjects) < 100
   RETURN
ELSE
   BEGIN
      PRINT "These are the user-created objects"
      SELECT name, type, id
      FROM sysobjects
      WHERE id < 100
END
```

- The following two statement blocks illustrate Transact-SQL and Watcom-SQL compatibility:

```
/* Transact-SQL IF statement */
IF @v1 = 0
   PRINT '0'
ELSE IF @v1 = 1
   PRINT '1'
ELSE
   PRINT 'other'
/* Watcom-SQL IF statement */
```

```
IF v1 = 0 THEN
   PRINT '0'
ELSEIF v1 = 1 THEN
   PRINT '1'
ELSE
   PRINT 'other'
END IF
```

INCLUDE statement [ESQL]

Function

Include a file into a source program to be scanned by the SQL source language preprocessor.

Syntax

INCLUDE *filename*

Parameters

filename:	*identifier*

Permissions

None.

Side effects

None.

Description

The INCLUDE statement is very much like the C preprocessor **#include** directive. The SQL preprocessor reads the given file, and inserts its contents into the output C file. Thus, if an include file contains information that the SQL preprocessor requires, it should be included with the Embedded SQL INCLUDE statement.

Two file names are specially recognized: SQLCA and SQLDA. Any C program using Embedded SQL must contain an

```
EXEC SQL INCLUDE SQLCA;
```

statement before any Embedded SQL statements. This statement must appear at a position in the C program where static variable declarations are allowed. Many Embedded SQL statements require variables (invisible to the programmer), which are declared by the SQL preprocessor at the position of the SQLCA include statement. The SQLDA file must be included if any SQLDAs are used.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Open Client/Open Server.

INPUT statement [ISQL]

Function

To import data into a database table from an external file or from the keyboard.

Syntax

INPUT INTO [*owner*.]*table-name*
... [**FROM** *filename* | **PROMPT**]
... [**FORMAT** *input-format*]
... [**ESCAPE CHARACTER** *character*]
... [**BY ORDER** | **BY NAME**]
... [**DELIMITED BY** *string*]
... [**COLUMN WIDTHS** (*integer*,...)]
... [**NOSTRIP**]
... [(*column-name*, ...)]

Permissions

Must have INSERT permission on the table or view.

Side effects

None.

See also

"OUTPUT statement" on page 514
"INSERT statement" on page 498
"UPDATE statement" on page 572
"DELETE statement" on page 442
"SET OPTION statement" on page 553
"LOAD TABLE statement" on page 504

Description

The INPUT statement allows efficient mass insertion into a database table. Lines of input are read either from the user via an input window (if PROMPT is specified) or from a file (if FROM filename is specified). If neither is specified, the input will be read from the command file that contains the input statement—this can even be directly from the Interactive SQL editor. In this case, input is ended with a line containing only the string END.

These lines are inserted into the named table. If a column list is specified, the data is inserted into the specified columns of the named table.

Normally, the INPUT statement stops when it attempts to insert a row that causes an error. Errors can be treated in different ways by setting the ON_ERROR and CONVERSION_ERROR options (see SET OPTION). Interactive SQL will print a warning in the statistics window if any string values are truncated on INPUT. Missing values for NOT NULL columns will be set to zero for numeric types and to the empty string for non-numeric types.

The default escape character for hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

The escape character can be changed, using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter:

```
... ESCAPE CHARACTER '!'
```

Only one single-byte character can be used as an escape character.

The BY clause allows the user to specify whether the columns from the input file should be matched up with the table columns based on their ordinal position in the lists (ORDER, the default) or by their names (NAME). Not all input formats have column name information in the file. NAME is allowed only for those formats that do. They are the same formats that allow automatic table creation listed below: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

The DELIMITED BY clause allows you to specify a string to be used as the delimiter in ASCII input format.

COLUMN WIDTHS can be specified for FIXED format only. It specifies the widths of the columns in the input file. If COLUMN WIDTHS is not specified, the widths are determined by the database column types.

Normally, for ASCII input format, trailing blanks will be stripped from unquoted strings before the value is inserted. NOSTRIP can be used to suppress trailing blank stripping. Trailing blanks are not stripped from quoted strings, regardless of whether the option is used. Leading blanks are stripped from unquoted strings, regardless of the NOSTRIP option setting.

If the ASCII file has entries such that a column appears to be null, LOAD TABLE treats it as null. If the column in that position cannot be null, it inserts a zero in numeric columns and an empty string in character columns.

Each set of values must occupy one input line and must be in the format specified by the FORMAT clause, or the format set by the SET INPUT_FORMAT statement if the FORMAT clause is not specified. When input is entered by the user, an empty screen is provided for the user to enter one row per line in the input format.

Certain file formats contain information about column names and types. Using this information, the INPUT statement will create the database table if it does not already exist. This is a very easy way to load data into the database. The formats that have enough information to create the table are: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

Allowable input formats are:

ASCII Input lines are assumed to be ASCII characters, one row per line, with values separated by commas. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing commas must be enclosed in either single or double quotes. If the string itself contains single or double quotes, double the quote character to use it within the string. Optionally, you can use the DELIMITED BY clause to specify a different delimiter string than the default, which is a comma.

Three other special sequences are also recognized. The two characters \n represent a newline character, \\ represents a single (\), and the sequence \xDD represents the character with hexadecimal code DD.

DBASE The file is in dBASE II or dBASE III format. Interactive SQL will attempt to determine which format, based on information in the file. If the table doesn't exist, it will be created.

DBASEII The file is in dBASE II format. If the table doesn't exist, it will be created.

DBASEIII The file is in dBASE III format. If the table doesn't exist, it will be created.

DIF Input file is in Data Interchange Format. If the table doesn't exist, it will be created.

FIXED Input lines are in fixed format. The width of the columns can be specified using the COLUMN WIDTHS clause. If they are not specified, column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.

The FIXED format cannot be used with binary columns that contain embedded newline and End of File character sequences.

FOXPRO The file is in FoxPro format (the FoxPro memo field is different than the dBASE memo field). If the table doesn't exist, it will be created.

LOTUS The file is a Lotus WKS format worksheet. INPUT assumes that the first row in the Lotus WKS format worksheet is column names. If the table doesn't exist, it will be created. In this case, the types and sizes of the columns created may not be correct because the information in the file pertains to a cell, not to a column.

WATFILE The input is a WATFILE file. If the table doesn't exist, it will be created.

Input from a command file is terminated by a line containing END. Input from a file is terminated at the end of the file.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not applicable

Example

- The following is an example of an INPUT statement from an ASCII text file.

```
INPUT INTO employee
FROM new_emp.inp
FORMAT ascii;
```

INSERT statement

Function

To insert a single row (format 1) or a selection of rows from elsewhere in the database (format 2) into a table.

Syntax 1

INSERT [**INTO**] [*owner*.]*table-name* [(*column-name*, ...)]
 ... **VALUES** (*expression* | **DEFAULT**, ...)

Syntax 2

INSERT [**INTO**] [*owner*.]*table-name* [(*column-name*, ...)]
 ... *select-statement*

Permissions

Must have INSERT permission on the table.

Side effects

None.

See also

"INPUT statement" on page 494
"UPDATE statement" on page 572
"DELETE statement" on page 442
"PUT statement" on page 524

Description

The INSERT statement is used to add new rows to a database table.

Format 1 allows the insertion of a single row, with the specified expression values. The keyword DEFAULT can be used to cause the default value for the column to be inserted. If the optional list of column names is given, the values are inserted one for one into the specified columns. If the list of column names is not specified, the values are inserted into the table columns in the order they were created (the same order as retrieved with SELECT *). The row is inserted into the table at an arbitrary position. (In relational databases, tables are not ordered.)

Format 2 allows the user to do mass insertion into a table with the results of a fully general SELECT statement. Insertions are done in an arbitrary order unless the SELECT statement contains an ORDER BY clause. The columns from the select list are matched ordinally with the columns specified in the column list, or sequentially in the order in which the columns were created.

Note
The NUMBER(*) function is useful for generating primary keys with format 2 of the INSERT statement (see "SQL Functions" on page 267).

Inserts can be done into views, if the SELECT statement defining the view has only one table in the FROM clause and does not contain a GROUP BY clause or an aggregate function, or involve a UNION operation.

Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. Thus a string *Value* inserted into a table is always held in the database with an upper-case V and the remainder of the letters lower case. SELECT statements return the string as *Value*. If the database is not case-sensitive, however, all comparisons make *Value* the same as *value*, *VALUE*, and so on. Further, if a single-column primary key already contains an entry *Value*, an INSERT of *value* is rejected, as it would make the primary key not unique.

Performance hint
To insert many rows into a table, it is more efficient to declare a cursor and use the PUT statement to insert the rows, where possible, than to carry out many separate INSERT statements.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Adaptive Server Enterprise.

Examples

- Add an Eastern Sales department to the database.

```
INSERT
INTO department ( dept_id, dept_name )
VALUES ( 230, 'Eastern Sales' )
```

- Fill the table dept_head with the names of department heads and their departments.

```
INSERT
INTO dept_head (name, dept)
SELECT emp_fname || ' ' || emp_fname
         AS name,
      dept_name
FROM employee JOIN department
ON emp_id = dept_head_id
```

INSTALL statement

Function

To make Java classes available for use within a database.

Syntax

```
INSTALL JAVA
    [ NEW | UPDATE ]
    [ JAR jar-name ]
    FROM FILE filename
```

Permissions

DBA permissions are required to execute the INSTALL statement.

All installed classes can be referenced in any way by any user.

Not supported on Windows CE.

See also

"REMOVE statement" on page 530

Description

Install mode If you specify an install mode of NEW, the referenced Java classes must be new classes, rather than updates of currently installed classes. An error occurs if a class with the same name exists in the database and the NEW install mode is used.

If you specify UPDATE, the referenced Java classes may include replacements for Java classes that are already installed in the given database.

If *install-mode* is omitted, the default is NEW.

JAR If this is specified, then the *filename* must designate a jar file. Jar files typically have extensions of *.jar* or *.zip*.

Installed jar and zip files can be compressed or uncompressed. However, jar files produced by the Sun JDK *jar* utility are not supported. Files produced by other zip utilities are supported.

If the JAR option is specified, the jar is retained as a jar after the classes that it contains have been installed. That jar is the associated jar of each of those classes. The jars installed in a database with the JAR option are called the **retained jars** of the database.

Retained jars are referenced in INSTALL and REMOVE statements. Retained jars have no effect on other uses of Java-SQL classes. Retained jars are used by the SQL system for requests by other systems for the class associated with given data. If a requested class has an associated jar, the SQL system can supply that jar, rather than the individual class.

The *jar-name* is a character string value, of up to 255 bytes long. The *jar-name* is used to identify the retained jar in subsequent INSTALL UPDATE and REMOVE statements.

source Specifies the location of the Java class(es) to be installed.

The formats supported for *file-name* include fully qualified file names, such as '*c:\libs\jarname.jar'* and '*/usr/u/libs/jarname.jar*', and relative file names, which are relative to the current working directory of the database server.

The *filename* must identify either a class file, or a jar file.

Class availability

The class definition for each class is loaded by each connection's VM the first time that class is used. When you INSTALL a class, the VM on your connection is implicitly restarted. Therefore, you have immediate access to the new class, whether the INSTALL has an *install-mode* of NEW or UPDATE.

For other connections, the new class is loaded the next time a VM accesses the class for the first time. If the class is already loaded by a VM, that connection does not see the new class until the VM is restarted for that connection (for example, with a STOP JAVA and START JAVA).

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Examples

- The following statement installs the user-created Java class named Demo, by providing the filename and location of the class.

  ```
  INSTALL JAVA NEW
  FROM FILE 'D:\JavaClass\Demo.class'
  ```

  After installation, the class is referenced using its name. Its original file path location is no longer used. For example, the following statement uses the class installed in the previous statement.

  ```
  CREATE VARIABLE d Demo
  ```

  If the Demo class was a member of the package *sybase.work*, the fully qualified name of the class must be used, for example:

  ```
  CREATE VARIABLE d sybase.work.Demo
  ```

- The following statement installs all the classes contained in a zip file, and associates them within the database with a JAR file name.

  ```
  INSTALL JAVA
  JAR 'Widgets'
  FROM FILE 'C:\Jars\Widget.zip'
  ```

  Again, the location of the zip file is not retained and classes must be referenced using the fully qualified class name (package name and class name). The zip file must be an uncompressed Jar file.

LEAVE statement

Function

Continue execution, by leaving a compound statement or LOOP.

Syntax

LEAVE *statement-label*

Permissions

None.

Side effects

None.

See also

"LOOP statement" on page 508
"FOR statement" on page 473
"BEGIN... END statement" on page 361
"Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*

Description

The LEAVE statement is a control statement that allows you to leave a labeled compound statement or a labeled loop. Execution resumes at the first statement after the compound statement or loop.

The compound statement that is the body of a procedure or trigger has an implicit label that is the same as the name of the procedure or trigger.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Not supported in Adaptive Server Enterprise. The BREAK statement provides a similar feature for Transact-SQL compatible procedures.

Examples

- The following fragment shows how the LEAVE statement is used to leave a loop.

```
SET i = 1;
lbl:
LOOP
   INSERT
   INTO Counters ( number )
   VALUES ( i ) ;
   IF i >= 10 THEN
      LEAVE lbl ;
   END IF ;
   SET i = i + 1
END LOOP lbl
```

- The following example fragment uses LEAVE in a nested loop.

```
outer_loop:
LOOP
   SET i = 1;
   inner_loop:
   LOOP
      ...
      SET i = i + 1;
      IF i >= 10 THEN
         LEAVE outer_loop
      END IF
   END LOOP inner_loop
END LOOP outer_loop
```

LOAD TABLE statement

Function

To import data into a database table from an external ASCII-format file.

Syntax

LOAD [**INTO**] **TABLE** [*owner*].*table-name*
 ... **FROM** '*filename-string*'
 ... [**FORMAT ASCII**]
 ... [**DELIMITED BY** *string*]
 ... [**STRIP** { **ON** | **OFF** }]
 ... [**QUOTES** { **ON** | **OFF** }]
 ... [**ESCAPES** { **ON** | **OFF** }]
 ... [**ESCAPE CHARACTER** *character*]
 ... [**WITH CHECKPOINT** { **ON** | **OFF** }]

Permissions

Must be the owner of the table or have DBA authority.

Requires an exclusive lock on the table.

The table cannot be a declared local temporary table.

Side effects

Triggers, including referential integrity actions, are not fired by the LOAD TABLE statement. A COMMIT is performed at the end of the load.

See also

"UNLOAD TABLE statement" on page 570
"INPUT statement" on page 494

Description

The LOAD TABLE statement allows efficient mass insertion into a database table from an ASCII file. LOAD TABLE is more efficient than the Interactive SQL statement INPUT.

> ***Caution***
> *LOAD TABLE is intended solely for fast loading of large amounts of data. It is not intended for routine use in applications.*

If the WITH CHECKPOINT ON clause is not specified, the file used for loading must be retained in case recovery is required. If WITH CHECKPOINT ON is specified, a checkpoint is carried out after loading, and recovery is guaranteed even if the data file is then removed from the system.

LOAD TABLE places an exclusive lock on the whole table; it does not fire any triggers associated with the table.

You can use LOAD TABLE on a global temporary table, but the temporary table must have been created with the ON COMMIT PRESERVE ROWS clause, because LOAD TABLE does a COMMIT after the load. LOAD TABLE cannot be used on declared temporary tables.

If the ASCII file has entries such that a column appears to be NULL, LOAD TABLE treats it as null. If the column in that position cannot be NULL, it inserts a zero in numeric columns and an empty string in character columns.

The following list describes each of the clauses of the statement.

FROM filename-string The filename-string is passed to the server as a string. The string is therefore subject to the same formatting requirements as other SQL strings. In particular:

- To indicate directory paths, the backslash character \ must be represented by two backslashes. The statement to load data from the file *c:\temp\input.dat* into the employee table is:

  ```
  LOAD TABLE employee
  FROM 'c:\\temp\\input.dat' ...
  ```

- The pathname is relative to the database server, not to the client application. If you are running the statement on a database server on another computer, the directory names refers to directories on the server machine, not on the client machine.
- You can use UNC path names to load data from files on computers other than the server. For example, on a Windows for Workgroups, Windows 95, or Windows NT network, you may use the following statement to load data from a file on the client machine:

  ```
  LOAD TABLE employee
  FROM '\\\\client\\temp\\input.dat'
  ```

FORMAT option The only file format currently supported is ASCII. Input lines are assumed to be ASCII characters, one row per line, with values separated by the column delimiter character.

DELIMITED BY option The default column delimiter character is a comma. You can specify an alternative column delimiter by providing a string. Only the first ASCII character of the string is read. The same formatting requirements apply as to other SQL strings. In particular, to specify tab-delimited values, the hexadecimal ASCII code of the tab character (0) is used. The DELIMITED BY clause is as follows:

```
...DELIMITED BY '\x09' ...
```

STRIP option With STRIP turned on (the default), trailing blanks are stripped from values before they are inserted. To turn the STRIP option off, the clause is as follows:

```
...STRIP OFF ...
```

Trailing blanks are stripped only for non-quoted strings. Quoted strings retain their trailing blanks. Leading blanks are trimmed, regardless of the STRIP setting, unless they are enclosed in quotes.

QUOTES option With QUOTES turned on (the default), the LOAD statement looks for a quote character. The quote character is either an apostrophe (single quote) or a quotation mark (double quote). The first such character encountered in the input file is treated as the quote character for the input file.

With quotes on, column delimiter characters can be included in column values. Also, quote characters are assumed not to be part of the value. Therefore, a line of the form

```
'123 High Street, Anytown',(715)398-2354
```

is treated as two values, not three, despite the presence of the comma in the address. Also, the quotes surrounding the address are not inserted into the database.

To include a quote character in a value, with QUOTES on, you must use two quotes. The following line includes a value in the third column that is a single quote character:

```
'123 High Street, Anytown','(715)398-2354',''''
```

ESCAPES option With ESCAPES turned on (the default), characters following the backslash character are recognized and interpreted as special characters by the database server. New line characters can be included as the combination \n, other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash.

ESCAPE CHARACTER option The default escape character for characters stored as hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

This can be changed using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter

```
... ESCAPE CHARACTER '!'
```

Only one single-byte character can be used as an escape character.

WITH CHECKPOINT option The default setting is OFF. If set to ON, a checkpoint is issued after successfully completing and logging the statement.

If WITH CHECKPOINT ON is not specified, and recovery is subsequently required, the data file used to load the table is needed for the recovery to complete successfully. If WITH CHECKPOINT ON is specified, and recovery is subsequently required, if will begin after the checkpoint, and the data file need not be present.

Standards and compatibility

♦ **SQL/92** Vendor extension.

- **Sybase** Not applicable.

LOOP statement

Function

Repeat the execution of a statement list.

Syntax

[*statement-label* :]
...[**WHILE** *search-condition*] **LOOP**
... *statement-list*
...**END LOOP** [*statement-label*]

Permissions

None.

Side effects

None.

See also

"FOR statement" on page 473
"LEAVE statement" on page 502

Description

The WHILE and LOOP statements are control statements that allow you to repeatedly execute a list of SQL statements while a *search-condition* evaluates to TRUE. The LEAVE statement can be used to resume execution at the first statement after the END LOOP.

If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **Sybase** Not supported in Adaptive Server Enterprise. The WHILE statement provides looping in Transact-SQL stored procedures.

Examples

- A While loop in a procedure.

```
...
SET i = 1 ;
WHILE i <= 10 LOOP
   INSERT INTO Counters( number ) VALUES ( i ) ;
   SET i = i + 1 ;
END LOOP ;
...
```

- A labeled loop in a procedure.

```
SET i = 1;
lbl:
LOOP
   INSERT
   INTO Counters( number )
   VALUES ( i ) ;
   IF i >= 10 THEN
      LEAVE lbl ;
   END IF ;
   SET i = i + 1 ;
END LOOP lbl
```

MESSAGE statement

Function

To display a message.

Syntax

MESSAGE *expression*, ...
  [TYPE { INFO | ACTION | WARNING | STATUS }]
  [TO { CLIENT | CONSOLE | LOG }]

Permissions

Must be connected to the database.

Side effects

None.

See also

"CREATE PROCEDURE statement" on page 403

Description

The MESSAGE statement displays a message, which can be any expression. Clauses can specify where the message is displayed.

Valid expressions can include a quoted string or other constant, variable, or function. However, queries are not permitted in the output of a Message statement even though the definition of an expression includes queries.

TYPE clause The TYPE clause only has an effect if the message is sent to the client. The client application must decide how to handle the message. Interactive SQL displays messages in the following locations:

♦ **INFO** The Message window.

♦ **ACTION** A Message box with an OK button.

♦ **WARNING** A Message box with an OK button.

♦ **STATUS** The Message window.

TO clause This clause specifies the destination of a message:

♦ The default is CONSOLE, which means the database server window.

♦ A destination of LOG sends messages to the server log file specified by the `-o` command-line option.

♦ A destination of CLIENT sends messages to the client application. Your application must decide how to handle the message, and you can use the TYPE as information on which to base that decision.

Standards and compatibility

♦ **SQL/92** Vendor extension.

♦ **Sybase** Not supported in Adaptive Server Enterprise. The Transact-SQL PRINT statement provides a similar feature, and is also available in Adaptive Server Anywhere.

Example

♦ The following procedure displays a message on the server message window:

```
CREATE PROCEDURE message_test ()
BEGIN
MESSAGE 'The current date and time: ', Now();
END
```

♦ The statement:

```
CALL message_test()
```

displays the string *The current date and time,* and the current date and time, on the database server message window.

OPEN statement [ESQL] [SP]

Function

To open a previously declared cursor to access information from the database.

Syntax

OPEN *cursor-name*
... [**USING** [**DESCRIPTOR** *sqlda-name* | *host-variable*, ...]]
... [**WITH HOLD**]
... [**ISOLATION LEVEL** *n*]
... [**BLOCK** *n*]

Parameters

cursor-name:	*identifier* or *host-variable*
sqlda-name:	*identifier*

Permissions

Must have SELECT permission on all tables in a SELECT statement, or EXECUTE permission on the procedure in a CALL statement.

When the cursor is on a CALL statement, OPEN causes the procedure to execute until the first result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436
"RESUME statement" on page 533
"PREPARE statement" on page 519
"FETCH statement" on page 468
"RESUME statement" on page 533
"CLOSE statement" on page 373

Description

The OPEN statement opens the named cursor. The cursor must be previously declared.

By default, all cursors are automatically closed at the end of the current transaction (COMMIT or ROLLBACK). The optional WITH HOLD clause keeps the cursor open for subsequent transactions. It will remain open until the end of the current connection or until an explicit CLOSE statement is executed. Cursors are automatically closed when a connection is terminated.

The ISOLATION LEVEL clause allows this cursor to be opened at an isolation level different from the current setting of the ISOLATION_LEVEL option. All operations on this cursor will be performed at the specified isolation level regardless of the option setting. If this clause is not specified, then the cursor's isolation level for the entire time the cursor is open is the value of the ISOLATION_LEVEL option when the cursor is opened. See "How locking works" on page 382 of the book *Adaptive Server Anywhere User's Guide*.

The cursor is positioned before the first row (see "Cursors in Embedded SQL" on page 33 of the book *Adaptive Server Anywhere Programming Interfaces Guide* or "Using cursors in procedures and triggers" on page 251 of the book *Adaptive Server Anywhere User's Guide*).

Embedded SQL

The USING DESCRIPTOR **sqlda-name**, **host-variable** and BLOCK **n** formats are for Embedded SQL only.

If the cursor name is specified by an identifier or string, the corresponding DECLARE CURSOR must appear prior to the OPEN in the C program; if the cursor name is specified by a host variable, the DECLARE CURSOR statement must execute before the OPEN statement.

The optional USING clause specifies the host variables that will be bound to the place-holder bind variables in the SELECT statement for which the cursor has been declared.

The multiuser support fetches rows in blocks (more than 1 at a time). By default, the number of rows in a block is determined dynamically based on the size of the rows and how long it takes the database server to fetch each row. The application can specify a maximum number of rows that should be contained in a block by specifying the BLOCK clause. For example, if you are fetching and displaying 5 rows at a time, use **BLOCK 5**. Specifying **BLOCK 0** will cause one row at a time to be fetched, and also cause a FETCH RELATIVE 0 to always fetch the row again.

After successful execution of the OPEN statement, the *sqlerrd[3]* field of the SQLCA (SQLIOESTIMATE) will be filled in with an estimate of the number of input/output operations required to fetch all rows of the query. Also, the *sqlerrd[2]* field of the SQLCA (SQLCOUNT) will be filled in with either the actual number of rows in the cursor (a value greater than or equal to 0), or an estimate thereof (a negative number whose absolute value is the estimate). It will be the actual number of rows if the database server can compute it without counting the rows. The database can also be configured to always return the actual number of rows (see the ROW_COUNTS option in "SET OPTION statement" on page 553.), but this can be expensive.

Standards and compatibility

- **SQL/92** Embedded SQL use is an entry-level feature. Procedures use is a Persistent Stored Modules feature.
- **Sybase** The simple OPEN *cursor-name* syntax is supported by Adaptive Server Enterprise. None of the other clauses are supported in Adaptive Server Enterprise stored procedures. Open Client/Open Server supports the USING descriptor or host variable syntax.

Examples

- The following examples show the use of OPEN in Embedded SQL.

```
1. EXEC SQL OPEN employee_cursor;
```

```
2. EXEC SQL PREPARE emp_stat FROM
```

```
'SELECT empnum, empname FROM employee WHERE name
like ?';
EXEC SQL DECLARE employee_cursor CURSOR FOR
emp_stat;
EXEC SQL OPEN employee_cursor USING :pattern;
```

- The following example is from a procedure or trigger.

```
BEGIN
DECLARE cur_employee CURSOR FOR
   SELECT emp_lname
   FROM employee ;
DECLARE name CHAR(40) ;
OPEN cur_employee;
LOOP
FETCH NEXT cur_employee into name ;
    ...
END LOOP
CLOSE cur_employee;
END
```

OUTPUT statement [ISQL]

Function To output the current query results to a file.

Syntax

OUTPUT TO *filename*
  ... [**FORMAT** *output_format*]
  ... [**ESCAPE CHARACTER** *character*]
  ... [**DELIMITED BY** *string*]
  ... [**QUOTE** *string* [**ALL**]]
  ... [**COLUMN WIDTHS** (*integer*,...)]

Permissions None.

Side effects The current query results that are displayed in the Interactive SQL data window are repositioned to the top.

See also "SELECT statement" on page 542
"INPUT statement" on page 494

Description The OUTPUT statement copies the information retrieved by the current query to a file. The output format can be specified with the optional FORMAT clause. If no FORMAT clause is specified, the OUTPUT_FORMAT option setting is used (see "SET OPTION statement" on page 553).

The **current query** is the SELECT or INPUT statement which generated the information displayed in the Interactive SQL data window. The OUTPUT statement will report an error if there is no current query.

The default escape character for characters stored as hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

This can be changed using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter

```
... ESCAPE CHARACTER '!'
```

The DELIMITED BY and QUOTE clauses are for the ASCII output format only. The delimiter string will be placed between columns (default comma) and the quote string will be placed around string values (default '—single quote). If ALL is specified in the QUOTE clause, the quote string will be placed around all values, not just strings.

The COLUMN WIDTH clause is used to specify the column widths for the FIXED format output.

Allowable output formats are:

ASCII The output is an ASCII format file with one row per line in the file. All values are separated by commas, and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the DELIMITED BY and QUOTE clauses. If ALL is specified in the QUOTE clause, all values (not just strings) will be quoted.

Three other special sequences are also used. The two characters \n represent a newline character, \\ represents a single \, and the sequence \xDD represents the character with hexadecimal code DD. This is the default output format.

If you are exporting Java methods that have string return values, you must use the HEXADECIMAL OFF clause.

DBASEII The output is a dBASE II format file with the column definitions at the top of the file. Note that a maximum of 32 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

DBASEIII The output is a dBASE III format file with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

DIF The output is a file in the standard Data Interchange Format.

FIXED The output is fixed format with each column having a fixed width. The width for each column can be specified using the COLUMN WIDTH clause. No column headings are output in this format.

If the COLUMN WIDTH clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. The exception is that LONG VARCHAR and LONG BINARY data defaults to 32 Kb.

FOXPRO The output is a FoxPro format file (the FoxPro memo field is different than the dBASE memo field) with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

LOTUS The output is a Lotus WKS format worksheet. Column names will be put as the first row in the worksheet. Note that there are certain restrictions on the maximum size of Lotus WKS format worksheets that other software (such as Lotus 1-2-3) can load. There is no limit to the size of file Interactive SQL can produce.

SQL The output is a Interactive SQL INPUT statement required to recreate the information in the table.

TEXT The output is a TEXT format file which prints the results in columns with the column names at the top and vertical lines separating the columns. This format is similar to that used to display data in the Interactive SQL data window.

WATFILE The output is a WATFILE format file with the column definitions at the top of the file.

Exporting Java data

When exporting Java data, you may wish to export objects as binary, or you may want to export them as strings using the **`toString()`** method. You can control which way Java data is exported using the DESCRIBE_JAVA_FORMAT Interactive SQL option.

For example, consider the following script:

```
CREATE VARIABLE JavaString java.lang.String;
SET JavaString = NEW java.lang.String( 'TestVar' );
SELECT JavaString FROM dummy;
```

If you set describe_java_format to **Varchar**:

♦ The following command gives the hexadecimal representation of TestVar in the output file.

```
OUTPUT TO filename
```

♦ The following command gives a text representation of TestVar in the output file (possibly escaped).

```
OUTPUT TO filename HEXADECIMAL OFF
```

If you set describe_java_format to **binary**:

♦ The following command gives the hexadecimal representation of JavaString in the output file.

```
OUTPUT TO filename
```

♦ The following command gives the actual JavaString object in the output file (with escape sequences).

```
OUTPUT TO filename HEXADECIMAL OFF
```

For more information, see "DESCRIBE_JAVA_FORMAT option" on page 154.

Standards and compatibility

♦ **SQL/92** Vendor extension

♦ **Sybase** Not applicable

Examples

♦ Place the contents of the employee table in a file, in ASCII format.

```
SELECT *
FROM employee ;
OUTPUT TO employee.txt
```

```
FORMAT ASCII
```

- Output the contents of the **`toString()`** method of the **JProd** column to file:

  ```
  SELECT JProd>>toString()
  FROM jdba.product;
  OUTPUT TO d:\temp\temp.txt
  FORMAT ASCII HEXADECIMAL OFF
  ```

PARAMETERS statement [ISQL]

Function

To specify parameters to a Interactive SQL command file.

Syntax

PARAMETERS *parameter1*, *parameter2*, ...

Permissions

None.

Side effects

None.

See also

"READ statement" on page 527
"Running command files" on page 80 of the book *First Guide to SQL Anywhere Studio*

Description

The PARAMETERS statement names the parameters for a command file, so that they can be referenced later in the command file.

Parameters are referenced by putting:

```
{parameter1}
```

into the file where you wish the named parameter to be substituted. There must be no spaces between the braces and the parameter name.

If a command file is invoked with fewer than the required number of parameters, Interactive SQL prompts for values of the missing parameters.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** Not applicable.

Example

- The following Interactive SQL command file takes two parameters.

```
PARAMETERS department_id, file ;
SELECT emp_lname
FROM employee
WHERE dept_id = {department_id}
>#{file}.dat;
```

PREPARE statement [ESQL]

Function

To prepare a statement to be executed later or used for a cursor.

Syntax

```
PREPARE statement-name
      FROM statement
    ...[ DESCRIBE describe-type INTO [ [ SQL ] DESCRIPTOR ] descriptor ]
    ...[ WITH EXECUTE ]
```

Parameters

statement-name: *identifier* or *host-variable*

statement : *string* or *host-variable*

describe-type:

```
    { ALL |  BIND VARIABLES | INPUT | OUTPUT  |  SELECT LIST  }
    ...[
          LONG NAMES  [ [ OWNER. ]TABLE. ]COLUMN ]
        | WITH VARIABLE RESULT
      ]
```

Permissions

None.

Side effects

Any statement previously prepared with the same name is lost.

See also

"DECLARE CURSOR statement" on page 436
"DESCRIBE statement" on page 446
"OPEN statement" on page 511
"EXECUTE statement" on page 460
"DROP STATEMENT statement" on page 458

Description

The PREPARE statement prepares a SQL statement from the **statement** and associates the prepared statement with **statement-name**. This statement name is referenced to execute the statement, or to open a cursor if the statement is a SELECT statement. **Statement-name** may be a host variable of type **a_sql_statement_number** defined in the **sqlca.h** header file that is automatically included. If an identifier is used for the **statement-name**, only one statement per module may be prepared with this **statement-name**.

If a host variable is used for **statement-name**, it must have the type **short int**. There is a typedef for this type in **sqlca.h** called **a_sql_statement_number**. This type is recognized by the SQL preprocessor and can be used in a DECLARE section. The host variable is filled in by the database during the PREPARE statement, and need not be initialized by the programmer.

If the DESCRIBE INTO DESCRIPTOR clause is used, the prepared statement is described into the specified descriptor. The describe type may be any of the describe types allowed in the DESCRIBE statement.

If the WITH EXECUTE clause is used, the statement is executed if and only if it is not a CALL or SELECT statement, and it has no host variables. The statement is immediately dropped after a successful execution. If the PREPARE and the DESCRIBE (if any) are successful but the statement cannot be executed, a warning SQLCODE 111, SQLSTATE 01W08 is set, and the statement is not dropped.

The DESRIBE INTO DESCRIPTOR and WITH EXECUTE clauses may improve performance, because they cut down on the required client/server communication.

Describing variable result sets

The WITH VARIABLE RESULT clause is used to describe procedures that may have more than one result set, with different numbers or types of columns.

If WITH VARIABLE RESULT is used, the database server sets the SQLCOUNT value after the describe to one of the following values:

- **0** The result set may change: The procedure call should be described again following each OPEN statement.
- **1** The result set is fixed. No redescribing is required.

Statements that can be prepared

The following is a list of statements that can be PREPARED.

- ALTER
- CALL
- COMMENT ON
- CREATE
- DELETE
- DROP
- GRANT
- INSERT
- REVOKE
- SELECT
- SET OPTION
- UPDATE
- VALIDATE TABLE

Compatibility issue
For compatibility reasons, preparing COMMIT, PREPARE TO COMMIT, and ROLLBACK statements is still supported. However, we recommend that you do all transaction management operations with static Embedded SQL because certain application environments may require it. Also, other Embedded SQL systems do not support dynamic transaction management operations.

Drop statement after use
You should make sure that you DROP the statement after use. If you do not, the memory associated with the statement is not reclaimed.

Standards and compatibility

- **SQL/92** Entry level feature
- **Sybase** Supported by Open Client/Open Server.

Example

- The following statement prepares a simple query:

```
EXEC SQL PREPARE employee_statement FROM
'SELECT emp_lname FROM employee';
```

PREPARE TO COMMIT statement

Function To check whether a COMMIT can be performed.

Syntax **PREPARE TO COMMIT**

Permissions None.

Side effects None.

See also "COMMIT statement" on page 377
"ROLLBACK statement" on page 538

Description The PREPARE TO COMMIT statement tests whether a COMMIT can be performed successfully. The statement will cause an error if a COMMIT is not possible without violating the integrity of the database.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **Sybase** Not supported in Adaptive Server Enterprise.

Examples

- ♦ The following sequence of statements leads to an error because of foreign key checking on the employee table.

```
EXECUTE IMMEDIATE
   "SET OPTION wait_for_commit = 'on'";

EXECUTE IMMEDIATE "DELETE FROM employee
   WHERE emp_id = 160";

EXECUTE IMMEDIATE "PREPARE TO COMMIT";
```

- ♦ The following sequence of statements allows the delete to take place, even though it causes integrity violations. The PREPARE TO COMMIT statement returns an error.

```
SET OPTION wait_for_commit= 'ON' ;

DELETE
FROM department
WHERE dept_id = 100 ;

PREPARE TO COMMIT ;
```

PRINT statement [T-SQL]

Function

To display a message on the message window of the database server or return a message to the client window.

Syntax

PRINT *format-string* [, *arg-list*]

Authorization

Must be connected to the database.

Side effects

None.

See also

"MESSAGE statement" on page 509

Description

The PRINT statement returns a message to the client window if you are connected from an Open Client application or JDBC application. If you are connected from an Embedded SQL or ODBC application, the message is displayed on the database server window.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form *%nn!*, where *nn* is an integer between 1 and 20.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Examples

- The following procedure displays a message on the server message window:

  ```
  CREATE PROCEDURE print_test
  AS
  PRINT 'Procedure called successfully'
  ```

  The statement

  ```
  EXECUTE print_test
  ```

  returns the string Procedure called successfully to the client.

- The following statement illustrates the use of placeholders in the PRINT statement:

  ```
  DECLARE @var1 INT, @var2 INT

  SELECT @var1 = 3, @var2 = 5

  PRINT 'Variable 1 = %1!, Variable 2 = %2!', @var1,
  @var2
  ```

PUT statement [ESQL] [SP]

Function

To insert a row into the table(s) specified by the cursor. See "SET statement" on page 546 for putting LONG VARCHAR or LONG BINARY values into the database.

Syntax

PUT *cursor-name*
... [**USING DESCRIPTOR** *sqlda-name*
| **FROM** *host-variable-list*]

... [**INTO DESCRIPTOR** *into-sqlda-name*
| **INTO** *into-host-variable-list*]

... [**ARRAY** :*nnn*]

Parameters

cursor-name:	*identifier* or *host-variable*
sqlda-name:	*identifier*
host-variable-list:	may contain indicator variables

Permissions

Must have INSERT permission.

Side effects

None.

See also

"UPDATE statement" on page 572
"UPDATE (positioned) statement" on page 575
"DELETE statement" on page 442
"DELETE (positioned) statement" on page 444
"INSERT statement" on page 498

Description

Inserts a row into the named cursor. Values for the columns are taken from the SQLDA or the host variable list, in a one-to-one correspondence with the columns in the INSERT statement (for an INSERT cursor) or the columns in the select list (for a SELECT cursor).

If the **sqldata** pointer in the SQLDA is the null pointer, no value is specified for that column. If the column has a DEFAULT VALUE associated with it, that will be used; otherwise, a NULL value will be used. If no values are specified for any of the columns of one particular table involved in the cursor, no row will be inserted into that table.

The second SQLDA or host variable list contains the results of the PUT statement.

The optional ARRAY clause can be used to carry out wide puts, which insert more than one row at a time and which may improve performance. The value **nnn** is the number of rows to be inserted. The SQLDA must contain **nnn * (columns per row)** variables. The first row is placed in SQLDA variables 0 to **(columns per row)-1**, and so on.

One table only
Values can be specified for columns of one table only. Inserting into two different tables through a cursor is not supported.

Inserting into a cursor
When inserting into a cursor, the position of the inserted row is unpredictable. If the cursor involves a temporary table, the inserted record will not show up in the current cursor at all.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Open Client/Open Server.

Example

- The following statement illustrates the use of PUT in Embedded SQL:

```
EXEC SQL PUT cur_employee FROM :emp_id, :emp_lname;
```

RAISERROR statement [T-SQL]

Function To signal an error, and send a message to the client.

Syntax **RAISERROR** *error-number* [*format-string*] [, *arg-list*]

Authorization Must be connected to the database.

Side effects None.

See also "CREATE TRIGGER statement" on page 427

Description The RAISERROR statement allows user-defined errors to be signaled, and sends a message on the client.

The *error-number* is a five-digit integer greater than 17000.

If *format-string* is not supplied or is empty, the error number is used to locate an error message in the system tables. Adaptive Server Enterprise obtains messages 17000-19999 from the SYSMESSAGES table. In Adaptive Server Anywhere this table is an empty view, so errors in this range should provide a format string. Messages for error numbers of 20000 or greater are obtained from the SYS.SYSUSERMESSAGES table. The error number is stored in the global variable **@@error**.

The *format-string* is a maximum of 70 bytes long. In Adaptive Server, Anywhere the *format-string* length can be up to 255 bytes.

The extended values supported by the Adaptive Server Enterprise RAISERROR statement are not supported in Adaptive Server Anywhere.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form *%nn!*, where *nn* is an integer between 1 and 20.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

- The following statement raises error 99999, which is in the range for user-defined errors, and sends a message to the client.

  ```
  RAISERROR 99999 'Invalid entry for this column:
  %1!', @val
  ```

  There is no comma between the *error-number* and the *format-string* parameters. The first item following a comma is interpreted as the first item in the argument list.

READ statement [ISQL]

Function To read Interactive SQL statements from a file.

Syntax **READ** *filename* [*parameters*]

Permissions None.

Side effects None.

See also "PARAMETERS statement" on page 518

Description The READ statement reads a sequence of Interactive SQL statements from the named file. This file can contain any valid Interactive SQL statement including other READ statements. READ statements can be nested to any depth. To find the command file, Interactive SQL will first search the current directory, then the directories specified in the environment variable **SQLPATH**, then the directories specified in the environment variable **PATH**. If the named file has no file extension, Interactive SQL searches each directory for the same file name with the extension *sql*.

Parameters can be listed after the name of the command file. These parameters correspond to the parameters named on the PARAMETERS statement at the beginning of the statement file (see "PARAMETERS statement" on page 518). Interactive SQL will then substitute the corresponding parameter wherever the source file contains

```
{ parameter-name }
```

where *parameter-name* is the name of the appropriate parameter.

The parameters passed to a command file can be identifiers, numbers, quoted identifiers, or strings. When quotes are used around a parameter, the quotes are put into the text during the substitution. Parameters which are not identifiers, numbers, or strings (contain spaces or tabs) must be enclosed in square brackets (**[ ]**). This allows for arbitrary textual substitution in the command file.

If not enough parameters are passed to the command file, Interactive SQL prompts for values for the missing parameters.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Examples

- The following are examples of the READ statement.

```
READ status.rpt '160'
```

```
READ birthday.sql [>= '1988-1-1'] [<= '1988-1-30']
```

READTEXT statement [T-SQL]

Function

Reads text and image values, starting from a specified offset and reading a specified number of bytes.

Syntax

READTEXT *table-name.column-name*
   ... *text-pointer offset size*
   ... [**HOLDLOCK**]

Authorization

Select permissions on the table.

Side effects

None.

See also

"WRITETEXT statement" on page 580

Description

READTEXT is used to read image and text values from the database. You cannot perform READTEXT operations on views.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

  Adaptive Server Enterprise supports the following clause, which is not supported by Adaptive Server Anywhere:

  ```
  USING { BYTES | CHARS | CHARACTERS }
  ```

  These options are identical for all single-byte character sets. Adaptive Server Anywhere uses **bytes** only, which is the Adaptive Server Enterprise default setting.

  Adaptive Server Enterprise also provides isolation level control in the READTEXT statement. This is not supported in Adaptive Server Anywhere.

RELEASE SAVEPOINT statement

Function

Release a savepoint within the current transaction.

Syntax

RELEASE SAVEPOINT [*savepoint-name*]

Permissions

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

"SAVEPOINT statement" on page 541
"ROLLBACK TO SAVEPOINT statement" on page 539

Description

Release a savepoint. The *savepoint-name* is an identifier specified on a SAVEPOINT statement within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is released.

For a description of savepoints, see "Savepoints within transactions" on page 418 of the book *Adaptive Server Anywhere User's Guide*. Releasing a savepoint does not do any type of COMMIT. It simply removes the savepoint from the list of currently active savepoints.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise. A similar feature is available in an Adaptive Server Enterprise-compatible manner using nested transactions.

REMOVE statement

Function

This statement removes a class, a package, or a jar file from a database. When a class is removed it is no longer available for use as a column or variable type.

The class, package, or jar must already be installed.

Syntax

REMOVE JAVA *classes_to_remove*

Parameters

classes_to_remove:
  CLASS *java_class_name* [, *java_class_name*,...]
  | **PACKAGE** *java_package_name* [, *java_package_name*,...]
  | **JAR** *jar_name* [, *jar_name*,...] [**RETAIN CLASSES**]

jar_name:
  character_string_expression

Permissions

Must have DBA authority.

Not supported on Windows CE.

Description

java_class_name The name of one or more Java class to be removed. These classes must be installed classes in the current database.

java_package_name The name of one or more Java packages to be removed. These packages must be in the current database.

jar_name A character string value of maximum length 255.

Each *jar_name* must be equal to the *jar_name* of a retained jar in the current database. Equality of *jar_name* is determined by the character string comparison rules of the SQL system.

If **JAR...RETAIN CLASSES** is specified, the specified jars are no longer retained in the database, and the retained classes have no associated jar. If **RETAIN CLASSES** is specified, this is the only action of the **REMOVE** statement.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise. A similar feature is available in an Adaptive Server Enterprise-compatible manner using nested transactions.

Examples

- The following statement removes a Java class named Demo from the current database.

```
REMOVE JAVA CLASS Demo
```

RESIGNAL statement

Function

Resignal an exception condition.

Syntax

RESIGNAL [*exception-name*]

Permissions

None.

Side effects

None.

See also

"SIGNAL statement" on page 560
"BEGIN... END statement" on page 361
"Using exception handlers in procedures and triggers" on page 261 of the book *Adaptive Server Anywhere User's Guide*
"RAISERROR statement" on page 526

Description

Within an exception handler, RESIGNAL allows you to quit the compound statement with the exception still active, or to quit reporting another named exception. The exception will be handled by another exception handler or returned to the application. Any actions by the exception handler before the RESIGNAL are undone.

Standards and compatibility

♦ **SQL/92** Persistent stored module feature.

♦ **Sybase** Not supported in Adaptive Server Enterprise. Error handling in Transact-SQL procedures is carried out using the RAISERROR statement.

Example

♦ The following fragment returns all exceptions except Column Not Found to the application.

```
...
DECLARE COLUMN_NOT_FOUND EXCEPTION
   FOR SQLSTATE '52003';
...
EXCEPTION
WHEN COLUMN_NOT_FOUND THEN
SET message='Column not found' ;
WHEN OTHERS THEN
RESIGNAL ;
```

RESTORE statement

Function Restore a backed up database from an archive.

Syntax

RESTORE DATABASE *database_location*
  FROM *archive_root*
  [[**RENAME** *dbspace_name* **TO** *new_dbspace_path*] ...]

Permissions Must be connected to the utility database.

Side effects None.

See also "BACKUP statement" on page 359

Description

database_location Specifies the location for the main database file.

Each RESTORE operation updates a history file called *backup.syb*, which is held in the same directory as your database server executable file.

Archive backups are only supported on NT and Unix platforms.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

RESUME statement

Function

To resume a procedure following a query.

Syntax 1

RESUME *cursor-name*

Syntax 2

RESUME [**ALL**]

Parameters

cursor-name:	*identifier*
cursor-name:	*identifier* or *host-variable*

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

"DECLARE CURSOR statement" on page 436
"Returning results from procedures" on page 246 of the book *Adaptive Server Anywhere User's Guide*

Description

This statement resumes execution of a procedure that returns result sets. The procedure executes until the next result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set. This warning is also set when you RESUME a cursor for a SELECT statement.

The Interactive SQL RESUME statement (Format 2) resumes the current procedure. If ALL is not specified, executing RESUME displays the next result set or, if no more result sets are returned, completes the procedure.

The Interactive SQL RESUME ALL statement cycles through all result sets in a procedure, without displaying them, and completes the procedure. This is useful mainly in testing procedures.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Examples

- Embedded SQL example

```
1. EXEC SQL RESUME cur_employee;
2. EXEC SQL RESUME :cursor_var;
```

- Interactive SQL examples

```
CALL sample_proc() ;
RESUME ALL;
```

RETURN statement

Function To exit from a function or procedure unconditionally, optionally providing a return value. Statements following RETURN are not executed.

Syntax **RETURN** [(*expression*)]

Permissions None.

Side effects None.

See also "CREATE FUNCTION statement" on page 397
"CREATE PROCEDURE statement" on page 403
"BEGIN... END statement" on page 361

Description A RETURN statement causes an immediate exit from the function or procedure. If *expression* is supplied, the value of *expression* is returned as the value of the function or procedure.

Within a function, the expression should be of the same data type as the function's RETURNS data type.

RETURN is used in procedures for Transact-SQL-compatibility, and is used to return an integer error code.

Standards and compatibility

- **SQL/92** Persistent stored module feature.
- **Sybase** Transact-SQL procedures use the RETURN statement to return an integer error code.

Example

- The following function returns the product of three numbers:

```
CREATE FUNCTION product ( a numeric,
                          b numeric ,
                          c numeric)
RETURNS numeric
BEGIN
   RETURN ( a * b * c ) ;
END
```

- Calculate the product of three numbers:

```
SELECT product (2, 3, 4)
```

product(2, 3, 4)
24

- The following procedure uses the RETURN statement to avoid executing a complex query if it is meaningless:

```
CREATE PROCEDURE customer_products
```

```
( in customer_id integer DEFAULT NULL)
RESULT ( id integer, quantity_ordered integer )
BEGIN
   IF customer_id NOT IN (SELECT id FROM customer)
   OR customer_id IS NULL THEN
      RETURN
   ELSE
      SELECT product.id,sum(
         sales_order_items.quantity )
      FROM  product,
            sales_order_items,
            sales_order
      WHERE sales_order.cust_id=customer_id
      AND sales_order.id=sales_order_items.id
      AND sales_order_items.prod_id=product.id
      GROUP BY product.id
   END IF
END
```

REVOKE statement

Function

To remove permissions for specified user(s).

Syntax 1

```
REVOKE
    {      CONNECT
         | DBA
         | INTEGRATED LOGIN
         | GROUP
         | MEMBERSHIP IN GROUP userid,...
         | RESOURCE
    }
... FROM userid,...
```

Syntax 2

```
REVOKE
    {      ALL [PRIVILEGES]
         | ALTER
         | DELETE
         | INSERT
         | REFERENCES  [ ( column-name,...) ]
         | SELECT  [ ( column-name,...) ]
         | UPDATE  [ ( column-name,...) ]
    }
... ON [ owner.]table-name FROM userid,...
```

Syntax 3

```
REVOKE EXECUTE ON [ owner.]procedure-name FROM userid,...
```

Permissions

Must be the grantor of the permissions that are being revoked or have DBA authority.

If you are revoking CONNECT permissions or table permissions from another user, the other user must not be connected to the database.

Side effects

Automatic commit.

See also

"GRANT statement" on page 484

Description

The REVOKE statement is used to remove permissions that were given using the GRANT statement. Form 1 is used to revoke special user permissions. Form 2 is used to revoke table permissions. Form 3 is used to revoke permission to execute a procedure. REVOKE CONNECT is used to remove a user ID from a database. REVOKE GROUP will automatically REVOKE MEMBERSHIP from all members of the group.

Standards and compatibility

- **SQL/92** Syntax 1 is a vendor extension. Syntax 2 is an entry-level feature. Syntax 3 is a Persistent Stored Module feature.

- **Sybase** Syntax 2 and 3 are supported by Adaptive Server Enterprise. Syntax 1 is not supported by Adaptive Server Enterprise. User management and security models are different for Adaptive Server Anywhere and Adaptive Server Enterprise.

Examples

- Prevent user Dave from updating the employee table.

```
REVOKE UPDATE ON employee FROM dave ;
```

- Revoke resource permissions from user Jim.

```
REVOKE RESOURCE FROM Jim ;
```

- Revoke integrated login mapping from user profile name Administrator

```
REVOKE INTEGRATED LOGIN FROM Administrator ;
```

- Disallow the Finance group from executing the procedure **sp_customer_list**.

```
REVOKE EXECUTE ON sp_customer_list
FROM finance ;
```

- Drop user ID **FranW** from the database.

```
REVOKE CONNECT FROM FranW
```

ROLLBACK statement

Function To undo any changes made since the last COMMIT or ROLLBACK.

Syntax **ROLLBACK** [**WORK**]

Permissions Must be connected to the database.

Side effects Closes all cursors not opened WITH HOLD.

See also "COMMIT statement" on page 377
"ROLLBACK TO SAVEPOINT statement" on page 539

Description The ROLLBACK statement ends a logical unit of work (transaction) and undoes all changes made to the database during this transaction. A **transaction** is the database work done between COMMIT or ROLLBACK statements on one database connection.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Supported by Adaptive Server Enterprise.

ROLLBACK TO SAVEPOINT statement

Function

To cancel any changes made since a SAVEPOINT.

Syntax

ROLLBACK TO SAVEPOINT [*savepoint-name*]

Permissions

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

"SAVEPOINT statement" on page 541
"RELEASE SAVEPOINT statement" on page 529
"ROLLBACK statement" on page 538

Description

The ROLLBACK TO SAVEPOINT statement will undo any changes that have been made since the SAVEPOINT was established. Changes made prior to the SAVEPOINT are not undone; they are still pending. For a description of savepoints, see "Savepoints within transactions" on page 418 of the book *Adaptive Server Anywhere User's Guide*.

The *savepoint-name* is an identifier that was specified on a SAVEPOINT statement within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is used. Any savepoints since the named savepoint are automatically released.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Savepoints are not supported by Adaptive Server Enterprise. To implement similar features in an Adaptive Server Enterprise-compatible manner, you can use nested transactions.

ROLLBACK TRIGGER statement

Function To undo any changes made in a trigger.

Syntax **ROLLBACK TRIGGER** [**WITH** *raiserror-statement*]

Permissions Must be connected to the database.

Side effects None

See also "CREATE TRIGGER statement" on page 424
"ROLLBACK statement" on page 538
"ROLLBACK TO SAVEPOINT statement" on page 539
"RAISERROR statement" on page 526

Description The ROLLBACK TRIGGER statement rolls back the work done in a trigger, including the data modification that caused the trigger to fire.

Optionally, a RAISERROR statement can be issued. If a RAISERROR statement is issued, an error is returned to the application. If no RAISERROR statement is issued, no error is returned.

If a ROLLBACK TRIGGER statement is used within a nested trigger and without a RAISERROR statement, only the innermost trigger and the statement which caused it to fire are undone.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

SAVEPOINT statement

Function

To establish a savepoint within the current transaction.

Syntax

SAVEPOINT [*savepoint-name*]

Permissions

None.

Side effects

None.

See also

"RELEASE SAVEPOINT statement" on page 529
"ROLLBACK TO SAVEPOINT statement" on page 539

Description

Establish a savepoint within the current transaction. The *savepoint-name* is an identifier that can be used in an RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT statement. All savepoints are automatically released when a transaction ends. See "Savepoints within transactions" on page 418 of the book *Adaptive Server Anywhere User's Guide*.

Savepoints that are established while a trigger or atomic compound statement is executing are automatically released when the atomic operation ends.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise. To implement similar features in an Adaptive Server Enterprise-compatible manner, you can use nested transactions.

SELECT statement

Function

To retrieve information from the database.

Syntax

SELECT [**ALL** | **DISTINCT**] [**FIRST** | **TOP** *n*] *select-list*
...[**INTO** { *host-variable-list* | *variable-list* }]
...[**FROM** *table-list*]
...[**WHERE** *search-condition*]
...[**GROUP BY** *column-name* | *alias* | *function* ...]
...[**HAVING** *search-condition*]
...[**ORDER BY** { *expression* | *integer* } [**ASC** | **DESC**], ...]

Parameters

select-list:
table-name
| *expression* [[**AS**] *alias-name*]
| *

Permissions

Must have SELECT permission on the named tables and views.

Side effects

None.

See also

"CREATE VIEW statement" on page 430
"DECLARE CURSOR statement" on page 436
"Expressions" on page 183
"FETCH statement" on page 468
"FROM clause" on page 476
"OPEN statement" on page 511
"Search conditions" on page 194
"UNION operation" on page 569

Description

The SELECT statement is used for retrieving results from the database.

A SELECT statement can be used in Interactive SQL to browse data in the database, or to export data from the database to an external file.

A SELECT statement can also be used in procedures and triggers or in Embedded SQL. The SELECT statement with an INTO clause is used for retrieving results from the database when the SELECT statement only returns one row. For multiple row queries, you must use cursors.

The INTO clause with *host-variable-list* is used in Embedded SQL only.

The INTO clause with *variable-list* is used in procedures and triggers only.

A SELECT statement can also be used to return a result set from a procedure. The various parts of the SELECT statement are described below:

ALL or DISTINCT All (the default) returns all rows that satisfy the clauses of the SELECT statement. If DISTINCT is specified, duplicate output rows are eliminated. This is called the **projection** of the result of the statement. Because many, statements take significantly longer to execute when DISTINCT is specified, You should reserve DISTINCT for cases where it is necessary.

If DISTINCT is used, the statement cannot contain an aggregate function with a DISTINCT parameter.

FIRST or TOP These keywords are principally for use with ORDER BY queries. You can explicitly retrieve only the first row of a query or the first *n* rows of a query.

The FIRST and TOP keywords cannot be used in a derived table query. You should not use the keywords in view definitions.

select list The select list is a list of expressions, separated by commas, specifying what will be retrieved from the database. Asterisk (*) means to select all columns of all tables in the FROM clause (if you specify a **table-name,** all comlumns of the specifed table are selected).

Aggregate functions are allowed in the select list (see "SQL Functions" on page 267). Subqueries are also allowed in the select list (see "Expressions" on page 183). Each subquery must be within parentheses.

Alias-names can be used throughout the query to represent the aliased expression.

Alias names are also displayed by Interactive SQL at the top of each column of output from the SELECT statement. If the optional alias name is not specified after an expression, Interactive SQL will display the expression itself.

INTO host-variable-list This clause is used in Embedded SQL only. It specifies where the results of the SELECT statement will go. There must be one host-variable item for each item in the select list. Select list items are put into the host variables in order. An indicator host variable is also allowed with each **host-variable,** so the program can tell if the select list item was NULL.

INTO variable-list This clause is used in procedures and triggers only. It specifies where the results of the SELECT statement will go. There must be one variable for each item in the select list. Select list items are put into the variables in order.

FROM table-list Rows are retrieved from the tables and views specified in the **table list**. Joins can be specified using join operators. For more information, see "FROM clause" on page 476. A SELECT statement with no FROM clause can be used to display the values of expressions not derived from tables. For example:

```
SELECT @@version
```

displays the value of the global variable @@version. This is equivalent to:

```
SELECT @@version
FROM DUMMY
```

WHERE search-condition This clause specifies which rows will be selected from the tables named in the FROM clause. It is also used to do joins between multiple tables. This is accomplished by putting a condition in the WHERE clause that relates a column or group of columns from one table with a column or group of columns from another table. Both tables must be listed in the FROM clause.

☞ For more information, see "Search conditions" on page 194.

GROUP BY { column-name | alias | function }, ... You can group by columns, or alias names, or functions. GROUP BY expressions must also appear in the select list. The result of the query contains one row for each distinct set of values in the named columns, aliases, or functions. The resulting rows are often referred to as **groups** since there is one row in the result for each group of rows from the table list. For the sake of GROUP BY, all NULL values are treated as identical. Aggregate functions can then be applied to these groups to get meaningful results.

When GROUP BY is used, the select list, HAVING clause, and ORDER BY clause cannot reference any identifiers except those named in the GROUP BY clause. The exception is that the select list and HAVING clause may contain aggregate functions.

HAVING search-condition This clause selects rows based on the group values and not on the individual row values. The HAVING clause can only be used if either the statement has a GROUP BY clause or the select list consists solely of aggregate functions. Any column names referenced in the HAVING clause must either be in the GROUP BY clause or be used as a parameter to an aggregate function in the HAVING clause.

ORDER BY expression, ... This clause sorts the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order (the default) or DESC for descending order. If the expression is an integer **n**, then the query results will be sorted by the **n**'th item in the select list.

In Embedded SQL, the SELECT statement is used for retrieving results from the database and placing the values into host variables via the INTO clause. The SELECT statement must return only one row. For multiple row queries, you must use cursors.

Standards and compatibility

- **SQL/92** Entry level feature. check clauses.
- **Sybase** Supported by Adaptive Server Enterprise, with some differences in syntax.

Examples

- List all the tables and views in the system catalog.

```
SELECT tname
FROM SYS.SYSCATALOG
WHERE tname LIKE 'SYS%' ;
```

- List all customers and the total value of their orders.

```
SELECT company_name,
   CAST( sum(sales_order_items.quantity *
   product.unit_price) AS INTEGER) VALUE
FROM customer
   LEFT OUTER JOIN sales_order
   LEFT OUTER JOIN sales_order_items
   LEFT OUTER JOIN product
GROUP BY company_name
ORDER BY VALUE DESC
```

- How many employees are there?

```
SELECT count(*)
FROM Employee ;
```

- The following statement shows an Embedded SQL SELECT statement:

```
SELECT count(*) INTO :size FROM employee
```

SET statement

Function
To assign a value to a SQL variable.

Syntax
SET *identifier* = *expression*

Permissions
None.

Side effects
None.

See also
"CREATE VARIABLE statement" on page 428
"DROP VARIABLE statement" on page 459
"Expressions" on page 183

Description
The SET statement assigns a new value to a variable that was previously created using the CREATE VARIABLE statement.

A variable can be used in a SQL statement anywhere a column name is allowed. If there is no column name that matches the identifier, the database server checks to see if there is a variable that matches and uses its value.

Variables are local to the current connection, and disappear when you disconnect from the database or use the DROP VARIABLE statement. They are not affected by COMMIT or ROLLBACK statements.

Variables are necessary for creating large text or binary objects for INSERT or UPDATE statements from Embedded SQL programs, because Embedded SQL host variables are limited to 32,767 bytes.

Standards and compatibility

♦ **SQL/92** Persistent stored module feature.

♦ **Sybase** Not supported. In Adaptive Server Enterprise, variables are assigned using the SELECT statement with no table, a Transact-SQL syntax that is also supported by Adaptive Server Anywhere. The SET statement is used to set database options in Adaptive Server Enterprise.

Example

♦ The following code fragment could be used to insert a large text value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
char buffer[5001];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_text LONG VARCHAR;
EXEC SQL SET hold_text = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( buffer, 1, 5000, fp );
   if( size <= 0 ) break;

   /* buffer must be null-terminated */
   buffer[size] = '\0';
   /* add data to blob using concatenation */
   EXEC SQL SET hold_text = hold_text || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES ( 1, hold_text );
EXEC SQL DROP VARIABLE hold_text;
```

- The following code fragment could be used to insert a large binary value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY( 5000 ) buffer;
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE VARIABLE hold_blob LONG BINARY;
EXEC SQL SET hold_blob = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( &(buffer.array), 1, 5000, fp );
   if( size <= 0 ) break;
   buffer.len = size;

   /* add data to blob using concatenation
      Note that concatenation works for
      binary data too! */
   EXEC SQL SET hold_blob = hold_blob || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES ( 1, hold_blob );
EXEC SQL DROP VARIABLE hold_blob;
```

SET statement [T-SQL]

Function To set database options for the current connection, in an Adaptive Server Enterprise-compatible manner.

Syntax **SET** *option-name option-value*

Permissions None.

Side effects None.

See also "SET OPTION statement" on page 553

Description The available options are as follows:

Option name	Option value
ANSINULL	**ON** | **OFF**
ANSI_PERMISSIONS	**ON** | **OFF**
CLOSE_ON_ENDTRANS	**ON** | **OFF**
QUOTED_IDENTIFIER	**ON** | **OFF**
ROWCOUNT	*integer*
SELF_RECURSION	**ON** | **OFF**
STRING_RTRUNCATION	**ON** | **OFF**
TEXTSIZE	*integer*
TRANSACTION_ISOLATION_LEVEL	**0** | **1** | **2** | **3**

Database options in Adaptive Server Anywhere are set using the SET OPTION statement. However, Adaptive Server Anywhere also provides support for the Adaptive Server Enterprise SET statement for options that are particularly useful for compatibility.

The SET statement sets the option for the duration of the current connection only. It is equivalent to SET TEMPORARY OPTION.

The following options can be set using the Transact-SQL SET statement in Adaptive Server Anywhere as well as in Adaptive Server Enterprise:

- **SET ANSINULL { ON | OFF }** The default behavior for comparing values to NULL in Adaptive Server Anywhere and Adaptive Server Enterprise is different. Setting ANSINULL to OFF provides Transact-SQL compatible comparisons with NULL.

- **SET ANSI_PERMISSIONS { ON | OFF }** The default behavior in Adaptive Server Anywhere and Adaptive Server Enterprise regarding permissions required to carry out an UPDATE or DELETE containing a column reference is different. Setting ANSI_PERMISSIONS to OFF provides Transact-SQL-compatible permissions on UPDATE and DELETE.
- **SET CLOSE_ON_ENDTRANS { ON | OFF }** The default behavior in Adaptive Server Anywhere and Adaptive Server Enterprise for closing cursors at the end of a transaction is different. Setting CLOSE_ON_ENDTRANS to OFF provides Transact-SQL compatible behavior.
- **SET QUOTED_IDENTIFIER { ON | OFF }** Controls whether strings enclosed in double quotes are interpreted as identifiers (ON) or as literal strings (OFF). For information about this option, see "Setting options for Transact-SQL compatibility" on page 794 of the book *Adaptive Server Anywhere User's Guide*.
- **SET ROWCOUNT** *integer* The Transact-SQL ROWCOUNT option limits the number of rows fetched for any cursor to the specified integer. This includes rows fetched by re-positioning the cursor. Any fetches beyond this maximum return a warning. The option setting is considered when returning the estimate of the number of rows for a cursor on an OPEN request.

  SET ROWCOUNT also limits the number of rows affected by a searched UPDATE or DELETE statement to *integer*. This might be used, for example, to allow COMMIT statements to be performed at regular intervals to limit the size of the rollback log and lock table. The application (or procedure) would need to provide a loop to cause the update/delete to be re-issued for rows that are not affected by the first operation. A simple example is given below:

```
begin
   declare @count integer

   set rowcount 20

   while(1=1) begin
      update employee set emp_lname='new_name'
      where emp_lname <> 'old_name'

      /* Stop when no rows changed */
      select @count = @@rowcount
      if @count = 0 break
      print string('Updated ',
               @count,' rows; repeating...')
      commit
   end
```

```
    set rowcount 0
end
```

In Adaptive Server Anywhere, if the ROWCOUNT setting is greater than the number of rows that Interactive SQL can display, Interactive SQL may do some extra fetches to reposition the cursor. Thus, the number of rows actually displayed may be less than the number requested. Also, if any rows are re-fetched due to truncation warnings, the count may be inaccurate.

A value of zero resets the option to get all rows.

- **SET SELF_RECURSION { ON | OFF }** The self_recursion option is used within triggers to enable (ON) or prevent (OFF) operations on the table associated with the trigger from firing other triggers.
- **SET STRING_RTRUNCATION { ON | OFF }** The default behavior in Adaptive Server Anywhere and Adaptive Server Enterprise when non-space characters are truncated on assigning SQL string data is different. Setting STRING_RTRUNCATION to ON provides Transact-SQL-compatible string comparisons.
- **SET TEXTSIZE** Specifies the maximum size (in bytes) of text or image type data to be returned with a select statement. The @@textsize global variable stores the current setting. To reset to the default size (32K), which is also the maximum setting, use the command:

  ```
  set textsize 0
  ```
- **SET TRANSACTION-ISOLATION-LEVEL { 0 | 1 | 2 | 3 }** Sets the locking isolation level for the current connection, as described in "Isolation levels and consistency" on page 386 of the book *Adaptive Server Anywhere User's Guide*. For Adaptive Server Enterprise, only 1 and 3 are valid options. For Adaptive Server Anywhere, any of 0, 1, 2, or 3 is a valid option.

In addition, the following SET statement is allowed by Adaptive Server Anywhere for compatibility, but has no effect:

- **SET PREFETCH {ON | OFF}**

Standards and compatibility

- **SQL/92** Transact-SQL extension
- **Sybase** Adaptive Server Anywhere supports a subset of the Adaptive Server Enterprise database options.

SET CONNECTION statement [ISQL][ESQL]

Function To change the active database connection.

Syntax **SET CONNECTION** [*connection-name*]

Parameters *connection-name*:

identifier, *string,* or *host-variable*

Permissions None.

Side effects None.

See also "CONNECT statement" on page 381
"DISCONNECT statement" on page 450

Description The SET CONNECTION statement changes the active database connection to **connection-name**. The current connection state is saved, and will be resumed when it again becomes the active connection. If **connection-name** is omitted and there is a connection that was not named, that connection becomes the active connection.

Cursors and connections
When cursors are opened in Embedded SQL, they are associated with the current connection. When the connection is changed, the cursor names will not be accessible. The cursors remain active and in position, and will become accessible when the associated connection becomes active again.

Standards and compatibility

- **SQL/92** Interactive SQL use is a vendor extension. Embedded SQL is a Full level feature.
- **Sybase** Supported by Open Client/Open Server.

Example

- The following example is in Embedded SQL.

```
EXEC SQL SET CONNECTION :conn_name;
```

- From Interactive SQL, set the current connection to the connection named conn1.

```
SET CONNECTION conn1 ;
```

SET DESCRIPTOR statement [ESQL]

Function

Describes the variables in a SQL descriptor area, and places data into the descriptor area.

Syntax

SET DESCRIPTOR *descriptor-name*
...{ **COUNT** = { *integer* | *hostvar* } | **VALUE** *n* *assignment* [,...] }

Parameters

assignment:
{ **TYPE** | **SCALE** | **PRECISION** | **LENGTH** | **INDICATOR** }
... = { *integer* | *hostvar* }
| **DATA** = *hostvar*

Permissions

None.

Side effects

None.

See also

"ALLOCATE DESCRIPTOR statement" on page 342
"DEALLOCATE DESCRIPTOR statement" on page 433
"The SQL descriptor area (SQLDA)" on page 45 of the book *Adaptive Server Anywhere Programming Interfaces Guide*

Description

The SET DESCRIPTOR statement is used to describe the variables in a descriptor area, and to place data into the descriptor area.

The SET ... COUNT statement sets the number of described variables within the descriptor area. The value for count must not exceed the number of variables specified when the descriptor area was allocated.

The value *n* specifies the variable in the descriptor area upon which the assignment(s) will be performed.

Type checking is performed when doing SET ... DATA, to ensure that the variable in the descriptor area has the same type as the host variable.

If an error occurs, the code is returned in the SQLCA.

Standards and compatibility

- **SQL/92** Intermediate level feature.
- **Sybase** Supported by Open Client/Open Server.

Example

- For an example, see "ALLOCATE DESCRIPTOR statement" on page 342.

SET OPTION statement

Function

To change database options.

Syntax

SET [**TEMPORARY**] **OPTION**
... [*userid*.| **PUBLIC**.]*option-name* = [*option-value*]

Parameters

userid:	{ *identifier* | *string* | *host-variable* }
option-name:	{ *identifier* | *string* | *host-variable* }
option-value:	{ *host-variable* (indicator allowed) | *string* | *identifier* | *number* }

Permissions

None required to set your own options.

DBA authority is required to set database options for another user or PUBLIC.

Side effects

If TEMPORARY is not specified, an automatic commit is performed.

See also

"General database options" on page 132
"Transact-SQLcompatibility options" on page 134
"Replication options" on page 137
"SET OPTION statement " on page 556

Description

The SET OPTION statement is used to change options that affect the behavior of the database server. Setting the value of an option can change the behavior for all users or only for an individual user. The scope of the change can be either temporary or permanent.

The classes of options are:

- ♦ General database options
- ♦ Transact-SQL compatibility
- ♦ Replication database options

For a listing and description of all available options, see "Database Options" on page 127.

Scope of database options

If you specify a user ID, the option value applies to that user (or, for a group user ID, the members of that group). If you specify **PUBLIC**, the option value applies to all users who don't have an individual setting for the option. By default, the option value applies to the currently logged on user ID that issued the **SET OPTION** statement..

For example, the following statement applies an option change to the user DBA, if DBA is the user issuing the SQL statement:

```
SET OPTION login_mode = mixed
```

However the following statement applies the change to the **PUBLIC** user ID, a user group to which all users belong.

```
SET OPTION Public.login_mode = standard
```

Only users with DBA privileges have the authority to set an option for the **PUBLIC** user ID.

In Embedded SQL, database options can be set only temporarily.

Changing the value of an option for the **PUBLIC** user ID sets the value of the option for any user which has not SET their own value. Option values cannot be set for an individual user ID unless there is already a **PUBLIC** user ID setting for that option.

Users can use the SET OPTION statement to change the values for their own user ID. Setting the value of an option for a user id other then your own is permitted only if you have DBA authority.

Temporary changes

Adding the TEMPORARY keyword to the SET OPTION statement changes the duration that the change takes effect. By default, the option value is permanent: it will not change until it is explicitly changed using the SET OPTION statement.

When the SET TEMPORARY OPTION statement is applied using an individual user ID, the new option value is in effect as long as that user is logged in to the database.

When the SET TEMPORARY OPTION is used with the **PUBLIC** user ID, the change is in place for as long as the database is running. When the database is shut down, TEMPORARY options for the **PUBLIC** user ID revert back to their permanent value.

Setting an option for the **PUBLIC** user ID temporarily as opposed to permanently, offers a security advantage. For example, when the LOGIN_MODE option is enabled, the database relies on the log in security of the system on which it is running. Enabling it temporarily means that a database relying on the security of a Windows NT domain will not be compromised if the database is shut down and copied to a local machine. In that case, the temporary enabling of the LOGIN_MODE option will revert to its permanent value, which could be Standard, a mode where integrated logins are not permitted.

Deleting options

If *option-value* is omitted, the specified option setting will be deleted from the database. If it was a personal option setting, the value will revert back to the PUBLIC setting. If a TEMPORARY option is deleted, the option setting will revert back to the permanent setting.

> **Caution**
> *Changing option settings while fetching rows from a cursor is not supported, as it can lead to ill-defined behavior. For example, changing the DATE_FORMAT setting while fetching from a cursor would lead to different date formats among the rows in the result set. Do not change option settings while fetching rows.*

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise. Adaptive Server Anywhere does support some Adaptive Server Enterprise options using the SET statement.

Examples

- Set the date format option.

```
SET OPTION public.date_format = 'Mmm dd yyyy' ;
```

- Set the 'wait for commit' option to on.

```
SET OPTION waitefor_commit = 'on' ;
```

Embedded SQL Examples

```
1. EXEC SQL SET OPTION :user.:option_name = :value;
```

```
2. EXEC SQL SET TEMPORARY OPTION Date_format =
'mm/dd/yyyy';
```

SET OPTION statement [ISQL]

Function

To change Interactive SQL options.

Syntax 1

SET [**TEMPORARY**] **OPTION**
... [*userid*. | **PUBLIC**.]*option-name* = [*option-value*]

Syntax 2

SET PERMANENT

Syntax 3

SET

Parameters

userid:	*identifier*, *string* or *host-variable*
option-name:	*identifier*, *string* or *host-variable*
option-value:	*host-variable* (indicator allowed), *string*, *identifier*, or *number*

See Also

"Interactive SQL options" on page 138

Description

SET PERMANENT (syntax 2) stores all current Interactive SQL options in the SYSOPTIONS system table. These settings are automatically established every time Interactive SQL is started for the current user ID.

Syntax 3 displays all of the current option settings. If there are temporary options set for Interactive SQL or the database server, these will be displayed; otherwise, the permanent option settings are displayed.

SET SQLCA statement [ESQL]

Function

To tell the SQL preprocessor to use a SQLCA other than the default global *sqlca*.

Syntax

SET SQLCA *sqlca*

Parameters

sqlca: *identifier* or *string*

Permissions

None.

Side effects

None.

See also

"SQLCA management for multi-threaded or reentrant code" on page 29 of the book *Adaptive Server Anywhere Programming Interfaces Guide*

Description

The SET SQLCA statement tells the SQL preprocessor to use a SQLCA other than the default global *sqlca*. The **sqlca** must be an identifier or string that is a C language reference to a SQLCA pointer.

The current SQLCA pointer is implicitly passed to the database interface library on every Embedded SQL statement. All Embedded SQL statements that follow this statement in the C source file will use the new SQLCA.

This statement is necessary only when you are writing code that is reentrant (see "SQLCA management for multi-threaded or reentrant code" on page 29 of the book *Adaptive Server Anywhere Programming Interfaces Guide*). The **sqlca** should reference a local variable. Any global or module static variable is subject to being modified by another thread.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Open Client/Open Server.

Example

- The owing function could be found in a Windows DLL. Each application that uses the DLL has its own SQLCA.

```
an_sql_code FAR PASCAL ExecuteSQL( an_application
*app, char *com )
{
   EXEC SQL BEGIN DECLARE SECTION;
   char *sqlcommand;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL SET SQLCA "&app->.sqlca";
   sqlcommand = com;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL EXECUTE IMMEDIATE :sqlcommand;
return( SQLCODE );
}
```

SETUSER statement

Function

To allow a database administrator to impersonate another user, and to enable connection pooling.

Syntax

{ **SET SESSION AUTHORIZATION** | **SETUSER** }
    [[**WITH OPTIONS**] *userid*]

Parameters

userid: The user ID to be impersonated

Permissions

Must have DBA authority.

See also

"EXECUTE IMMEDIATE statement" on page 464
"GRANT statement" on page 484
"REVOKE statement" on page 536
"SET OPTION statement" on page 553

Description

The SETUSER statement is provided to make database administration easier. It enables a user with DBA authority to impersonate another user of the database.

Also, you can use SETUSER from an application server to take advantage of connection pooling. This cuts down the number of distinct connections that need to be made, and so helps performance.

If WITH OPTIONS is specified, the database options in effect are changed to the current database options of *userid*. By default, only permissions (including group membership) are altered.

SETUSER with no user ID undoes all earlier SETUSER statements. It returns the active user ID to that of the original connection.

There are several uses for the SETUSER statement, including the following:

- **Creating objects** You can use SETUSER to create a database object that is to be owned by another user.
- **Permissions checking** By acting as another user, with their permissions and group memberships, a DBA can test the permissions and name resolution of queries, procedures, views, and so on.
- **Providing a safer environment for administrators** The DBA has permission to carry out any action in the database. If you wish to ensure that you do not accidentally carry out an unintended action, you can use SETUSER to switch to a different user ID with fewer permissions.

Standards and compatibility

- **SQL/92** SET SESSION AUTHORIZATION is SQL 92 compliant. SETUSER is a vendor extension.
- **Sybase** Adaptive Server Enterprise supports SETUSER, but not the WITH OPTIONS keywords.

Examples

♦ The following statements, executed by a user named DBA, change the user ID to be Joe, then Jane, and then DBA

```
SETUSER 'Joe'
// ... operations...
SETUSER 'Jane'
// ... operations...
SETUSER
```

SIGNAL statement

Function

Signal an exception condition.

Syntax

SIGNAL *exception-name*

Permissions

None.

Side effects

None.

See also

"RESIGNAL statement" on page 531
"BEGIN... END statement" on page 361
"Using exception handlers in procedures and triggers" on page 261 of the book *Adaptive Server Anywhere User's Guide*

Description

SIGNAL allows you to raise an exception. See "Using exception handlers in procedures and triggers" on page 261 of the book *Adaptive Server Anywhere User's Guide* for a description of how exceptions are handled.

Standards and compatibility

- ♦ **SQL/92** Persistent Stored Module feature.
- ♦ **Sybase** SIGNAL is not supported by Adaptive Server Enterprise.

START DATABASE statement [ISQL]

Function

To start a database on the specified database server

Syntax

START DATABASE *database-file*
  ... [**AS** *database-name*]
  ... [**ON** *engine-name*]
  ... [**AUTOSTOP** { **YES** | **NO** }]

Permissions

The required permissions are specified by the database server `-gd` command-line option. This option defaults to **none** on the personal database server, and **dba** on the network server.

Side effects

None

See also

"STOP DATABASE statement" on page 564
"CONNECT statement" on page 381

Description

The START DATABASE statement starts a specified database on a specified database server. The database server must be running. The full path must be specified for the database-file, unless the file is located in the same directory.

The START DATABASE statement does not connect Interactive SQL to the specified database: a CONNECT statement needs to be issued in order to make a connection.

If *database-name* is not specified, a default name is assigned to the database. This default name is the root of the database file. For example, a database in file *c:\asa6\asademo.db* would be given the default name of **asademo**.

If *engine-name* is not specified, the default database is the first started server among those currently running.

The default setting for the AUTOSTOP clause is YES. With AUTOSTOP set to YES, the database is unloaded when the last connection to it is dropped. If AUTOSTOP is set to NO, the database is not unloaded.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Example

- Start the database file *c:\asa6\sample_2.db* on the current server.

```
START DATABASE 'c:\asa6\sample_2.db'
```

- Start the database file *c:\asa6\sample_2.db* as sam2 on the server named sample.

```
START DATABASE 'c:\asa6\sample_2.db'
AS sam2
ON sample ;
```

START ENGINE statement [ISQL]

Function

To start a database server.

Syntax

START ENGINE AS *engine-name* [**STARTLINE** *command-string*]

Permissions

The required permissions are specified by the database server `-gk` command-line option. This option defaults to **none** on the personal database server, and **dba** on the network server.

Side effects

None

See also

"STOP ENGINE statement" on page 565
"The database server" on page 12

Description

The START ENGINE statement starts a database server. If you wish to specify a set of options for the server, use the STARTLINE keyword together with a command string. Valid command strings are those that conform to the database server command-line description in "The database server" on page 12.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Example

- Start a database server, named sample, without starting any databases on it.

  ```
  START ENGINE AS sample ;
  ```

- Start a Windows 3.x personal server with a maximum cache size of 4 megabytes, loading the sample database.

  ```
  START ENGINE AS sample
  STARTLINE 'dbeng6w -c 4096 path\asademo.db'
  ```

- The following example shows the use of a STARTLINE clause.

  ```
  START ENGINE AS eng1 STARTLINE 'dbeng6 -c 8096'
  ```

START JAVA statement

Function

To start the Java VM.

Syntax

START JAVA

Permissions

DBA authority

Side effects

None

See also

"STOP JAVA statement" on page 566

Description

The START JAVA statement starts the Java VM. The main use is to load the VM at a convenient time so that when the user starts to use Java functionality there is no initial pause while the VM is loaded.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Example

- Start the Java VM.

```
START JAVA
```

STOP DATABASE statement [ISQL]

Function

To stop a database on the specified database server.

Syntax

STOP DATABASE *database-name*
   ... [**ON** *engine-name*]
   ... [**UNCONDITIONALLY**]

Permissions

The required permissions are specified by the database server `-gd` command-line option. This option defaults to **none** on the personal database server, and **dba** on the network server.

Side effects

None

See also

"START DATABASE statement" on page 561
"DISCONNECT statement" on page 450

Description

The STOP DATABASE statement stops a specified database on a specified database server. If *engine-name* is not specified, all running engines will be searched for a database of the specified name.

If the UNCONDITIONALLY keyword is supplied, the database will be stopped even if there are connections to the database. By default, the database will not be stopped if there are connections to it.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Examples

- Stop the database named *sample* on the default server.

```
STOP DATABASE sample ;
```

STOP ENGINE statement [ISQL]

Function

To stop a database server.

Syntax

STOP ENGINE *engine-name* [**UNCONDITIONALLY**]

Permissions

None

Side effects

None

See also

"START ENGINE statement" on page 562

Description

The STOP ENGINE statement stops the specified database server. If the UNCONDITIONALLY keyword is supplied, the database server is stopped even if there are connections to the server. By default, the database server will not be stopped if there are connections to it.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Example

- Stop the database server named sample.

```
STOP ENGINE sample
```

STOP JAVA statement

Function To stop the Java VM.

Syntax **STOP JAVA**

Permissions DBA authority

Side effects None

See also "START JAVA statement" on page 563

Description The STOP JAVA statement unloads the Java VM when it is not in use. The main use is to economize on the use of system resources.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Example

- Stop the Java VM.

```
STOP JAVA
```

SYSTEM statement [ISQL]

Function

To execute an operating system command from within Interactive SQL.

Syntax

SYSTEM [*operating-system-command*]

Permissions

None.

Side effects

None.

See also

"COMMIT statement" on page 377
"CONNECT statement" on page 381

Description

Executes the specified operating system command. If no command is specified, the DOS command interpreter or UNIX shell is started. You can return to Interactive SQL by using the system **exit** command or by pressing CTRL+D in UNIX.

- The SYSTEM statement must be entirely contained on one line.
- Comments are not allowed at the end of a SYSTEM statement.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not applicable.

Example

- The following statement executes the system command **date**.

```
SYSTEM date
```

TRUNCATE TABLE statement

Function

To delete all rows from a table, without deleting the table definition.

Syntax

TRUNCATE TABLE [*owner*.]*table-name*

Permissions

Must be the table owner, or have DBA authority, or have ALTER permissions on the table.

For base tables, the TRUNCATE TABLE statement requires exclusive access to the table, as the operation is atomic (either all rows are deleted, or none are).

For temporary tables, each user has their own copy of the data, and exclusive access is not required.

Side effects

Delete triggers are not fired by the TRUNCATE TABLE statement.

See also

"DELETE statement" on page 442

Description

The TRUNCATE TABLE statement deletes all rows from a table. It is equivalent to a DELETE statement without a WHERE clause, except that no triggers are fired as a result of the TRUNCATE TABLE statement and each individual row deletion is not entered into the transaction log.

After a TRUNCATE TABLE statement, the table structure and all of the indexes continue to exist until you issue a DROP TABLE statement. The column definitions and constraints remain intact, and triggers and permissions remain in effect.

The TRUNCATE TABLE statement is entered into the transaction log as a single statement, like data definition statements. Each deleted row is not entered into the transaction log.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

- Delete all rows from the **department** table.

```
TRUNCATE TABLE department
```

UNION operation

Function

To combine the results of two or more select statements.

Syntax

select-without-order-by
  ... **UNION** [**ALL**] *select-without-order-by*
  ... [**UNION** [**ALL**] *select-without-order-by*] ...
  ... [**ORDER BY** *integer* [**ASC** | **DESC**], ...]

Permissions

Must have SELECT permission for each of the component SELECT statements.

Side effects

None.

See also

"SELECT statement" on page 542

Description

The results of several SELECT statements can be combined into a larger result using UNION. The component SELECT statements must each have the same number of items in the select list, and cannot contain an ORDER BY clause.

The results of UNION ALL are the combined results of the component SELECT statements. The results of UNION are the same as UNION ALL except that duplicate rows are eliminated. Eliminating duplicates requires extra processing, so UNION ALL should be used instead of UNION where possible.

If corresponding items in two select lists have different data types, Adaptive Server Anywhere will choose a data type for the corresponding column in the result and automatically convert the columns in each component SELECT statement appropriately.

If ORDER BY is used, only integers are allowed in the order by list. These integers specify the position of the columns to be sorted.

The column names displayed are the same column names which would be displayed for the first SELECT statement.

Standards and compatibility

- **SQL/92** Entry level.
- **Sybase** Supported by Adaptive Server Enterprise, which also supports a COMPUTE clause.

Examples

- List all distinct surnames of employees and customers.

```
SELECT emp_lname
FROM Employee
UNION
SELECT lname
FROM Customer
```

UNLOAD TABLE statement

Function

To export data from a database table into an external ASCII-format file.

Syntax

UNLOAD [**FROM**] **TABLE** [*owner.*]*table-name*
 ... **TO** '*filename-string*'
 ... [**FORMAT ASCII**]
 ... [**DELIMITED BY** *string*]
 ... [**QUOTES ON** | **OFF**] [**ESCAPES ON** | **OFF**]
 ... [**ORDER ON** | **OFF**]
 ... [**ESCAPE CHARACTER** *character*]

Permissions

Must have SELECT permission on the table.

Side effects

None.

See also

"LOAD TABLE statement" on page 504
"OUTPUT statement" on page 514

Description

The UNLOAD TABLE statement allows efficient mass exporting from a database table into an ASCII file. UNLOAD TABLE is more efficient than the Interactive SQL statement OUTPUT, and can be called from any client application.

UNLOAD TABLE places an exclusive lock on the whole table.

When unloading columns with binary data types, UNLOAD TABLE writes hexadecimal strings, of the form \xnnnn where n is a hexadecimal digit.

For descriptions of the FORMAT, DELIMITED BY, and ESCAPE CHARACTER options, see "LOAD TABLE statement" on page 504. The other options are as follows:

QUOTES option With QUOTES turned on (the default), single quotes are placed around all exported strings.

ESCAPES option With ESCAPES on (the default), backslash-character combinations are used to identify special characters where necessary on export.

ORDER option With ORDER on (the default), the data is exported ordered by primary key values. With ORDER off, the data is exported in the same order you see when selecting from the table without an ORDER BY clause.

Exporting is slower with ORDER on. However, reloading using the LOAD TABLE statement is quicker because of the simplicity of the indexing step.

Standards and compatibility

- **SQL/92** Vendor extension.

- **Sybase** UNLOAD TABLE is not supported by Adaptive Server Enterprise. Similar functionality is provided by the Adaptive Server Enterprise bulk copy utility (**bcp**).

UPDATE statement

Function

To modify existing rows in database tables.

Syntax 1

UPDATE *table-list*
  ... **SET** *column-name*[.*field-name*] = *expression*, ...
  ... [**FROM** *table-list*]
  ... [**WHERE** *search-condition*]
  ... [**ORDER BY** *expression* [**ASC** | **DESC**] ,...]

Syntax 2

UPDATE *table-list*
  ... **SET** *column-name* [.*field-name*] = *expression*, ...
  ... [**VERIFY** (*column-name*, ...) **VALUES** (*expression*, ...)]
  ... [**WHERE** *search-condition*]
  ... [**ORDER BY** *expression* [**ASC** | **DESC**] ,...]

Syntax 3

UPDATE *table*
  ...**PUBLICATION** *publication*
  ...{ **SUBSCRIBE BY** *expression*
    | **OLD SUBSCRIBE BY** *expression* **NEW SUBSCRIBE BY**
  expression
    }
  ...**WHERE** *search-condition*

Permissions

Must have UPDATE permission for the columns being modified.

Side effects

None.

See also

"DELETE statement" on page 442
"INSERT statement" on page 498
"FROM clause" on page 476

Description

The UPDATE statement modifies values in rows of one or more tables. Each named column is set to the value of the expression on the right hand side of the equal sign. There are no restrictions on the *expression*. Even *column-name* can be used in the expression—the old value will be used. For a description of the table list and how to do joins, see "Joins: Retrieving Data from Several Tables" on page 129 of the book *Adaptive Server Anywhere User's Guide*.

The *field-name* is for use with Java columns, to update the value of a public field in the column.

Syntax 2 and Syntax 3 are applicable only to SQL Remote.

If no WHERE clause is specified, every row is updated. If a WHERE clause is specified, only rows satisfying the search condition are updated.

Normally, the order that rows are updated doesn't matter. However, in conjunction with the NUMBER(*) function, an ordering can be useful to get increasing numbers added to the rows in some specified order. Also, if you wish to do something like add 1 to a set of sequential primary key, it is necessary to do this in descending order by primary key, so that you do not get duplicate primary keys during the operation.

Views can be updated unless the SELECT statement defining the contains a GROUP BY clause or aggregate function, or involves a UNION operation.

Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. Thus, a character data type column updated with a string **Value** is always held in the database with an upper-case V and the remainder of the letters lower case. SELECT statements return the string as **Value**. If the database is not case-sensitive, however, all comparisons make **Value** the same as **value**, **VALUE**, and so on. Further, if a single-column primary key already contains an entry **Value**, an INSERT of **value** is rejected, as it would make the primary key not unique.

Updates based on joins

The optional FROM clause allows tables to be updated based on joins. If the FROM clause is present, the WHERE clause qualifies the rows of the FROM clause. Data is updated only in the table list immediately following the UPDATE keyword.

If a FROM clause is used, it is important to qualify the table name that is being updated the same way in both parts of the statement. If a correlation name is used in one place, the same correlation name must be used in the other. Otherwise, an error is generated.

For a full description of joins, see "Joins: Retrieving Data from Several Tables" on page 129 of the book *Adaptive Server Anywhere User's Guide*.

SQL Remote updates

Syntax 2 is intended for use with SQL Remote only, in single-row updates executed by the Message Agent. The VERIFY clause contains a set of values that are expected to be present in the row being updated. If the values do not match, any RESOLVE UPDATE triggers are fired before the UPDATE proceeds. The UPDATE does not fail simply because the VERIFY clause fails to match.

Syntax 3 is intended for use with SQL Remote only, inside a BEFORE trigger. The purpose is to provide a full list of subscribe by values any time the list changes. It is placed in SQL Remote triggers so that the database server can compute the current list of SUBSCRIBE BY values. Both lists are placed in the transaction log.

The Message Agent uses the two lists to make sure that the row moves to any remote database that did not have the row and now needs it. The Message Agent also removes the row from any remote database that has the row and no longer needs it. A remote database that has the row and still needs it is not be affected by the UPDATE statement.

Syntax 3 of the UPDATE statement allows the old SUBSCRIBE BY list and the new SUBSCRIBE BY list to be explicitly specified, which can make SQL Remote triggers more efficient. In the absence of these lists, the database server computes the old SUBSCRIBE BY list from the publication definition. Since the new SUBSCRIBE BY list is commonly only slightly different from the old SUBSCRIBE BY list, the work to compute the old list may be done twice. By specifying both the old and new lists, you can avoid this extra work.

The SUBSCRIBE BY expression is either a value or a subquery.

Using UPDATE to maintain subscriptions

Syntax 3 of the UPDATE statement is used to implement a specific SQL Remote feature, and is to be used inside a BEFORE trigger.

For publications created using a subquery in a SUBSCRIBE BY clause, you must write a trigger containing syntax 3 of the UPDATE statement in order to ensure that the rows are kept in their proper subscriptions.

Syntax 3 of the UPDATE statement makes an entry in the transaction log, but does not change the database table.

Standards and compatibility

- **SQL/92** Syntax 1 is an entry-level feature. Syntax 2 and 3 are vendor extensions for use only with SQL Remote.
- **Sybase** Subject to the expressions being compatible, the syntax of the UPDATE statement (syntax 1) is compatible between Adaptive Server Enterprise and Adaptive Server Anywhere. Syntax 2 and 3 are not supported.

Examples

- Transfer employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 129 ;
```

- Sales orders currently start at ID 2001. Renumber all existing sales orders by subtracting 2000 from the ID.

```
UPDATE sales_order_items AS items ,
   sales_order AS orders
SET items.id = items.id - 2000,
   orders.id = orders.id - 2000 ;
```

UPDATE (positioned) statement

Function

To modify the data at the current location of a cursor.

Syntax 1

UPDATE WHERE CURRENT OF *cursor-name*
... { **USING DESCRIPTOR** *sqlda-name* | **FROM** *host-variable-list* }

Syntax 2

UPDATE *table-list*
... **SET** *column-name* = *expression*, ...
... **WHERE CURRENT OF** *cursor-name*

Parameters

host-variable-list:	*indicator variables allowed*
sqlda-name:	*identifier*

Permissions

Must have UPDATE permission on the columns being modified.

Side effects

None.

See also

"DELETE statement" on page 442
"UPDATE statement" on page 572

Description

This form of the UPDATE statement updates the current row of the specified cursor. The current row is defined to be the last row FETCHed from the cursor and the last operation on the cursor must not have been a DELETE (positioned).

For syntax 1, columns from the SQLDA or values from the host variable list correspond one-to-one with the select list items of the specified cursor. If the **sqldata** pointer in the SQLDA is the null pointer, the corresponding select list item is not updated.

In syntax 2, the requested columns are set to the specified values for the row at the current row of the specified query. The columns do not need to be in the select list of the specified open cursor. This format can be prepared.

The USING DESCRIPTOR, FROM *host-variable-list*, and *host-variable* formats are for Embedded SQL only.

Updating views

Updates are not allowed on cursors on views that have more than one table in the FROM clause.

Standards and compatibility

- **SQL/92** Entry level feature.
- **Sybase** Embedded SQL use is supported by Open Client/Open Server, and procedure and trigger use is supported in Adaptive Server Anywhere.

Example

- The following is an example of an UPDATE statement WHERE CURRENT OF cursor:

```
UPDATE Employee
```

```
SET emp_lname = 'Jones'
WHERE CURRENT OF emp_cursor ;
```

VALIDATE TABLE statement

Function

To validate a table in the database.

Syntax

VALIDATE TABLE [*owner*.]*table-name*

Permissions

Must be the owner of the table, have DBA authority, or have REMOTE DBA authority (SQL Remote).

Side effects

None.

See also

"The Validation utility" on page 119

Description

The VALIDATE TABLE statement will scan every row of a table, and look up each row in each index on the table. If the database file is corrupt, an error will be reported. This should not happen. However, because DOS and Windows are unprotected operating environments, other software can corrupt memory used by the database server. This problem may be detected through software errors or crashes, or the corrupt memory could get written to the database, creating a corrupt database file. Also, in any operating system, hardware problems with the disk could cause the database file to get corrupted.

If you do have errors reported, you can drop all of the indexes and keys on a table and recreate them. Any foreign keys to the table will also need to be recreated. Another solution to errors reported by VALIDATE TABLE is to unload and reload your entire database. You should use the `-u` option of DBUNLOAD so that it will not try to use a possibly corrupt index to order the data.

Standards and compatibility

- **SQL/92** Vendor extension
- **Sybase** VALIDATE TABLE is not supported in Adaptive Server Enterprise. The procedure **dbcc checktable** provides a similar function.

WHENEVER statement [ESQL]

Function

To specify error handling in an Embedded SQL program.

Syntax

WHENEVER { SQLERROR | SQLWARNING | NOTFOUND }
... **GOTO** *label* | **STOP** | **CONTINUE** | { *C code;* }

Parameters

label: *identifier*

Permissions

None.

Side effects

None.

Description

The WHENEVER statement is used to trap errors, warnings and exceptional conditions encountered by the database when processing SQL statements. The statement can be put anywhere in an Embedded SQL C program and does not generate any code. The preprocessor will generate code following each successive SQL statement. The error action remains in effect for all Embedded SQL statements from the source line of the WHENEVER statement until the next WHENEVER statement with the same error condition, or the end of the source file.

Errors based on source position
The error conditions are in effect based on positioning in the C language source file, not based on when the statements are executed.

The default action is CONTINUE.

Note that this statement is provided for convenience in simple programs. Most of the time, checking the sqlcode field of the SQLCA (SQLCODE) directly is the easiest way to check error conditions. In this case, the WHENEVER statement would not be used. If fact, all the WHENEVER statement does is cause the preprocessor to generate an *if (SQLCODE)* test after each statement.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** Supported by Open Client/Open Server.

Examples

- The following are examples of the WHENEVER statement:

```
EXEC SQL WHENEVER NOTFOUND GOTO done;
```

```
EXEC SQL WHENEVER SQLERROR
   {
      PrintError( &sqlca );
      return( FALSE );
   };
```

WHILE statement [T-SQL]

Function

To provide repeated execution of a statement or compound statement.

Syntax

WHILE *expression*
    ... *statement*

Authorization

None.

Side effects

None.

See also

"LOOP statement" on page 508

Description

The WHILE conditional affects the performance of only a single SQL statement, unless statements are grouped into a compound statement between the keywords BEGIN and END.

The BREAK statement and CONTINUE statement can be used to control execution of the statements in the compound statement. The BREAK statement terminates the loop, and execution resumes after the END keyword marking the end of the loop. The CONTINUE statement causes the WHILE loop to restart, skipping any statements after the CONTINUE.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

- The following illustrates the use of WHILE:

```
WHILE (SELECT AVG(unit_price) FROM product) < $30
BEGIN
   UPDATE product
   SET unit_price = unit_price + 2
   IF ( SELECT MAX(unit_price) FROM product ) > $50
      BREAK
END
```

The BREAK statement breaks the WHILE loop if the most expensive product has a price above $50. Otherwise, the loop continues until the average price is greater than or equal to $30.

WRITETEXT statement [T-SQL]

Function

Permits non-logged, interactive updating of an existing text or image column.

Syntax

WRITETEXT *table-name.column-name*
    ... *text_pointer* [**WITH LOG**] *data*

Authorization

None.

Side effects

WRITETEXT does not fire triggers, and by default WRITETEXT operations are not recorded in the transaction log.

Description

Updates an existing text or image value. The update is not recorded in the transaction log, unless the WITH LOG option is supplied. You cannot carry out WRITETEXT operations on views.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

- The following code fragment illustrates the use of the WRITETEXT statement. The SELECT statement in this example returns a single row. The example replaces the contents of the **column_name** column on the specified row with the value **newdata**.

```
EXEC SQL create variable textpointer binary(16);
EXEC SQL set textpointer =
   (  select textptr(column_name)
      from table_name where ... ) ;
EXEC SQL writetext table_name.column_name
   textpointer 'newdata' ;
```