About This Manual

This manual explains how the Pervasive.SQL relational data access system implements Structured Query Language (SQL).

Who Should Read This Manual

This manual is intended for software developers using Pervasive.SQL statements to develop database applications. It assumes that you understand the basic concepts of SQL and relational database design.

Pervasive Software would appreciate your comments and suggestions about this manual. As a user of our documentation, you are in a unique position to provide ideas that can have a direct impact on future releases of this and other manuals. Please complete the User Comments form that appears on our Web site and fill in part number 100-003039-006.

Manual Organization

- Chapter 1—“SQL Overview”

  This chapter describes the types of SQL statements you can create using Pervasive.SQL.

- Chapter 2—“Scalable SQL Syntax”

  This chapter describes each command that Pervasive.SQL supports and provides the syntax for constructing valid SQL statements using these commands.

- Appendix A—“Data Types”

  This appendix provides detailed information about the data types that Pervasive.SQL supports.

- Appendix B—“Scalable SQL Keywords”

  This apppendix lists Pervasive.SQL keywords.

- Appendix C—“System Tables”

  This appendix describes the system tables that comprise the Pervasive.SQL data dictionary.

- Appendix D—“SQLSTATE Classes and Values”

  This appendix describes the different classes and values for the SQLSTATE session variable.

This manual also contains an index.

Conventions

Unless otherwise noted, command syntax, code, and code examples use the following conventions:

Case	Commands and reserved words typically appear in uppercase letters. Unless the manual states otherwise, you can enter these items using uppercase, lowercase, or both. For example, you can type MYPROG, myprog, or MYprog.
[ ]	Square brackets enclose optional information, as in [*log_name*]. If information is not enclosed in square brackets, it is required.
|	A vertical bar indicates a choice of information to enter, as in [*file name* | *@file name*].
< >	Angle brackets enclose multiple choices for a required item, as in /D=<5|6|7>.
variable	Words appearing in italics are variables that you must replace with appropriate values, as in *file name*.
...	An ellipsis following information indicates you can repeat the information more than one time, as in [*parameter* ...].
::=	The symbol ::= means one item is defined in terms of another. For example, a::=b means the item *a* is defined in terms of *b*.

SQL Overview

Structured Query Language (SQL) is a database language consisting of English-like statements you can use to perform database operations. Both the American National Standards Institute (ANSI) and IBM have defined standards for SQL. (The IBM standard is the Systems Application Architecture [SAA].) The Pervasive.SQL product implements most of the features of both ANSI SQL and IBM SAA SQL and provides additional extensions that neither standard specifies.

Pervasive.SQL allows you to create different types of SQL statements. The following table lists the types of SQL statements you can create and the tasks you can accomplish using each type of statement:

SQL Statement Type	Tasks
Data Definition	Create and delete dictionaries. Create, modify, and delete tables. Define column attributes. Create and delete indexes. Create and drop triggers.
Data Manipulation	Retrieve, insert, update, and delete data in tables. Define transactions. Define and delete views. Create, delete, and execute stored SQL procedures.
Data Control	Enable and disable security for a dictionary. Create users and groups. Grant and revoke table access rights.
Data Administration	Specify Pervasive.SQL session values that define isolation levels, file open modes, and file owner names.

The rest of this chapter briefly describes the SQL statements used in each statement category. For detailed information about each statement, refer to Chapter 2, “Scalable SQL Syntax.”

The following are the statement category overview sections found in this chapter:

- “Data Administration Statements”
- “Data File Paths”

Data Definition Statements

Data definition statements let you specify the characteristics of your database. When you execute data definition statements, Pervasive.SQL stores the description of your database in a data dictionary. You must define your database in the dictionary before you can store or retrieve information.

Pervasive.SQL allows you to construct data definition statements to do the following:

- Create and delete dictionaries.
- Create, modify, and delete tables.
- Define column attributes.
- Create and delete indexes.
- Create and delete triggers.

The following sections briefly describe the SQL statements associated with each of these tasks. For general information about defining the characteristics of your database, refer to the *Pervasive.SQL Programmer's Guide*.

Creating and Deleting Dictionaries

You can create and delete dictionaries by constructing statements using the following statements:

CREATE DICTIONARY	**Creates a data dictionary.**
DROP DICTIONARY	**Deletes a data dictionary and the associated data files.**

Creating, Modifying, and Deleting Tables

You can create, modify, and delete tables from a database by constructing statements using the following statements:

CREATE TABLE	**Defines a table and creates the corresponding data file.**
ALTER TABLE	**Makes a single change to a table definition. With an ALTER TABLE statement, you can change the path or file name of the corresponding data file, add a column to the table definition, remove a column from the table definition, change a column's data type or length, and add or remove a primary key or a foreign key.**
DROP TABLE	**Deletes a table from the data dictionary and deletes the associated data file from the disk.**

Defining Column Attributes

You can define certain column attributes by constructing statements using the following statements:

SET MASK	**Specifies the display format of a column.**
SET RANGE	**Specifies one or more acceptable ranges of values for a column.**
SET VALUES	**Specifies all acceptable values for a column.**
SET CHAR	**Specifies a list of valid input characters for a string column.**
SET DEFAULT	**Specifies the value to which Pervasive.SQL defaults if you do not specify a column value.**
SET (global null value)	**Specifies the value (such as an ASCII character) to use as the null value in all columns of a certain data type.**

Creating and Deleting Indexes

You can create and delete indexes from a database by constructing statements using the following statements:

CREATE INDEX	**Defines a new index (a named index) for an existing table.**
DROP INDEX	**Deletes a named index.**

Creating and Deleting Triggers

You can create and delete triggers from a database by constructing statements using the following statements:

CREATE TRIGGER	Defines a trigger for an existing table.
DROP TRIGGER	Deletes a trigger.

Data Manipulation Statements

Data manipulation statements let you access and modify the contents of your database. Pervasive.SQL allows you to construct data manipulation statements to do the following:

- Retrieve data from tables.
- Modify data in tables.
- Define transactions.
- Create and delete views.
- Create, delete, and execute stored procedures.

The following sections briefly describe the SQL statements associated with each of these tasks. For general information about accessing and modifying the contents of your database, see the *Pervasive.SQL Programmer's Guide*.

Retrieving Data

All statements you use to retrieve information from a database are based on the SELECT statement.

SELECT	Retrieves data from one or more tables in the database.

When you create a SELECT statement, you can use various clauses to specify different options. (See the entry for the SELECT statement in Chapter 2, "Scalable SQL Syntax" for detailed information about each type of clause.) The types of clauses you use in a SELECT statement are as follows:

FROM	Specifies the tables or views from which to retrieve data.
WHERE	Defines search criteria that qualify the data a SELECT statement retrieves.
GROUP BY	Combines sets of rows according to the criteria you specify and allows you to determine aggregate values for one or more columns in a group.
HAVING	Allows you to limit a view by specifying criteria that the aggregate values of a group must meet.
ORDER BY	Determines the order in which Pervasive.SQL returns selected rows.

In addition, you can use the UNION keyword to obtain a single result table from multiple SELECT queries.

Modifying Data

You can add, change, or delete data from tables and views by issuing statements such as the following:

INSERT	Adds rows to one or more tables or a view.
UPDATE	Changes data in a table or a view.
DELETE	Deletes rows from a table or a view.

When you create a DELETE or UPDATE statement, you can use a WHERE clause to define search criteria that restrict the data upon which the statement acts.

Defining Transactions

To update the data in a database, you can issue SQL statements individually or you can define *transactions* (logical units of related statements). By defining transactions, you can ensure that either all the statements in a unit of work are executed successfully or none are executed. You can use transactions to group statements to ensure the logical integrity of your database.

Pervasive.SQL provides the following statements to allow you to use transactions:

COMMIT WORK	Ends a transaction and makes the changes that occurred during that transaction permanent.
ROLLBACK WORK	Ends a transaction and reverses all the changes that the previous statements made in the transaction.
START TRANSACTION	Begins a transaction, except in implicit transaction processing, in which a transaction begins with the first statement following a COMMIT WORK or ROLLBACK WORK statement.
SAVEPOINT	Provides markers in a SQL transaction that allow you to undo a partial set of changes in a transaction and continue with additional changes before requesting the final commit or abort of the entire transaction. Working in conjunction with the ROLLBACK TO SAVEPOINT statement, savepoints provide a way to nest transactions.

Creating and Deleting Views

You can create and delete views by constructing statements using the following statements:

CREATE VIEW	**Defines a database view and stores the definition**

	in the dictionary.
DROP VIEW	**Deletes a view from the data dictionary.**

Creating, Deleting, and Executing Stored Procedures

A stored procedure consists of statements you can precompile and save in the dictionary. To create, delete, and execute stored procedures, construct statements using the following:

CREATE PROCEDURE	Stores a new procedure in the data dictionary.
DROP PROCEDURE	Deletes a stored procedure from the data dictionary.
CALL	Recalls a previously compiled procedure and executes it.

Pervasive.SQL provides additional SQL control statements, which you can only use in the body of a stored procedure or trigger. You can use the following statements in stored procedures and triggers:

BEGIN...END (compound statement)	Called a compound statement; allows you to group other statements together.
IF...THEN...ELSE	Provides conditional execution based on the truth value of a condition.
LEAVE	Continues execution by leaving a block or loop statement.
LOOP	Repeats the execution of a block of statements.
WHILE	Repeats the execution of a block of statements while a specified condition is true.

The following statements allow you to retrieve information about the last statement that completed execution and provide a means to handle exception conditions:

DECLARE CONDITION	Allows you to declare a warning or exception condition in a stored procedure by associating a condition name with a SQLSTATE value. SQLSTATE is a system variable that

	contains the status of the last completed statement.
DECLARE HANDLER	Allows you to assign a condition handler to a named condition.
SIGNAL	Allows you to signal an exception condition or a completion condition other than successful completion.
RESIGNAL	Allows you to resignal an exception condition or a completion condition other than successful completion when you don't know what the original condition was.

Data Control Statements

Data control statements let you define security for your database. When you create a dictionary, no security is defined for it until you explicitly enable security for that dictionary. Pervasive.SQL allows you to construct data control statements to do the following:

- Enable and disable security.
- Create users and groups.
- Grant and revoke rights.

The following sections briefly describe the SQL statements associated with each of these tasks. For general information about Pervasive.SQL security, see the *Pervasive.SQL Programmer's Guide*.

Enabling and Disabling Security

You can enable or disable security for a database by issuing statements using the following statement:

SET SECURITY	Enables or disables security for the database and sets the master password.

Creating and Deleting Users and Groups

You can create or delete users and user groups for the database by constructing statements using the following statements:

CREATE GROUP	**Creates a new group of users.**
DROP GROUP	**Deletes a group of users.**
GRANT LOGIN	**Creates users and passwords, or adds users to groups.**
REVOKE LOGIN	**Removes a user from the dictionary.**

Granting and Revoking Rights

You can assign or remove rights from users or groups by issuing statements using the following:

GRANT (access rights)	**Grants a specific type of rights to a user or a group. The rights you can grant with a GRANT (access rights) statement are All, Insert, Delete, Alter, Select, Update, and References.**

GRANT CREATETAB	**Grants the right to create tables to a user or a group.**
REVOKE (access rights)	**Revokes access rights from a user or a group.**
REVOKE CREATETAB	**Revokes the right to create tables from a user or a group.**

Data Administration Statements

Data administration statements let you specify settings for some special Pervasive.SQL session variables. You can construct data administration statements to specify the following:

- Isolation level.
- File open mode.
- File owner names.

You can set the special Pervasive.SQL session variables by issuing statements using the following:

SET ISOLATION	Restricts access to tables from other tasks or users.
SET OPENMODE	Specifies the file open mode for accessing your database. The open modes are Normal, Accelerated, Read-Only, Verify, and Exclusive.
SET OWNER	Specifies file owner names so that Pervasive.SQL can access data in files that have owner names.

Data administration statements do *not* set any operating system variables; they set variables only for a specific Pervasive.SQL session. The values you assign with these statements apply only during a single login session. Once you log out of the dictionary, Pervasive.SQL clears the settings you made and does not store them in the data dictionary.

For more information, see the discussions of the SET ISOLATION, SET OPENMODE, and SET OWNER statements in Chapter 2, “Scalable SQL Syntax.”

Data File Paths

When you create a table in Pervasive.SQL, you can specify a physical file name (DOS formatted) with or without directory locations. Pervasive.SQL stores the path information you specify at creation in the Xf$Loc column of the X$File system table.

> **Note:** When you use named databases, the paths in Xf$Loc must be simple file names or relative paths. For more information about these system tables, refer to Appendix C, "System Tables."

Pervasive.SQL combines the physical file name with the first directory you list in the data file path. You specify the data file path either when logging in with paths or when setting up a database name with the Setup utility. Refer to the *Pervasive.SQL User's Guide* for more information about this utility.

The resulting path tells Pervasive.SQL where to create the data file associated with the table. Although the current data file path may contain as many as eight entries for opening existing tables, Pervasive.SQL always uses the first path when creating a table.

Table 1-1 shows the maximum length you can specify for the data file path, Xf$Loc path, and the resulting full path.

Table 1-1
Maximum Data File Path Lengths

Path Type	Maximum Path Length
Data file path (single entry)	64
Xf$Loc path	64
Resulting full path	80

When you use a CREATE TABLE statement to create a table, the physical file name is optional. If you do not specify a file name in the USING clause, Pervasive.SQL generates a unique name, appends the extension.MKD to it, and creates the data file in the first directory specified in the data file path.

When applications subsequently attempt to access the table, Pervasive.SQL combines the session's current data file path with the Xf$Loc column to obtain the data file's full path. When multiple data file path entries exist, Pervasive.SQL attempts to open the file in each successive location until it succeeds or no file is located. If Scalable SQL cannot find the file, you receive a Status Code 12.

Database Names

A database name is a name you associate with the location of a dictionary and its data files. An application can log in to a database using either the database name or a path. Database names are stored in the database names configuration file (DBNAMES.CFG). If you add a primary key, foreign key, or trigger to a table, the database name is also written to the data file associated with the table. Bound named databases also force the database name to be written to the data file for every table in the database. (For more information about bound databases, refer to the

Pervasive.SQL Programmer's Guide.)

You can pass database names as strings when logging in or using the database names functions. Database names must follow these conventions:

- Begin with a letter.
- Cannot contain blanks.
- Cannot be a reserved keyword.
- Must not exceed 20 characters.
- Database names are not case-sensitive.
- When logging in to a database using a database name, you must precede the name with an @ character.

Path Strings

If you do not use a database name when logging in, you must specify a path to the dictionary, and possibly one or more paths to the data files. You may also specify a path for a data file inside a SQL statement when creating or altering a table.

You must specify an operating system path as a string. The string can be up to 64 characters long. If you specify a dictionary path or a single data file path that is shorter than 64 characters, you must terminate the string with a binary 0. If you specify multiple data file paths, you must separate each path with a semicolon and terminate the string with a binary 0.

Scalable SQL Syntax

This chapter discusses the statements that Scalable SQL supports and explains how to construct valid SQL statements using these statements. The SQL statements are listed alphabetically. Each statement description provides valid syntax, discusses the statement's purpose and use, and shows examples of valid SQL statements that use the statement.

Comments in SQL Statements

Scalable SQL allows you to embed comments in your statements. Comments begin with a delimiter (--) and end with a carriage return. You can place a comment on any line of your statement as long as the comment follows all the statement text on that line. The following example illustrates the use of comments; the <CR> symbol shows where carriage returns terminate the lines:

Figure 2-1
Embedded Comments in SQL Statements

```
--generate local mailing list<CR>

SELECT Last_Name, First_Name, Street, City, State, Zip<CR>

    FROM Person<CR>

    WHERE Zip >= ’78730’

    AND Zip <= '78797' --
limit the list to the immediate area<CR>
```

When Scalable SQL compiles the statement, it treats any text between the comment delimiter and the carriage return as a comment. Scalable SQL does not compile comments.

ALTER TABLE

ALTER TABLE table_name [IN DICTIONARY]

< USING 'path_name' [WITH REPLACE]

| ADD

  < column_definition

  | PRIMARY KEY primary_key_definition

  | FOREIGN KEY foreign_key_definition

  >

| MODIFY column_definition

| DROP

  < column_name

  | PRIMARY KEY

  | FOREIGN KEY foreign_key_name

  >

>

where

column_definition ::= *column_name*
data_specification

[CASE]

data_specification ::= data_type [(data_length)]

data_type ::=

< AUTOINC | BFLOAT | BIT | CHAR | CHARACTER

| CURRENCY | DATE | DEC | DECIMAL | FLOAT |
INT

| INTEGER | LOGICAL | LSTRING | LVAR | MONEY
| NOTE | NUMERIC | NUMERICSA | NUMERICSTS
| TIME | TIMESTAMP | UNSIGNED | ZSTRING >

data_length ::= length [, decimal]

primary_key_definition ::= (*column_name_list*)

column_name_list ::= column_name [, column_name]
...

foreign_key_definition ::=

[foreign_key_name] (column_name_list)

REFERENCES table_name

[ON DELETE < CASCADE | RESTRICT >]

[ON UPDATE RESTRICT]

The ALTER TABLE statement allows you to change a table definition as follows:

- Change the path or file name associated with the table (with a USING clause).
- Add a column, primary key, or foreign key to the table definition (with an ADD clause).
- Change a column's data type or length (with a MODIFY clause).
- Remove a column, primary key, or foreign key from the table definition (with a DROP clause).

You can specify only one of these operations with each ALTER TABLE statement. Also, if security is enabled, you must have the Alter right on a table in order to change it with an ALTER TABLE statement.

Scalable SQL does not retain the owner name or the owner name access type when you issue an ALTER TABLE statement that adds, modifies, or drops a column definition.

Under certain circumstances, an ALTER TABLE statement causes Scalable SQL to discard the page size specification of the table and calculate the optimal page size. If you specify a data file page size when you create the table (see "Specifying Data File Options"), issuing an ALTER TABLE statement that adds or drops a column or modifies a column definition causes Scalable SQL to adjust the page size.

Note: Scalable SQL commits the changes specified in an ALTER TABLE statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

You cannot alter a table definition during a transaction if you have previously referred to the table during the transaction. For example, if you start a transaction, insert a row into the Person table, and then try to alter the Person table definition, the ALTER TABLE statement fails. You must commit or roll back the work from the transaction first, and then alter the table definition.

IN DICTIONARY

To change only the dictionary definition of a table, include IN DICTIONARY in your ALTER TABLE statement. This can be useful when you want to put a definition in the dictionary to match an existing data file, or when you want to use a USING clause to change the data file pathname for a table.

Including IN DICTIONARY causes Scalable SQL to modify only the table's dictionary definition, not the actual data file. If you do not include IN DICTIONARY, Scalable SQL attempts to modify the existing data file to match the new table definition. On a large file, this process may be lengthy.

IN DICTIONARY is effective when you add, modify, or delete columns, but Scalable SQL ignores it when you add or delete a primary or foreign key.

Note: You cannot use the IN DICTIONARY clause on tables in a bound named

database. For more information about bound databases, refer to the *Database Design Guide*.

USING

Include a USING clause to specify the physical location and name of the data file to associate with the table. A USING clause also allows you to create a new data file at a particular location using an existing dictionary definition. (The string supplied in the USING clause is stored in the Xf$Loc column of the dictionary file X$File.)

In the sample database, the Person table is associated with the file PERSON.MKD. If you create a new file named PERSON2.MKD, the statement in the following example changes the dictionary definition of the Person table so that the table is associated with the new file.

```
ALTER TABLE Person IN DICTIONARY USING
'person2.mkd';
```

If you are altering a table definition that is part of a named database, you must use either a simple file name or a relative path in the USING clause. If you specify a relative path, Scalable SQL interprets it relative to the data file path associated with the database name.

WITH REPLACE

Include WITH REPLACE in a USING clause to instruct Scalable SQL to replace an existing file (the file must reside at the location you specified in the USING clause). If you include WITH REPLACE, Scalable SQL creates a new file, discarding any data stored in the original file with the same name. If you do not include WITH REPLACE and a file exists at the specified location, Scalable SQL returns a status code and does not create the new file.

ADD

Include an ADD clause to add a column, primary key, or foreign key to a table definition.

Adding a Column

To add a new column to a table, define the column in an ADD clause using the same format you use when defining columns with the CREATE TABLE statement (see “Defining Columns”).

The names you specify for column definitions are the column names that Scalable SQL stores in the dictionary. Column names must be unique within a table, but you can use the same name in more than one table.

You must always specify the column’s data type. However, the internal storage length is mandatory only for the LVAR and NOTE data types. If you do not specify the length for any other data type, Scalable SQL creates the column with the default length. For detailed information about each data type, refer to Appendix A, “Data Types.”

To specify *data_length* for the DECIMAL, NUMERIC, NUMERICSA, and NUMERICSTS data types, use the following length notation:

length,decimal

In this notation, *length* is the total internal length in bytes, and *decimal* is the number of displayable decimal places to the right of the decimal point. (The number of digits represented by length depends on the data type. Refer to Appendix A, “Data Types.” for information about the internal storage formats of these data types.) Because the decimal places are implied, they are also included in the overall length of the column. The value of *decimal* is optional, and Scalable SQL uses the default values that are shown in Appendix A, “Data Types.”

For the rest of the data types, you can only specify one value, the total internal length in bytes, as *data_length*.

The following statement adds the Emergency_Phone column to the Department table:

```
ALTER TABLE Department ADD Emergency_Phone NUMERIC
(10,0);
```

a. When you add a column to an existing table, Scalable SQL always places the column at the end of the column list; the new column becomes the last column of your table. This is especially important if the table you are altering contains a variablelength column. Because a variablelength column must be the last column in a table, if you attempt to add a column to a table that contains a variable-length column, rScalable SQLeturns Status Code 261. You must drop the variablelength column before adding new columns to the table.

Note: When you add a column, Scalable SQL modifies both the table definition and the data file associated with the table, unless you include IN DICTIONARY.

CASE

When you add a string column with an ADD clause, include the CASE keyword if you want Scalable SQL to ignore case when evaluating restriction clauses involving the column.

Adding a Primary Key

To add a primary key to a table definition, include PRIMARY KEY in the ADD clause and define the key using the same format as defining keys with the CREATE TABLE statement (see "Defining the Primary Key"). Before adding the primary key, you must ensure that the columns in the primary key column list are defined as a unique index that does not include null values. If such an index does not exist, create one with the CREATE INDEX statement.

The following statement defines a primary key on a table called Faculty. (The ID column is defined as a unique index that does not include null values.)

```
ALTER TABLE Faculty ADD PRIMARY KEY (ID);
```

Because a table can have only one primary key, you cannot add a primary key to a table that already has a primary key defined. To change the primary key of a table, delete the existing key using a DROP clause in an ALTER TABLE statement and add the new primary key.

Note: You must be logged in to the database using a database name before you can add a primary key or conduct any other referential integrity (RI) operation.

Adding a Foreign Key

To add a foreign key to a table definition, include FOREIGN KEY in the ADD clause and define the key using the same format as defining foreign keys with the CREATE TABLE statement (see "Defining Foreign Keys"). Before adding the foreign key, you must ensure that the columns in the foreign key column list are defined as an index that does not include null values. If such an index does not exist, create one with the CREATE INDEX statement.

The following statement adds a new foreign key to the Class table. (The Faculty column is defined as an index that does not include null values.)

```
ALTER TABLE Class ADD FOREIGN KEY Teacher
(Faculty_ID)

 REFERENCES Faculty ON DELETE RESTRICT;
```

In this example, the restrict rule for deletions prevents someone from removing a faculty member from the database without first either changing or deleting all of that faculty's classes.

If you add a foreign key to a table that already contains data, use the SQLScope or RI utility to find any data that does not conform to the new referential constraint. See the *Pervasive.SQL User's Guide* for information about these utilities.

> **Note:** You must be logged in to the database using a database name before you can add a foreign key or conduct any other RI operation. Also, when security is enabled, you must have the Reference right on the table to which the foreign key refers before you can add the key.

MODIFY

Include a MODIFY clause to change a column's data type or length. Use the same format you use when specifying the data type and length in a CREATE TABLE statement (see "Defining Columns"). If Scalable SQL detects an incompatibility between the old and new data types (for example, if you try to change a LOGICAL column to a FLOAT column), it returns Status Code 850 and does not change the database.

> **Note:** When you modify a column, Scalable SQL modifies both the table definition and the data file associated with the table unless you include IN DICTIONARY. (See "IN DICTIONARY" for more information about this clause.)

When you change a string column definition with a MODIFY clause, include the CASE keyword if you want Scalable SQL to ignore case when evaluating restriction clauses involving the column.

DROP

Include a DROP clause to delete a column, primary key, or foreign key from a table definition.

Dropping a Column

To drop a column from a table definition, specify the name of the column in a DROP clause. The following statement drops the emergency phone column from the Person table:

```
ALTER TABLE Person DROP Emergency_Phone;
```

> **Note:** When you drop a column, Scalable SQL modifies both the table definition and the data file associated with the table unless you include IN DICTIONARY.

(See “IN DICTIONARY” for more information about this clause.)

Dropping a Primary Key

To drop a table’s primary key, include PRIMARY KEY in a DROP clause. You must be logged in to the database using a database name before you can drop a primary key or conduct any other RI operation.

The following statement drops the primary key from a table called Faculty:

```
ALTER TABLE Faculty DROP PRIMARY KEY;
```

Before you can drop a primary key from a parent table, you must drop any corresponding foreign keys from dependent tables.

Dropping a Foreign Key

To drop a foreign key, include a DROP clause with FOREIGN KEY followed by the foreign key name. Scalable SQL drops the foreign key from the dependent table and eliminates the referential constraints between the dependent table and the parent table.

Note: You must be logged in to the database using a database name before you can drop a foreign key or conduct any other RI operation.

The following statement drops the foreign key Faculty from the Class table:

```
ALTER TABLE Class

DROP FOREIGN KEY Teacher;
```

BEGIN...END (compound statement)

[*beginning_label* :]

BEGIN [[NOT] ATOMIC]

[*local_declaration_list*]

[*handler_declaration_list*]

SQL_statement_list

END [*ending_label*]

where

local_declaration_list ::= { *local_declaration* ; }
...

local_declaration ::=

< *SQL_variable_declaration*

| *SQL_cursor_declaration*

| *condition_declaration*

>

handler_declaration_list ::= { *handler_declaration* ;
} ...

SQL_statement_list ::= { *SQL_statement* ; } ...

SQL_variable_declaration ::= see DECLARE (variable)

SQL_cursor_declaration ::= see DECLARE CURSOR

condition_declaration ::= see DECLARE CONDITION

A compound statement groups other statements together. You can only use a compound statement in the body of a stored procedure or in a trigger declaration.

If you specify a beginning label for a compound statement, it is called a labeled compound statement.

Rules for Creating Compound Statements

The following rules apply to creating compound statements.

- When a compound statement is the body of a stored procedure and you do not specify an explicit beginning label, the procedure name of that procedure is the beginning label by default. When a compound statement is the body of a stored procedure and you do not specify an ending label, the procedure name of that procedure is the ending label by default. When you use a compound statement in a trigger, you can still specify the beginning label; however, if you do not, then the beginning label is undefined.

- If you specify an ending label, you must specify an identical explicit or implicit beginning label. When you use a compound statement in a trigger and specify an ending label, then you must provide an identical beginning label.

- A specified or implicit (default) beginning label must be different from all other statement labels inside that compound statement.
- If a compound statement is the body of a stored procedure, no SQL variable name declared in that procedure can be identical to a parameter name in the parameter list of that procedure.
- No two declarations (variable, cursor, or condition) in a local declaration list can have the same declared name.
- If you do not specify ATOMIC or NOT ATOMIC, then NOT ATOMIC is the default.
- If you specify ATOMIC, then the SQL statement list must not contain either a COMMIT or a ROLLBACK statement.
- No regular SELECT statement is allowed in a compound statement.
- The SQL statements of the SQL statement list are executed in the order in which you specify them.
- Although you may nest BEGIN...END statements within other BEGIN...END statements, only the outermost BEGIN...END statement can contain DECLARE statements.
- If an exception condition occurs during the execution of a SQL statement of the SQL statement list, then the execution of the SQL statement list terminates. If the compound statement contains a handler declaration associated with the raised exception condition, the handler activates; otherwise, the compound statement terminates with the unhandled exception condition.
- If a completion condition other than successful completion occurs during the execution of a SQL statement and the compound statement contains a handler declaration associated with the raised completion condition, then the handler activates; otherwise, execution resumes with the next SQL statement of the SQL statement list.
- If you specify ATOMIC and either a compound statement terminates with an unhandled exception condition, or an activated handler resignals an exception condition for which no handler is defined (causing the compound statement to terminate with the unhandled condition), then all changes to SQL data resulting from execution of the compound statement are cancelled. All savepoints established during the execution of the compound statement are rolled back.
- For any SQL cursor that is declared in a stored procedure and is in an open state at the time of completion or termination of the procedure, a CLOSE statement is executed by default.

Example

The following BEGIN...END statement inserts values for the student ID, class ID, and grade into the Enrolls table.

```
BEGIN

SET studID = '450-52-0400';

SET studclassID = 43;

SET studGrade = 0.0000;


INSERT INTO Enrolls

 VALUES ('450-52-0400', 43, 0.0000);


END
```

For more examples using the BEGIN...END statement, refer to the CREATE PROCEDURE examples on page 2-37 and the CREATE TRIGGER example on page 2-57.

CALL

CALL *procedure_name* (

[< *positional_argument*_list

  [, keyword_argument_list]

| keyword_argument_list

 >

])

where

positional_argument_list ::=

argument [, *argument*] ...

keyword_argument_list ::=

keyword_argument [, *keyword_argument*] ...

keyword_argument ::= : *parameter_name* => *argument*

argument ::= *simple_value*

simple_value ::=

< *literal*

| *SQL_variable_name*

| *parameter_name*

| *column_reference*

>

column_reference ::= [*column_qualifier.*]
column_name

column_qualifier ::= < *table_name* | *view_name* |
alias_name >

Use the CALL statement to invoke a stored procedure.

Rules for Using the CALL Statement

The following rules apply to using the CALL statement:

- You must define a value for every parameter in the procedure declaration. You can assign a value to a parameter using the associated argument in the CALL statement or with the associated default clause in the CREATE PROCEDURE statement. An argument value for a parameter in a CALL statement overrides any associated default value.

- You cannot assign a parameter value twice in the argument list. If you use both positional arguments and

keyword arguments, the keyword arguments must not refer to a parameter that receives its value through the positional arguments. Also, when using keyword arguments, the same parameter name must not occur twice.

- If you submit a CALL through an XQLCompile and provide a SQL variable as an argument, then you must declare the variable as a session variable. If you issue a CALL in a procedure and provide a SQL variable as an argument, then the variable must be either a session variable or a variable that the calling procedure owns.

- If you provide a parameter name as an argument, you must declare it as a parameter of the same procedure in which you issue the CALL statement. A CALL submitted through an XQLCompile must not contain a parameter name.

- If you specify a constant or a column reference as an argument for a parameter that has been declared with parameter mode OUT or INOUT, you receive Status Code 904 as a result of the call.

- If you specify a constant or a column reference as an argument for a parameter that has been declared with parameter mode IN or with no parameter mode, you receive an error status if the parameter is needed as a target (for example, in a SET statement) during the execution of the procedure. The effects of any statements executed in the procedure up to this point are rolled back only if you declare the compound statement of the procedure to be ATOMIC or if a declared condition handler handles the roll back.

Note: You should declare the parameter mode for each parameter in order to avoid this kind of error.

- An argument passed into a parameter has to be a compatible data type. Refer to Table 2-5 for a list of data type conversions.

Examples

The following example calls a procedure without parameters:

```
CALL NoParms()
```

The following examples call a procedure with parameters:

```
CALL Parms(vParm1, vParm2);
```

```
CALL CheckMax (N.Class_ID);
```

CLOSE (cursor)

CLOSE *cursor_name*

A CLOSE CURSOR statement closes a SQL cursor.

The cursor that the cursor name specifies must be open.

Example

The following example closes the cursor BTUCursor.

```
CLOSE BTUCursor;
```

COMMIT WORK

COMMIT [WORK][AND [NO] CHAIN]

The COMMIT WORK statement signals the end of a logical transaction. When you issue this statement, the keyword WORK is optional. You can issue a COMMIT WORK statement only after issuing a START TRANSACTION statement.

Issuing a COMMIT WORK statement commits to the tables all the operations you have performed since the preceding START TRANSACTION statement; you can no longer use a ROLLBACK WORK statement to undo the operations performed within that transaction. All changes made during the committed transaction become visible to other clients.

Note: If you start a transaction and then log out of the dictionary before issuing a COMMIT WORK or ROLLBACK WORK statement, Scalable SQL automatically issues a ROLLBACK WORK statement before completing the logout.

Rules for Using the COMMIT WORK Statement

The following rules apply to using the COMMIT WORK statement:

- The keyword WORK is optional.
- AND NO CHAIN is the default clause in a COMMIT WORK statement.
- A SQL transaction must be currently active before you can issue this statement.
- All savepoints that the current SQL transaction defines are destroyed.
- All cursors opened during the current transaction are closed, and the current SQL transaction is terminated.
- The current SQL transaction is terminated after you issue the COMMIT WORK statement. If you specify AND CHAIN, a new transaction begins.

Example

The following statement begins a transaction which updates the Amount_Owed column in the Billing table. This work is committed; the AND CHAIN clause begins another transaction that updates the Amount_Paid column and sets it to zero. The final COMMIT WORK statement ends the second transaction.

```
START TRANSACTION;

UPDATE Billing B

SET Amount_Owed = Amount_Owed - Amount_Paid

WHERE Student_ID IN

 (SELECT DISTINCT E.Student_ID

 FROM Enrolls E, Billing B

 WHERE E.Student_ID = B.Student_ID);
```

```
COMMIT WORK AND CHAIN;

UPDATE Billing B

SET Amount_Paid = 0

WHERE Student_ID IN

 (SELECT DISTINCT E.Student_ID

 FROM Enrolls E, Billing B

 WHERE E.Student_ID = B.Student_ID);

COMMIT WORK;
```

CREATE DICTIONARY

CREATE DICTIONARY USING 'path_name' [WITH REPLACE]

The CREATE DICTIONARY statement allows you to create a dictionary or replace an existing one. When you create a dictionary, you must specify a valid directory path for the dictionary in a USING clause. In the directory you specify, Scalable SQL creates three files in which to store the dictionary information: FILE.DDF, FIELD.DDF, and INDEX.DDF.

Scalable SQL creates more dictionary files when you enable security and define column attributes, views, stored procedures, triggers, or referential integrity constraints. For more information about the dictionary files and their related system tables, refer to Appendix C, "System Tables." For more information about creating bound databases, refer to the *Database Design Guide*.

If you specify WITH REPLACE, Scalable SQL creates the dictionary even if one already exists in the specified directory, which destroys the existing dictionary. If you do not include WITH REPLACE, Scalable SQL does not create a new dictionary if one already exists.

You must be logged in to an existing dictionary before you can create a new dictionary. Also, you cannot create a new dictionary in the directory that contains the dictionary to which you are logged in. The Demodata directory, which is created by the Pervasive Database installation program, contains an initial set of dictionary files to which you can log in to create your own dictionaries.

To access a new dictionary after creating it, you must log into the dictionary.

To assign referential constraints, your database must be a named database. Use the Scalable SQL Setup utility to name the database. For information about the Scalable SQL Setup utility, see the *Pervasive.SQL User's Guide*.

Example

The following statement replaces a dictionary in the C:\DEMODATA directory.

```
CREATE DICTIONARY USING 'c:\demodata' WITH REPLACE
```

CREATE GROUP

CREATE GROUP group_name [, group_name] ...

The CREATE GROUP statement allows you to create one or more security groups. Use a group_name list to specify the names of the groups to add. Each name must be unique in the dictionary for which you are creating the groups.

Security must be enabled in order to create a group.

After you create a group, use a GRANT (access rights) statement to define the rights for the members of the group.

> **Note:** Scalable SQL commits the changes specified in a CREATE GROUP statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Example

The following statement creates a group named Top_Students:

```
CREATE GROUP Top_Students;
```

The next statement uses a list to create several groups at once:

```
CREATE GROUP Admin, Instructors, Registrars;
```

If an error occurs and Scalable SQL is unable to create a group, it does not create any group in the list. For example, if Scalable SQL is unable to create the Instructors group, then it does not create the Admin or Registrars group.

CREATE INDEX

CREATE [UNIQUE] INDEX index_name ON table_name
index_definition

where

index_definition ::=

(*segment_definition* [, *segment_definition*] ...
)

segment_definition ::= column_name [attribute] ...

['alternate_sequence']

attribute ::= < NULL | CASE | MOD | DESC | ASC >

The CREATE INDEX statement creates a *named index* for a table. The name cannot exceed 20 characters and must differ from all other index and column names in the dictionary. Also, the table name you specify must be of a table that already exists in the dictionary.

If the index name contains a blank, you must use double quotes (") around the name when you pass the name to Scalable SQL to allow Scalable SQL to distinguish between the blanks in index names and the blanks between elements in restriction clauses. For more information about naming indexes, refer to the *Database Design Guide*.

When you no longer need a named index, use a DROP INDEX statement to delete it. In contrast, you cannot drop unnamed indexes (those you created using a WITH INDEX clause in a CREATE TABLE statement).

When you create an index, Scalable SQL indexes every row in the table. The length of time Scalable SQL requires to execute a CREATE INDEX statement depends on the number of rows in the table.

Note: Scalable SQL commits the changes specified in a CREATE INDEX statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Index Columns and Attributes

In a CREATE INDEX statement, you specify a single column or multiple columns (in the case of a segmented index) that compose the index and define the index attributes for each column. You cannot use a column of data type BIT, NOTE, or LVAR in an index. For more information on segmented indexes, refer to "AUTOINC" on page A-6.

Note: If you create a Scalable SQL 4.0 table using an unsigned binary data type, this data type must be converted to integer before you can use the table with Scalable SQL 3.01, because Scalable SQL 3.0.1 does not support the unsigned binary type.

Unless you specify otherwise, an index is collated in ascending order, includes null values, and can contain the same

value in multiple rows. In addition, it is case-sensitive and non-modifiable. To change the defaults, you can use the following keywords in your CREATE INDEX statement:

Attribute Keyword	Description
DESC	Collate in descending order.
NULL	Do not index null values. Applying this attribute to one segment of a segmented index is equivalent to applying this attribute to all segments of the given index.
CASE	Index is not case-sensitive.
MOD	Data is modifiable.
ASC	Collate in ascending order (this is the default behavior).
UNIQUE	Multiple rows cannot have the same index column value. Applying this attribute to one segment of a segmented index is equivalent to applying this attribute to all segments of the given index.

Because you can create only one index at a time with a CREATE INDEX statement, you do not need to use the SEG keyword to specify a segmented index. If you specify more than one column, Scalable SQL creates a segmented index using the columns in the order in which you specify them.

You can specify a column as a segment in more than one index. For example, you can specify a person's last name and first name as one segmented index, and a person's last name and ID as another segmented index.

To specify that the index not allow duplicate values, include the UNIQUE keyword. If the column or columns that make up the index contains duplicate values when you execute the CREATE INDEX statement with the UNIQUE keyword, Scalable SQL returns Status Code 5 and does not create the index.

Note: You should not include the UNIQUE keyword in the list of index attributes following the column name you specify; the preferred syntax is CREATE UNIQUE INDEX.

Alternate Collating Sequence

You can specify an alternate collating sequence (ACS) for columns that are string data types. Specify the ACS as a DOS-formatted path to an ASCII file that contains a valid alternate collating sequence. However, it is recommended that you do not specify an explicit path for the ACS file in the CREATE statement. The Server assumes that the ACS file is in the same directory as the database currently in use. If you must specify a path explicitly, ensure the path is

relative to the Server and not to the Client. For example, if you want to create an index using an ACS file that is on the Client's path "M:\newdata" and M: is mapped to the server's local C: drive, then the CREATE statement should include the path to "c:\newdata" (a path the server understands) as the path to the ASC file. Otherwise, the Scalable SQL engine may return a Status Code 557.

Additionally, you cannot specify an ACS for a column that is case-insensitive, because case-insensitivity itself designates a special collating sequence.

Alternate Collating Sequence File

The 265 bytes of an alternate collating sequence file contain the definition of a collating sequence other than the standard ASCII sequence. You can create a file for the alternate collating sequence for an index in either a CREATE INDEX statement or a CREATE TABLE statement. Following are the directories that are searched for this ACS file:

1. The dictionary location.
2. The data file location.
3. The current directory.

To create an alternate collating sequence file, generate a file in the format specified in Table 2-1.

Table 2-1
Alternate Collating Sequence File Format

Offset	Length	Description
0	1	Signature byte. This byte should contain AC hex.
1	8	An 8byte name that uniquely identifies the alternate - collating sequence to the .MicroKernel Database
9	256	A 256byte map. Each 1byte position in the map - corresponds to the code point that has the same value as the position's offset in the map. The value of the byte at that position is the collating weight assigned to the code point.

For example, to insert a character with 5Dh between the letters U (55h) and V (56h) in the following sequence, byte 5Dh in the sequence contains the value 56h, and bytes 56h through 5Ch in the sequence contain the values 57h through 5Dh:

Figure 2-2
Sample Alternate Collating Sequence

Following is a 9byte header and a 256byte body that represent a collating sequence named UPPER. The header appears as follows:

AC 55 50 50 45 52 20 20 20

The 256byte body appears as follows (with the exception of the offset values in the leftmost column):

00: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

10: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

20: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F

30: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F

40: 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F

50: 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

60: 60 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F

70: 50 51 52 53 54 55 56 57 58 59 5A 7B 7C 7D 7E 7F

80: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

90: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

A0: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF

B0: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF

C0: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF

D0: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF

E0: E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF

F0: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF

The header and body forming this ACS are shipped with Pervasive.SQL as the file UPPER.ALT. UPPER.ALT provides a way to sort keys without regard to case. You can use the UPPER ACS as an example when writing your own ACS.

Offset 61h through 7Ah in the example have been altered from the standard ASCII collating sequence. In the standard ASCII collating sequence, offset 61h contains a value of 61h (representing lowercase *a*). When a key is sorted with the UPPER ACS, the MicroKernel sorts lowercase *a* (61h) with the collation weight at offset 61h: 41h. The lowercase *a* is sorted as if it were uppercase *A* (41h). Therefore, for sorting purposes, UPPER converts all lowercase letters to their uppercase equivalents when sorting a key.

Each 1byte position in the map corresponds to the code point that has the same value as the position's offset in the map. The value of the byte at that position is the collating weight assigned to the code point. For example, to force code point 61h (*a*) to sort with the same weight as code point 41h (*A*), place the same values at offsets 61h and 41h.

The following 256-byte body basically performs the same function as UPPER.ALT's body, except that ASCII

characters preceding the ASCII space (20h) are sorted *after* all other ASCII characters:

00: E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF

10: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF

20: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

30: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

40: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F

50: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F

60: 40 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F

70: 30 31 32 33 34 35 36 37 38 39 3A 5B 5C 5D 5E 5F

80: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F

90: 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F

A0: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

B0: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

C0: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF

D0: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF

E0: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF

F0: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF

In this body, different collating weights have been assigned so that a character's weight no longer equals its ASCII value. For example, offset 20h, representing the ASCII space character, has a collating weight of 00h; offset 41h, representing the ASCII uppercase *A*, has a collating weight of 21h.

To sort keys without regard to case, offset 61h through 7Ah in this example have been altered. As in the body for UPPER.ALT, offset 61h has the same collating weight as offset 41h: 21h. By having the same collating weight, offset 41h (uppercase *A*) sorts the same as offset 61h (lowercase *a*).

Using an Alternate Collating Sequence for a Segmented Index

On a segmented index, you can specify an alternate collating sequence for the segments that are string data types. In such an index, you can use the alternate sequence for some string segments and the standard sequence for others.

Scalable SQL uses the standard collating sequence for all segments of nonstring data types. Scalable SQL supports only one alternate collating sequence for an index. Therefore, when you specify an alternate collating sequence for more than one segment, you must use the *same* sequence for each.

Examples

The following statement creates a modifiable, case-insensitive index for the Person table, based on the Perm_Zip column:

```
CREATE INDEX Zipcode
```

```
ON Person (Perm_Zip MOD CASE);
```

The following statement adds a segmented index (consisting of the City and State columns) to the Person table. (Such an index improves performance if you frequently look up students by city and state.)

```
CREATE INDEX Citystate

ON Person (City CASE MOD, State CASE MOD);
```

CREATE PROCEDURE

CREATE PROCEDURE procedure_name parameter_list ;
procedure_body

where

parameter_list ::= ([*parameter_declaration*

[, *parameter_declaration*] ...])

parameter_declaration ::=

[*parameter_mode*] *parameter_name*

data_specification [*default_clause*]

parameter_mode ::= < IN | OUT | INOUT >

data_specification ::= data_type [(data_length)]

data_type ::= < AUTOINC | BFLOAT | BIT | CHAR |
CHARACTER

| CURRENCY | DATE | DEC | DECIMAL | FLOAT |
INT

| INTEGER | LOGICAL | LSTRING | LVAR | MONEY
| NOTE | NUMERIC

| NUMERICSA | NUMERICSTS | TIME | TIMESTAMP
| UNSIGNED | ZSTRING >

data_length ::= length [, decimal]

default_clause ::= < = | DEFAULT > *literal*

procedure_body ::= < *SQL_procedure_statement* |
EXTERNAL >

SQL_procedure_statement ::=

< *single_SQL_statement* | *compound_statement* >

compound_statement ::= see “BEGIN...END (compound statement)”

The CREATE PROCEDURE statement allows you to create a new stored procedure. This statement does not implicitly drop an existing stored procedure of the same name; instead, Scalable SQL returns Status Code 366 to inform you that the procedure name already exists. (Scalable SQL also returns Status Code 366 if you attempt to create a stored statement in place of an existing stored procedure or create a stored procedure in place of an existing stored statement. Note that stored statements are a v3.0 construct and may not be supported in future versions of Scalable SQL.)

To execute stored procedures, use the CALL statement.

Note: Scalable SQL creates the procedure upon successful execution of the CREATE PROCEDURE statement. Even if you include the statement in a transaction, you cannot roll back the creation of the procedure.

There are some Scalable SQL statements that you can only use in the body of a stored procedure or trigger; these are SQL control statements:

- Compound statements (BEGIN...END)
- If statements (IF...THEN...ELSE)
- Loop statements (LOOP and WHILE)
- Leave statements (LEAVE)

For more information, refer to the syntax descriptions of each SQL control statement.

You can create external stored procedures using Inscribe, Pervasive Software's scripting language. The EXTERNAL keyword in the CREATE PROCEDURE statement is used to define external Inscribe procedures. For more information about using Inscribe to write external stored procedures, refer to the *Inscribe User's Guide*.

Rules for Creating Stored Procedures

The following rules apply to creating stored procedures:

- Do not include the following types of statements within a stored procedure:

CREATE DICTIONARY	DROP DICTIONARY
CREATE PROCEDURE	DROP PROCEDURE
CREATE TRIGGER	DROP TRIGGER
CREATE VIEW	DROP VIEW

- SELECT statements are allowed in stored statements but not in stored procedures. See "Use of SELECT Statements in Stored Procedures".
- Parentheses are required in the parameter list. Parameter names do not require a prefix.
- You cannot reference a cursor, variable, or condition name without declaring it first.
- You cannot use duplicate names for cursors, variables, and conditions. (For example, a variable cannot have the same name as a condition.)
- If a procedure is EXTERNAL, then the default parameter mode of a parameter is IN.
- If a procedure is INTERNAL, then the default parameter mode is undefined. If you specify a constant or a column reference as an argument for a parameter that has been declared with no parameter mode, you

receive an error status if the parameter is needed as a target during the execution of the procedure.

Note: To avoid this error, declare the parameter mode for each parameter.

- You cannot use clauses that specify referential constraints within a stored procedure.

When you create a stored procedure, Scalable SQL performs a cursory semantic check only. Compiling a CREATE PROCEDURE statement does not assume that necessary tables have been created or that necessary variables or cursors have been declared. If you execute a stored procedure and have not declared or created the necessary objects, you will receive an error. Test your procedures thoroughly before using them in a production environment.

Using Data Definition and Data Modification Statements with Stored Procedures

You can use data definition (DDL) and data modification (DML) statements within a stored procedure with the following restrictions:

- You cannot execute a DDL statement in a loop. If a procedure executes a DDL statement, then the procedure must be recalled via a CALL statement before executing it again.
- You cannot execute a DML statement prior to a DDL statement if both statements refer to the same table.
- If a SQL DDL statement refers to a table to which a DECLARE CURSOR statement also refers, you must execute the DDL statement prior to opening the cursor for the first time.

Several layers of looping may exist within the outermost loop; however, every statement that executes after the outer loop begins and before it ends is within the same loop, including any stored procedures being called from within the loop.

For more information about data definition and data modification statements, refer to "Data Definition Statements" in Chapter 1, "SQL Overview."

Use of SELECT Statements in Stored Procedures

SELECT statements are allowed in stored procedures, but only if the SELECT statement is the only statement in the body of the procedure.

SELECT statements are allowed in stored statements but not in stored procedures because the ANSI standard does not allow stand-alone SELECT statements.

Valid Syntax for Stored Procedure with SELECT

CREATE PROCEDURE <procedure name> <parameter list> ;

<SELECT statement>

where <procedure name> and <parameter list> are as defined for current stored procedures.

Note: if you run this statement through an interactive SQL utility that requires

statement separators (such as SQLScope), then change the statement separator in the utility to a character other than the semicolon. (In SQLScope, the only other option is the pound sign [#])

Notice that the BEGIN and END as allowed in the procedure body of current stored procedures are not allowed for this special case stored procedure. The BEGIN and END are not needed and not allowing them reinforces the restriction that a SELECT statement may appear in a stored procedure only if it is the only statement in the body of the stored procedure.

Note: If a SELECT statement appears in the body of a CREATE PROCEDURE statement other than according to the above syntax, then XQLCompile of the statement will return status 902.

Example of Incorrect Usage of SELECT in Stored Statements

A stored procedure consisting of a SELECT statement is not allowed to be called from within another stored procedure. A stored procedure that calls a stored procedure that consists of a SELECT statement will return status 902 during at run time, as shown in the following example:

```
XQLCompile ("CREATE PROCEDURE proc1 ();

                    SELECT * FROM X$File");

XQLCompile ("CREATE PROCEDURE proc2 ();

                     BEGIN

                       INSERT INTO t VALUES (1);

                       CALL proc1 ();

                     END");

XQLCompile ("CALL proc2 ()");
```

The third XQLCompile() will execute the INSERT statement in stored procedure proc2 and attempt to execute the CALL to proc1 with will cause an error and a status 902 to be returned.

The status 902 is returned at run time as opposed to the time the procedure is created (the second XQLCompile() in the above example) because at creation time, a stored procedure can contain a call to a stored procedure that does not yet exist or to a stored procedure that will be dropped and recreated (and redefined) between the time a stored procedure is created and then executed.

SELECT Statement Behavior

The behavior of a stored procedure consisting of a SELECT statement is identical to that of the SELECT statement alone.

Continuing from the previous example:

- Use XQLCompile() to call the stored procedure, XQLCompile ("CALL proc1 ()"); returns status 0 just as XQLCompile() of SELECT statement.
- Use xDescribe() or XQLDescribe() to retrieve view information.
- Use xFetch() and XQLFetch() to fetch records.
- Use xInsert(), xRemove(), xUpdate(); just as you would if SELECT statement had been used with XQLCompile().

Table 2-2
Status Codes Returned when using SELECT Statements in Stored Procedures

Type of Statement	XQL Compile of "EXEC proc" or "CALL proc ()"	XQLExec	Second XQLExec
stored statement with SELECT	-115	0 (executes until SELECT encountered)	-116 (continues execution with statement following the SELECT)
stored procedure without SELECT	-125	-125 (reexecutes the procedure)	-125 (reexecutes the procedure)
stored procedure with SELECT	0	0	0

Parameters of Stored Procedures

In a stored procedure, you can define parameters that allow you to pass values to the procedure when you execute it. When you create a stored procedure that uses parameters, you must declare each parameter. If you call a procedure and use keyword arguments, the parameter names must be those used when the procedure was created.

If you do not specify the data type for the parameters, Scalable SQL returns an error. However, if you do not specify a size, Scalable SQL uses the default size for the data type. For more information about default lengths for data types, refer to Appendix A, “Data Types.”

Stored procedures have four possible parameter modes. These modes determine when Scalable SQL checks the validity of the parameter. The available parameter modes are as follows:

- IN
- OUT

- INOUT
- None of the above (default)

If you do not specify a parameter mode, only the execution path of the procedure determines whether or not it is a valid parameter.

If you do not specify a default value for an IN parameter in a CREATE PROCEDURE statement, you must supply an argument for this parameter when calling the procedure; otherwise, Scalable SQL returns Status Code 808. Values supplied at execution time always take precedence over default values.

If you specify a parameter mode of OUT or INOUT, then the parameter must be a variable, whether you actually assign a value to it or not; you cannot specify a constant.

Examples for CREATE PROCEDURE

The following example creates stored procedure EnrollStudent, which inserts a record into the Enrolls table, given the Student ID and the Class ID. If you execute the following statement in SQLScope, change the statement separator under environment settings.

```
CREATE PROCEDURE EnrollStudent (Stud_id INT(4),
Class_Id INT(4));


INSERT INTO Enrolls VALUES (Stud_id, Class_Id, 0.0);
```

The following example procedure is called by the trigger CheckCourseLimit (see page 2-57 for an example of this trigger). The procedure reads the Class table, using the classId parameter passed in by the caller and validates that the course enrollment is not already at its limit before updating the Enrolls table.

```
CREATE PROCEDURE CheckMax (classId INT(4));

BEGIN

DECLARE NumEnrolled INT(4);

DECLARE MaxEnrollment INT(4);



DECLARE failEnrollment CONDITION FOR SQLSTATE
'09000';



-- Get number currently enrolled for this class.



SET NumEnrolled = (SELECT COUNT (*)

 FROM enrolls

 WHERE Class_ID = classId);



-- Get maximum allowed for this class.
```

```
SET MaxEnrollment = (SELECT Max_Size

 FROM class

 WHERE ID = classId);


-- Test that current enrollment is less than the
maximum allowed.


IF ( NumEnrolled >= MaxEnrollment ) THEN

 SIGNAL failEnrollment;


END IF;

END
```

CREATE TABLE

CREATE TABLE table_name [option] ...

([PRIMARY KEY primary_key_definition,]

[FOREIGN KEY foreign_key_definition,] ...

column_definition_list

)

[WITH INDEX (index_definition)]

where

option ::=

< USING '*path_name*' [WITH REPLACE]

| DCOMPRESS

| PAGESIZE page_size

| PREALLOCATE number_of_pages

| THRESHOLD < 0|1|2|3 >]

| OWNER 'owner_name'

| OWNERACCESS < 0|1|2|3 >

>

primary_key_definition ::= (column_name_list)

column_name_list ::= column_name [, column_name]
...

foreign_key_definition ::=

[foreign_key_name] (column_name_list)

REFERENCES table_name

[ON DELETE < CASCADE | RESTRICT >]

[ON UPDATE RESTRICT]

column_definition_list ::=

column_definition [, column_definition] ...

column_definition ::= column_name
data_specification [**CASE**]

data_specification ::= data_type [(data_length)]

data_type ::=< AUTOINC | BFLOAT | BIT | CHAR |
CHARACTER

| CURRENCY | DATE | DEC | DECIMAL | FLOAT |
INT

| INTEGER | LOGICAL | LVAR | LSTRING | MONEY
| NOTE | NUMERIC

| NUMERICSA | NUMERICSTS | TIME | TIMESTAMP
| UNSIGNED | ZSTRING >

data_length ::= length [, decimal]

index_definition ::=

(*segment_definition* [, *segment_definition*] ...
)

segment_definition ::= *column_name* [*attribute*] ...

['*alternate_sequence*']

attribute ::= < NULL | CASE | MOD | DESC | ASC |
SEG | UNIQUE >

The CREATE TABLE statement allows you to define a table and create the corresponding data file. The table name must be unique within the dictionary. If security is enabled, you must have the Create Table right for the dictionary.

Note: Scalable SQL commits the changes specified in a CREATE TABLE statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Defining the Data File Location

Include a USING clause to specify the physical location of the data file associated with the table. This is necessary when you are creating a table definition for an existing data file, or when you want to specify explicitly the name or physical location of a new data file.

If you do not include a USING clause, Scalable SQL generates a unique file name (with the extension .MKD) and creates the data file in the first directory specified in the data path directory list. (This is the path that the application you are using specifies, or, if the database is named, the first data file path associated with the database name.)

If you include in the USING clause the name of an existing data file, the MicroKernel returns Status Code 59, indicating that the specified file already exists. Scalable SQL creates the data table anyway; the Status Code 59 is simply a warning that the file exists, and Scalable SQL assumes that you specified the appropriate table.

If you are creating a table in a named database, you must use either a simple file name or a relative path in the USING clause. If you specify a relative path, Scalable SQL interprets it relative to the first data file path associated with the database name.

Specifying Data File Options

The options in a CREATE TABLE statement allow you to define physical parameters, an owner name, and owner

access options for the data file.

Using Physical Parameters

The following keywords allow you to specify physical data file parameters in a CREATE TABLE statement:

- DCOMPRESS – Creates the file with the data compression flag set. The default is no compression.
- PAGESIZE – Specifies the data file page size. The pageSize parameter must be a valid page size. Valid page sizes are increments of 512 bytes (e.g., 512, 1024, 1536, etc.). If you do not include the PAGESIZE keyword, Scalable SQL uses a default page size of 4096 bytes.
- PREALLOCATE – Sets the preallocation flag to ON and the allocation word to numberOfPages, where number_of_pages is the number of preallocated pages. The default setting is no preallocation. The valid range for numberOfPages is 1 through 65,535.
- THRESHOLD – Sets the corresponding data file free space flag to the setting you specify (where 0 = 5%, 1 = 10%, 2 = 20%, and 3 = 30%). Include the THRESHOLD keyword *only* if the file contains a variablelength data type. The default setting is 0.

For more information about checking these parameter settings, refer to the *Pervasive.SQL User's Guide*.

Specifying the Owner Name

To assign a file owner name to a table's data file, include the OWNER keyword followed by the appropriate owner name. A file owner name works like a password at the MicroKernel level. A user must provide it (using the SET OWNER statement) before the MicroKernel allows access to the data file. You can use file owner names to protect files from unauthorized access by Btrieve applications or other Scalable SQL applications.

Specifying the Owner Access Restriction

Include the OWNERACCESS keyword followed by the appropriate owner flag value to specify the access restriction placed on the data file. The access restriction applies only if you also assign a file owner name for the file. Table 2-3 shows the meanings of the different flag values.

Table 2-3
File Owner Flag Values

Flag Value	Access Restriction
0	Requires the owner name for any type of access to the file; does not enable data encryption.
1	Permits readonly access without the owner name; does not enable data - encryption.
2	Requires the owner name for any type of access to the file; enables data encryption.

3	**Permits readonly access without the owner name; enables data encryption.**

WITH REPLACE

If you include WITH REPLACE in your CREATE TABLE statement, Scalable SQL creates a new data file to replace the existing file (if the file exists at the location you specified in the USING clause). Scalable SQL discards any data stored in the original file with the same name. If you do not include WITH REPLACE and a file exists at the specified location, Scalable SQL returns Status Code 257 and does not create a new file.

Note: WITH REPLACE affects only the data file; it does not affect the table definition in the dictionary.

Defining Columns

Separate each column definition with a comma. The names you specify for column definitions are the column names that Scalable SQL stores in the dictionary. Column names must be unique within a table, but you can use the same name in more than one table.

For each column name, specify the column's data type and internal storage length in bytes. If you do not specify the length, Scalable SQL creates the column with the default length. For detailed information about each data type, refer to Appendix A, "Data Types."

To specify dataLength for the DECIMAL, NUMERIC, NUMERICSA, and NUMERICSTS data types, use the following length notation:

length,decimal

In this notation, length is the total internal length in bytes, and decimal is the number of displayable decimal places to the right of the decimal point. (The number of digits represented by length depends on the data type. Refer to Appendix A, "Data Types." for information about the internal storage formats of these data types.) Because the decimal places are implied, they are also included in the overall length of the column. You are required to supply a data length for LVAR (maximum length is 32,761 bytes) and NOTE data types. All other data types do not require a data length value.

The column in the following example allows a maximum value of 9.99, using 3 internal bytes:

```
grade_point_average NUMERIC (3,2)
```

Because decimal values are stored with two digits per byte (with a half-byte reserved for the sign), the column in the following example allows a maximum value of 999.99, but uses only 3 internal bytes.

```
interest_rate DECIMAL (3,2)
```

The MONEY data type is equivalent to a DECIMAL type with a predefined decimal specification of 2. The following two definitions have the same internal representation:

Current_Balance DECIMAL(6,2)

Current_Balance MONEY(6)

The default display mask for the MONEY type includes a dollar sign.

Use the CASE keyword when defining a CHARACTER or STRING column if you want Scalable SQL to ignore case when evaluating restriction clauses involving the column.

Note: Use double quotes (") to specify a reserved word as a column name.

Defining the Primary Key

To define referential constraints on your database, you must include a PRIMARY KEY clause to specify the primary key on the parent table. The primary key can consist of one column or multiple columns. The columns you specify must also appear in the columnDefinitions list of the CREATE TABLE statement.

You must define the columns that make up a primary key as a unique index that does not include null values. When you specify a primary key, Scalable SQL creates an index with the specified attributes on the defined group of columns, even if you do not include a WITH INDEX clause.

For more information about primary keys, see the *Database Design Guide*.

Note: You must be logged in to the database using a database name before you can define a primary key or perform any other RI operation.

Defining Foreign Keys

To define a foreign key on a dependent table, include a FOREIGN KEY clause in your CREATE TABLE statement. In addition to specifying a list of columns for the key, you can define a name for the key.

Note: The foreign key name must be unique in the dictionary. If you omit the foreign key name, Scalable SQL uses the name of the first column in the key as the foreign key name. This can cause a duplicate foreign key name error if your dictionary already contains a foreign key with that name.

When you specify a foreign key, Scalable SQL creates an index on the columns that make up the key, even if you do not include a WITH INDEX clause. This index has the same attributes as the index on the corresponding primary key except that it allows duplicate values. To assign other attributes to the index, create it explicitly using a CREATE INDEX statement or a WITH INDEX clause. Then, define the foreign key with an ALTER TABLE statement. When you create the index, ensure that it does not allow null values and that its case and collating sequence attributes match those of the index on the corresponding primary key column.

The columns in a foreign key must be the same data types and lengths and in the same order as the referenced columns in the primary key. The only exception is that you can use an integer column in the foreign key to refer to an AUTOINC column in the primary key. In this case, the two columns must be the same length.

You must be logged in to the database using a database name before you can define a foreign key or perform any other RI operation.

Scalable SQL checks for anomalies in the foreign keys before it creates the table. If it finds conditions that violate previously defined RI constraints, it generates a status code and does not create the table. For more information about RI anomalies, refer to the *Database Design Guide*.

When you define a foreign key, you must include a REFERENCES clause indicating the name of the table that contains the corresponding primary key. The primary key in the parent table must already be defined. In addition, if security is enabled on the database, you must have the Reference right on the table that contains the primary key.

You cannot create a self-referencing foreign key with the CREATE TABLE statement. Use an ALTER TABLE statement to create a foreign key that references the primary key in the same table.

Also, you cannot create a primary key and a foreign key on the same set of columns in a single statement. Therefore, if the primary key of the table you are creating is also a foreign key on another table, you must use an ALTER TABLE statement to create the foreign key.

Delete Rule

You can include an ON DELETE clause to define the delete rule Scalable SQL enforces if a user attempts to delete the parent row to which a foreign key value refers. The delete rules you can choose are as follows:

- If you specify CASCADE, Scalable SQL uses the *delete cascade* rule. When a user deletes a row in the parent table, Scalable SQL deletes the corresponding row in the dependent table.

- If you specify RESTRICT, Scalable SQL enforces the *delete restrict* rule. A user cannot delete a row in the parent table if a foreign key value refers to it.

If you do not specify a delete rule, Scalable SQL applies the restrict rule by default.

Update Rule

Scalable SQL enforces the *update restrict* rule. This rule prevents the addition of a row containing a foreign key value if the parent table does not contain the corresponding primary key value. Scalable SQL enforces this rule whether or not you use the optional ON UPDATE clause, which allows you to specify the update rule explicitly.

For further discussion of delete and update rules, see the *Database Design Guide*.

Defining Indexes

You can define one or more unnamed indexes for a table by including a WITH INDEX clause in your CREATE TABLE statement. If you do not include a WITH INDEX clause, Scalable SQL creates a table without indexes. However, you may add named indexes to that table later using a CREATE INDEX statement.

You cannot delete an unnamed index. (You *can* delete named indexes.)

In a CREATE TABLE statement that creates a primary or foreign key, Scalable SQL creates the required index automatically. Do not use a WITH INDEX clause to create this index because doing so creates a redundant index on the table.

Index Columns and Attributes

In a WITH INDEX clause, you specify the columns that compose the index and define the index attributes for each column.

Note: You cannot use a column of data type BIT, NOTE, or LVAR in an index. For more information on segmented indexes, refer to "AUTOINC" on page -6.

Unless you specify otherwise, an index is collated in ascending order, includes null values, and can contain the same value in multiple rows. In addition, it is case-sensitive, nonsegmented, and non-modifiable. To change the defaults, you can use the following keywords in your WITH INDEX clause:

Attribute Keyword	Description	Default
DESC	Collate in descending order.	Collate in ascending order.
NULL	Do not index null values. Applying this attribute to one segment of a segmented index is equivalent to applying this attribute to all segments of the given index.	Index null values.
CASE	Index is not case-sensitive.	Index is case-sensitive.
SEG	Index consists of multiple columns.	Index consists of one column.
MOD	Data is modifiable.	Data is non-modifiable.
UNIQUE	Multiple rows cannot have the same index column value.Applying this attribute to one segment of a segmented index is equivalent to applying this attribute to all segments of the given index.	Multiple rows can have the same index column value.
ASC	Collate in ascending order.	Collate in ascending order.

To define a segmented index, specify the SEG keyword for each column in the index except the last one (see "Example 2" on page 2-51).

You can specify a column as a segment in more than one index. For example, you can specify a person's last name and first name as one segmented index, and a person's last name and ID as another segmented index.

Alternate Collating Sequence

You can specify an alternate collating sequence for indexes provided the indexes are of a string data type. You cannot, however, specify an alternate collating sequence for an index that is case-insensitive because case-insensitivity is a special instance of an alternate collating sequence.

Specify the alternate collating sequence as a DOS-formatted pathname to an ASCII file that contains a valid alternate collating sequence. Create the alternate collating sequence file as specified in "Alternate Collating Sequence File".

However, it is recommended that you do not specify an explicit path for the ACS file in the CREATE statement. The Server assumes that the ACS file is in the same directory as the database currently in use. If you must specify a path explicitly, ensure the path is relative to the Server and not to the Client. For example, if you want to create an index using an ACS file that is on the Client's path "M:\newdata" and M: is mapped to the server's local C: drive, then the CREATE statement should include the path to "c:\newdata" (a path the server understands) as the path to the ASC file. Otherwise, the Scalable SQL engine may return a Status Code 557.

On a segmented index, you can specify an alternate collating sequence for the segments that are of string data types. In such an index, you can use the alternate sequence for some string segments and the standard sequence for others. Scalable SQL uses the standard collating sequence for all segments of nonstring data types.

Scalable SQL supports only one alternate collating sequence for an index. Therefore, when you specify an alternate collating sequence for more than one segment, use the *same* sequence for each. Otherwise, Scalable SQL uses the first alternate collating sequence you specify.

Example 1

The following statement creates a table that contains four columns, an index that does not allow duplicate values, and a modifiable index that is case-insensitive.

```
CREATE TABLE Course USING 'Course.mkd'

(Name CHAR(7) CASE,

 Description CHAR(50) CASE,

 Credit_Hours UNSIGNED(2),

 Dept_Name CHAR(20))

WITH INDEX (Name UNIQUE CASE, Dept_Name CASE
MOD);
```

You can create a file for the alternate collating sequence for an index in either a CREATE INDEX statement or a CREATE TABLE statement. Following are the directories that are searched for in this ACS file:

1. The dictionary location.
2. The data file location.
3. The current directory.

Note: The current directory in Windows may be unpredictable. The Scalable SQL for Windows engine may change the current directory unpredictably.

Example 2

The following example creates multiple indexes using a single WITH INDEX clause. The first index is segmented, and the second index assumes all of the default attributes.

```
CREATE TABLE Room USING 'Room.mkd'

(Building_Name CHAR(25) CASE,

 Number UNSIGNED(4),
```

```
 Capacity INTEGER(2),
 "Type" CHAR(20) CASE)
WITH INDEX (Building_Name SEG,
 Number UNIQUE, "Type");
```

The SEG (segmentation) keyword following the Building_Name column in the WITH INDEX clause notifies Scalable SQL that this index contains more than one column. Therefore, Scalable SQL creates a segmented index using the Building_Name column and the next column listed, Number. Because the SEG keyword does not follow the Number column, Scalable SQL begins a new index with the next column listed, the Type column.

Example 3

The statements in the following example define three tables and apply RI constraints to them.

```
CREATE TABLE Room USING 'Room.mkd'
(PRIMARY KEY (Building_Name, Number),
 Building_Name CHAR(25) CASE,
 Number UNSIGNED(4),
 Capacity INTEGER(2),
 "Type" CHAR(20) CASE)
WITH INDEX ("Type");
CREATE TABLE Department USING 'Dept.mkd'
(PRIMARY KEY (Name),
FOREIGN KEY DeptLocation (Building_Name,
 Room_Number)
REFERENCES Room ON DELETE RESTRICT,
 Name CHAR(20) CASE,
 Phone_Number NUMERIC(10,0),
 Building_Name CHAR(25) CASE,
 Room_Number UNSIGNED (4),
 Head_Of_Dept UNSIGNED(8))
WITH INDEX (Name UNIQUE CASE,
Building_Name MOD CASE SEG,
Room_Number MOD,
Head_Of_Dept UNIQUE MOD);
```

The Name column is included in the WITH INDEX clause to set additional key attributes. Also, because the primary key for Faculty.Head_Of_Dept does not exist yet, you must add that foreign key to the Department table using an

ALTER TABLE statement after creating the Faculty table.

```
CREATE TABLE Faculty USING 'Faculty.mkd'
(PRIMARY KEY (ID),
FOREIGN KEY Dept (Dept_Name)
REFERENCES Department ON DELETE RESTRICT,
FOREIGN KEY FacultyLocation (Building_Name,
 Room_Number)

REFERENCES Room ON DELETE RESTRICT,
 ID UNSIGNED(8),
 Dept_Name CHAR(20) CASE,
 Designation CHAR(10) CASE,
 Salary CURRENCY(8),
 Building_Name CHAR(25) CASE,
 Room_Number UNSIGNED(8),
 Rsch_Grant_Amount BFLOAT(8))
WITH INDEX (Dept_Name CASE,
Building_Name SEG, Room_Number);
ALTER TABLE Department
ADD FOREIGN KEY DeptHead (Head_Of_Dept)
REFERENCES Faculty ON DELETE RESTRICT;
```

The preceding example creates the foreign key Dept on the Department table. This foreign key refers to the primary keys Building_Name and Room_Number on the Room table. If a user deletes a row from the Room table and a department uses that location, Scalable SQL prevents the row from being deleted (applying the delete restrict rule).

This statement also creates the foreign key FacultyLocation on the Faculty table. This foreign key refers to the primary keys Building_Name and Room_Number on the Room table. If a user attempts to delete a row from the Room table and a faculty member uses that room as an office, Scalable SQL prevents the row from being deleted (applying the delete restrict rule).

CREATE TRIGGER

CREATE TRIGGER [*trigger_name*]
triggered_action_time trigger_event

ON *table_name*

[ORDER *order_value*]

[REFERENCING

< OLD *[* AS *]* *old_correlation_name*

| NEW *[* AS *]* *new_correlation_name*

>

FOR EACH ROW *triggered_action*

where

triggered_action_time ::= < BEFORE | AFTER >

trigger_event ::= < INSERT | DELETE | UPDATE >

order_value ::= *unsigned_integer*

triggered_action ::=

[WHEN (*Boolean_value_expression*)]
SQL_procedure_statement

SQL_procedure_statement ::=

< *single_SQL_statement* | *compound_statement* >

Boolean_value_expression ::= see page 2-121

compound_statement ::= see ”BEGIN...END (compound statement)” on page 2-12.

You use the CREATE TRIGGER statement to declare triggers. This statement does not implicitly drop an existing trigger of the same name; instead, Scalable SQL returns Status Code 364 to inform you that the trigger name already exists.

When a SQL data change statement (INSERT, DELETE, or UPDATE) executes on a table, Scalable SQL checks for triggers defined for that table that contain a trigger event corresponding to the statement. If there is such a trigger, Scalable SQL activates it. If there are multiple such triggers, Scalable SQL activates them in the order that the unsigned integer in the ORDER clause specifies.

No table can have DELETE triggers defined on it if that table already has a foreign key whose primary key has a DELETE rule of CASCADE. No table can have a foreign key defined on it if it already has a DELETE trigger defined on it and its primary key’s DELETE rule is CASCADE. Whichever is defined first blocks the other, and Scalable SQL returns Status Code 368.

When Scalable SQL activates a trigger, the triggered action is executed once for each row: either before the row operation (with BEFORE specified) or after the row operation (with AFTER specified). The triggered action includes the effects of any procedures that it invokes.

Rules for Creating Triggers

The following rules apply to creating triggers:

- The maximum size for a trigger name is 30.
- There is no pre-defined limit on the nesting or cascading of triggers.
- The triggered action must not modify the subject table.
- You cannot use the old correlation name for an INSERT trigger; there is no old image.
- You cannot use the new correlation name for a DELETE trigger; there is no new image.
- The scope of the old correlation name, new correlation name, and subject table name is the entire trigger definition.
- The order number specified by ORDER must be unique. If a duplicate order occurs in conjunction with a duplicate time and event, Scalable SQL returns an error. If you anticipate inserting new triggers within a current ordering scheme, leave gaps in the numbering.
- If you do not designate an order for a trigger, then the trigger is created with a unique order value that is higher than that of any trigger currently defined for that table, time, and event.
- Do not include the following types of statements within a trigger (or within any stored procedure directly or indirectly invoked by a trigger):

ALTER TABLE	BEGIN ATOMIC...END	CREATE DICTIONARY
CREATE GROUP	CREATE INDEX	CREATE PROCEDURE
CREATE TABLE	CREATE TRIGGER	CREATE VIEW
DROP DICTIONARY	DROP GROUP	DROP INDEX
DROP PROCEDURE	DROP TABLE	DROP TRIGGER
DROP VIEW	GRANT *access rights*	GRANT CREATETAB
GRANT LOGIN	REVOKE *access rights*	REVOKE CREATETAB
REVOKE LOGIN	SELECT	

- A trigger body cannot reference a session-level cursor or variable.
- You cannot use a cursor, variable, or condition label without declaring it first.
- You cannot use duplicate names for cursors, variables, and conditions. (For example, a variable cannot have the same name as a condition.)

- With a DELETE trigger, the subject table cannot be the referencing table in any referential integrity definition that specifies ON DELETE CASCADE.
- Use the old correlation name when a column of the old row image is referenced in the triggered action; use the new correlation name for referencing columns of the new row image.
- If the triggered action includes a WHEN clause, then the triggered statement executes only if the Boolean value expression in the WHEN clause evaluates to true. If the Boolean value expression is not true, the triggered SQL statement is not executed. If no WHEN clause is present, the triggered SQL statement executes unconditionally.
- When you create a trigger, any database elements referred to within the triggered action (namely procedures, views, or tables other than the subject table) become dependent elements for that trigger, and you cannot drop or alter them as long as that trigger is defined.
- You must be logged into the database using a database name before you can create a trigger.

Invoking a Trigger

You do not invoke a trigger directly. Scalable SQL automatically invokes triggers as necessary when executing INSERT, UPDATE, or DELETE statements.

Examples

The following example creates a trigger that records any new values inserted into the Tuition table in TuitionIDTable.

```
CREATE TABLE TuitionIDTable USING ’TuitID.mkd’

(Primary Key (ID),

ID unsigned (8))


CREATE TRIGGER InsTrig

BEFORE INSERT ON Tuition

REFERENCING NEW AS InData

FOR EACH ROW

INSERT INTO TuitionIDTable VALUES(InData.ID);
```

The following example is a BEFORE INSERT trigger on the Enrolls table. It invokes the stored procedure CheckMax to ensure that the specified course is not already at its maximum. (Refer to page 2-37 for the example of the CheckMax stored procedure.) The Class_ID column is passed to the stored procedure for purposes of performing the check.

```
CREATE TRIGGER CheckCourseLimit

BEFORE INSERT

ON Enrolls

ORDER 1

REFERENCING NEW AS N
```

```
FOR EACH ROW


BEGIN

 CALL CheckMax (N.Class_ID);

END
```

CREATE VIEW

CREATE VIEW view_name

[(column_heading_name [, column_heading_name]
...)]

AS *select*_statement

where

select_statement ::= see "SELECT" on page 2-117

The CREATE VIEW statement allows you to define a stored view. The name you specify for the view can be a string up to 20 characters long. View names must be unique within a dictionary.

A SELECT statement following the AS keyword defines the data that the view includes. Create the SELECT statement using the same syntax you use in a plain SELECT statement. (See page 2-117 for more information about the SELECT statement.)

Note: Scalable SQL commits the changes specified in a CREATE VIEW statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

The SELECT statement in a CREATE VIEW statement can contain only one query. You cannot use the UNION keyword to combine multiple queries in a single stored view.

Column Headings

If the SELECT statement selects only simple column names from other tables or views, Scalable SQL uses these names as the view headings. However, if any of the selection list items are computed columns, you must define column headings. In all cases, heading names must be unique within the view you are creating.

If you specify a heading for any column in your view, you must specify headings for all the columns. List the headings in the same order that you list their corresponding columns in the SELECT statement.

ReadOnly Views

Generally, you can update and delete rows in views like those in a table. However, you cannot update or delete rows in *readonly* views. A read-only view is one that contains data from a table that has been opened in readonly mode, or one whose SELECT statement meets certain criteria. For more information, refer to "Read-Only Views" on page 2-148.

Example

The following statement creates a view named Phones, which creates a phone list of all the people in the university. This view lists the persons' last names, first names, and telephone numbers, with a heading for each column.

CREATE VIEW phones (lastn, firstn, phone)

AS SELECT Last_Name, First_Name, Phone

```
FROM Person;
```

In a subsequent query on the view, you may use the column headings in your SELECT statement:

```
SELECT lastn, firstn

FROM phones;
```

DECLARE (variable)

DECLARE *SQL_variable_name data_specification* [*default_clause*]

where

data_specification ::= data_type [(data_length)]

data_type ::= < AUTOINC | BFLOAT | BIT | CHAR | CHARACTER

| CURRENCY | DATE | DEC | DECIMAL | FLOAT | INT

| INTEGER | LOGICAL | LSTRING | LVAR | MONEY | NOTE | NUMERIC

| NUMERICSA | NUMERICSTS | TIME | TIMESTAMP | UNSIGNED | ZSTRING >

data_length ::= length [, decimal]

default_clause ::= < = | DEFAULT > *literal*

You use the DECLARE (variable) statement to define a SQL variable. Once declared, you can use a SQL variable in other statements within its defined scope to refer to the current value of the variable. You can also change the value of the variable using SET or FETCH statements.

See the section "Naming Conventions" in the *Database Design Guide* for information regarding the syntax requirements of variable names.

Rules for Declaring a SQL Variable

The following rules apply to declaring a SQL variable:

- A SQL variable defined inside a stored procedure is a procedure-owned variable. Its scope is that procedure in which it is declared, and you can only reference it within that procedure.
- If a procedure calls another procedure, you cannot use a procedure-owned variable of the caller procedure directly in the called procedure; it must be passed as a parameter.
- You must not declare a procedure-owned variable more than once in the same stored procedure; doing so causes Scalable SQL to return Status Code 807.
- When you use the DECLARE statement outside of a stored procedure, the variable is a session variable. Its scope is the login session in which it was declared, and you can refer to it anywhere inside or outside of stored procedures (except within a trigger). You cannot declare a session variable more than once during the same user’s login session.
- The declaration of a SQL variable must precede any reference to the SQL variable name. (An exception to this rule is in a DECLARE CURSOR statement. Refer to DECLARE CURSOR for more information.)
- You can declare the same SQL variable name in different procedures as well as declaring that name as a

session variable. If the same SQL variable name is declared both as a procedure-owned variable and as a session variable, a reference to the variable name within the procedure refers to the procedure-owned variable. If you wish to refer to a session variable inside a procedure, you must not declare a variable of the same name inside the procedure.

Examples

The following examples declare the variables Counter and CurrentCapacity.

```
DECLARE counter INT(2) = 0;

DECLARE CurrentCapacity INT(4) = 0;
```

DECLARE CONDITION

DECLARE *condition_name* CONDITION

FOR SQLSTATE [*value*]
character_string_literal

You use the DECLARE CONDITION statement to declare a condition name and an associated SQLSTATE value. The SQLSTATE value corresponds to a success, warning, or exception condition. For more information about SQLSTATE values, refer to Appendix D, "SQLSTATE Classes and Values."

Rules for Declaring Conditions

The following rules apply to the declaration of conditions:

- You can only declare the same condition name once inside a compound statement.
- No two condition declarations with the same scope can have the same SQLSTATE value associated with them.
- You cannot use the SQLSTATE value for successful completion as the associated value of a condition name.
- You can use the condition label anywhere you use the associated SQLSTATE value.

Example

The following partial example declares a warning condition. A handler is defined to insert a value in the TuitionIDTable table and then continue execution if it encounters the warning.

```
DECLARE Cond1 CONDITION FOR SQLSTATE '01111';


DECLARE CONTINUE HANDLER FOR Cond1

BEGIN

 SET vInteger = vInteger + 1;


 INSERT INTO TuitionIDTable

 VALUES(vInteger);

END;
```

For other examples that use the DECLARE CONDITION statement, refer to the DECLARE HANDLER examples on page 2-69.

DECLARE CURSOR

DECLARE *cursor_name* [SCROLL] CURSOR

FOR *select_statement* [*updatability_clause*]

where

select statement ::= see *"SELECT"*

updatability_clause ::= FOR < READ ONLY | UPDATE
>

You use the DECLARE CURSOR statement to define a SQL cursor. A declared cursor logically ties a SELECT statement to a cursor name.

As the syntax diagram shows, the SELECT statement may or may not include an ORDER BY specification, which has certain implications for the ability to scroll and update the cursor, as described in the rules that follow.

Rules for Declaring Cursors

The following rules apply to declaring cursors:

- A SQL cursor defined inside a stored procedure is a procedure-owned cursor. Its scope is the procedure itself; therefore, it can only be referenced within the procedure. Any procedure may refer to cursors declared within the procedure or to session-level cursors. Cursors declared in procedures are undefined for procedures other than the one in which the cursor is declared.
- A SQL cursor defined outside of any procedure is a session cursor. Its scope is the user's login session. You can refer to it anywhere inside or outside of procedures.
- A SQL cursor declaration must precede any reference to the cursor name.
- You can declare the same cursor name in different procedures as well as declaring that name as a session cursor.
- If the same SQL cursor name appears in a procedure and as a session cursor, then a reference to the cursor name within the procedure references the procedure-owned cursor.
- If you specify SCROLL in a SQL cursor declaration, you can use additional syntax options on a cursor-based FETCH statement.
- Scalable SQL checks a cursor semantically only when it is used in an OPEN statement; therefore, it can contain undeclared variables at the time of its declaration. However, you must define all unresolved references either in a SQL variable declaration or in a parameter declaration of the stored procedure when the cursor is used in an OPEN statement.
- While a cursor is open, it is sensitive to changes committed by other cursors and non-cursor-based statements made by the same and other applications.

Updatability Clause for SQL Cursor Declaration

Table 2-4 illustrates the ability to update SQL cursors based on how the cursor is defined. You may not specify a non-updatable cursor as FOR UPDATE or use it in an UPDATE or DELETE statement.

Table 2-4
Cursor Updatability

Cursor is Updatable	No	Yes	No	No
View is inherently updatable	N	Y	Y	Y
SCROLL specified	N/A	N/A	Y	N/A
ORDER BY specified	N/A	N/A	N/A	Y
FOR UPDATE specified	N/A	Y	N	N

As Table 2-4 shows, a cursor's updatability is based on a combination of the characteristics of the view's query expression and the manner in which you declare the cursor. The following rules apply:

- If the view the query expression defines is not inherently updatable (if it is read-only), then the cursor is non-updatable, regardless of how you specify it.
- If the view is inherently updatable and you specify FOR UPDATE, then the cursor is updatable.
- If the view is inherently updatable and you do not specify FOR UPDATE, then the presence of SCROLL or ORDER BY has the effect of making the cursor non-updatable.
- If the cursor is updatable by the previous criteria, then the presence of FOR READ ONLY makes it non-updatable.

A read-only view is one that contains data from a table that has been opened in readonly mode, or one whose SELECT clause meets certain criteria. For more information, refer to "Read-Only Views" on page 2-148. .

Example

The following example creates a cursor that selects values from the Degree, Residency, and Cost_Per_Credit columns in the Tuition table and orders them by ID number.

```
DECLARE BTUCursor CURSOR

FOR SELECT Degree, Residency, Cost_Per_Credit

FROM Tuition

ORDER BY ID;
```

DECLARE HANDLER

DECLARE *handler_type* HANDLER

FOR *condition_value* [, *condition_value*] ...
handler_action

where

handler_type ::= < CONTINUE | EXIT >

condition_value ::= SQLSTATE [VALUE]

< *character_string*

| *condition_name*

| SQLEXCEPTION

| SQLWARNING

| NOT FOUND

>

handler_action ::= *SQL_procedure_statement*

You can use the DECLARE HANDLER statement to provide handlers for exception or completion conditions in a compound statement. You can only use compound statements and condition handlers in the body of a stored procedure or trigger declaration.

The SQLSTATE value corresponds to a success, warning, or exception condition. For more information about the SQLSTATE session variable, refer to Appendix D, “SQLSTATE Classes and Values.”

In Scalable SQL, execution proceeds until one of the following occurs:

- The procedure terminates normally, based on the procedure logic.
- The procedure encounters an unhandled exception condition.
- The procedure encounters an EXIT handler.

A statement only has the possibility of continuing if no unhandled condition occurs.

Rules for Declaring Handlers

The following rules apply to the declaration of handlers:

- No two handler declarations with the same scope can have components (condition name or character string) in their condition value list that represent the same SQLSTATE value.
- A character string contained in a condition value must not represent the condition for successful completion.
- SQLEXCEPTION corresponds to SQLSTATE values with a class value other than "00", "01", and "02". NOT FOUND and SQLWARNING corresponds to SQLSTATE values with class values of "02" and "01",

respectively.

- If there is a general handler (such as SQLEXCEPTION, NOT FOUND, or SQLWARNING) and a specific handler for the same SQLSTATE value, then only the specific handler is associated with that SQLSTATE value.

- A handler action must not contain a SQL schema statement or a SQL transaction statement.

- During the execution of a compound statement, if an exception condition or a completion condition other than successful completion occurs and there is a declared handler in the compound statement that is associated with the generated SQLSTATE value, then the corresponding handler activates.

  A handler effectively extends the execution of any statement that raised the handler's condition value by appending the operations of the handler to those of the statement. Once activated, the handler and statements within the handler follow the same rules as all other statements:

  - Unhandled exceptions cause execution of the compound statement to terminate.

  - Unhandled, non-exception conditions cause execution to continue to the next logical step (as though the originating statement had caused the most recent condition).

  - No handler can reinvoke itself either due to resignalling or raising some other handled condition.

  - The type of handler (EXIT or CONTINUE) is only taken into account when it causes a non-exception condition to have the same terminating effect as an exception condition. A handler of type CONTINUE cannot cause execution to continue in spite of an exception condition having been raised.

  - The final condition after the execution of all relevant handlers is an unhandled condition that remains after the last statement executed the last handler. This condition may itself be an exception or a non-exception. Successful completion is always *unhandled*.

Examples for DECLARE HANDLER

The following partial example declares a continue handler for Cond1. If the SQLSTATE warning condition occurs, execution continues.

```
DECLARE Cond1 CONDITION FOR SQLSTATE '01111';


DECLARE CONTINUE HANDLER FOR Cond1

BEGIN

 SET vInteger = vInteger + 1;

 INSERT INTO TuitionIDTable

 VALUES(vInteger);

END;
```

The following example returns the total capacity of the largest rooms on campus in the university database. (This example provides the same results as the LargeRooms procedure used for the WHILE example on page 2-174 and the LargeRooms2 procedure used for the LOOP example on page 2-101. The following procedure illustrates a different way of achieving the same results, this time using an exit handler with a LOOP statement.)

```
CREATE PROCEDURE LargeRooms3
```

```
(IN NumRooms INT(4), OUT TotalCapacity INT(4));
BEGIN
DECLARE counter INT(2) = 0;
DECLARE CurrentCapacity INT(4) = 0;
DECLARE tempCapacity INT(4) = 0;

DECLARE cRooms CURSOR
 FOR SELECT Capacity
 FROM Room
 ORDER BY Capacity DESC;

DECLARE endData CONDITION
 FOR SQLSTATE '02000';

DECLARE EXIT HANDLER
 FOR endData

BEGIN
 SET TotalCapacity = tempCapacity;
END;

OPEN cRooms;

SET tempCapacity = 0;

LOOP
 FETCH NEXT FROM cRooms INTO CurrentCapacity;

 IF (counter = NumRooms) THEN
   SIGNAL endData;
 END IF;
```

```
 SET counter = counter + 1;

 SET tempCapacity = tempCapacity +
CurrentCapacity;


END LOOP;


CLOSE cRooms;

END
```

DELETE

DELETE FROM table_reference

[WHERE restriction_clause]

where

table_reference ::= < *table_name* | *view_name* > [*alias_name*]

The DELETE statement allows you to remove rows from a table or an updatable view. When you delete a row from a view, Scalable SQL deletes the corresponding row(s) from the table(s) that makes up the view. You must specify the name of the table or view from which to delete rows. To restrict the rows deleted to those that meet certain criteria, include a WHERE clause in your DELETE statement. (See page 2-121 for the valid syntax of a WHERE clause.)

Note: If you do not specify a WHERE clause, Scalable SQL deletes all the rows in the table or view.

If you define referential integrity (RI) constraints on your database, and if the table (or a table in the view) from which to delete rows is a parent table in a reference path, Scalable SQL enforces the defined delete rules before deleting any rows.

Note: Exercise care when deleting rows from a self-referencing table. If many rows are dependent on the row you delete, the delete cascade rule could cause Scalable SQL to delete all or most of the rows in the table.

For more information about delete rules when RI is enforced, see the *Database Design Guide*.

Example

The following statement deletes the row for Modern European History (HIS 305) from the course table in the sample database:

```
DELETE FROM Course WHERE Name = 'HIS 305';
```

DELETE: positioned

DELETE [FROM *table_reference*]

WHERE CURRENT OF *cursor_name*

where

table_reference ::= < *table_name* | *view_name* > [*alias_name*]

The Positioned DELETE statement deletes the current row of a view associated with a SQL cursor.

Rules for Using the Positioned DELETE Statement

The following rules apply using the Positioned DELETE statement.

- The specified cursor name must be an updatable cursor, and it must be open.
- The FROM table reference clause is optional; the underlying tables to be modified are specified by the declaration of the cursor.
- You must establish a valid position with the FETCH statement before executing a positioned DELETE. If a cursor is not positioned to a row, Scalable SQL returns Status Code 8 (invalid positioning).
- All concurrency controls and rules apply to positioned DELETE statements, including isolation levels, locking, and passive control.
- When the Positioned DELETE statement deletes a row, the new position of the cursor is before the next row. If the Positioned DELETE statement deletes the last row of the table, the new position of the cursor is after the last row.
- All security constraints are enforced as usual.

Example

The following sequence of statements provide the setting for the Positioned DELETE statement. The required statements for the Positioned DELETE statement are DECLARE CURSOR, OPEN CURSOR, and FETCH FROM *cursorname*.

The Modern European History class has been dropped from the schedule, so this example deletes the row for Modern European History (HIS 305) from the Course table in the sample database:

```
DECLARE CourseName CHAR(7);

DECLARE c1 CURSOR

FOR SELECT Name

FROM Course

WHERE Name = CourseName;

SET CourseName = 'HIS 305';

OPEN c1;
```

```
FETCH NEXT FROM c1 INTO CourseName;

DELETE WHERE CURRENT OF c1;
```

DROP DICTIONARY

DROP DICTIONARY USING 'path_name'

The DROP DICTIONARY statement allows you to delete a dictionary from your database. In a USING clause, specify the directory path associated with the dictionary to be dropped. When you drop a dictionary, Scalable SQL drops the dictionary files from the disk, but does not drop the associated data files.

> **Note:** You cannot drop a dictionary if someone is logged into it. Also, Scalable SQL commits the changes specified in a DROP DICTIONARY statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement. For more information about bound databases, refer to the *Database Design Guide*.

Example

The following statement drops the dictionary located in the \TEST directory:

```
DROP DICTIONARY USING '\test'
```

DROP GROUP

DROP GROUP group_name [, group_name] ...

The DROP GROUP statement allows you to delete one or more groups from the dictionary. Specify the names of the groups to drop, separating the names with a comma.

Scalable SQL does not drop a group that has members. Before issuing a DROP GROUP statement, you must first revoke the Login right from each user in the group.

> **Note:** Scalable SQL commits the changes specified in a DROP GROUP statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Examples

If the university decides to move the Registrar group to the Admin group, it should drop the Registrar group. The following statement removes the Registrar group from the dictionary:

```
DROP GROUP Registrar;
```

The following statement removes the Registrar and Instructors group from the dictionary:

```
DROP GROUP Registrar, Instructors;
```

If an error occurs and Scalable SQL is unable to drop a group, it does not drop any group in the list. For example, if Scalable SQL is unable to drop the Instructors group, then it does not drop the Registrar group.

DROP INDEX

DROP INDEX index_name

The DROP INDEX statement allows you to delete a named index. Named indexes are those you create with a CREATE INDEX statement. Specify the name of the index to drop. Because index names are unique in the dictionary, you do not have to specify the corresponding table name.

Note: You cannot drop indexes that are created with a CREATE TABLE statement because these are not named indexes.

The length of time required to drop an index depends on the number of rows in the table. After you drop an index, any subsequent SELECT statement that orders rows by the columns that were indexed requires Scalable SQL to build a temporary index during the SELECT operation.

Note: Scalable SQL commits the changes specified in a DROP INDEX statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement. Also, you cannot drop an index that a foreign or primary key uses. You must drop the key (using an ALTER TABLE statement) first.

Example

The following statement drops the Birthday named index from the Person table:

```
DROP INDEX Birthday;
```

DROP PROCEDURE

DROP PROCEDURE procedure_name

The DROP PROCEDURE statement allows you to delete a stored procedure from the data dictionary. You cannot call a DROP PROCEDURE statement from within a stored procedure.

If you drop a stored procedure that you call from within another stored procedure, Scalable SQL returns Status Code 862 when you execute the other statement.

> **Note:** Scalable SQL commits the changes specified in a DROP PROCEDURE statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

You can drop a stored procedure only if no trigger calls that statement or procedure as its triggered SQL statement.

Example

The following statement drops the stored procedure CheckMax from the dictionary:

```
DROP PROCEDURE CheckMax;
```

DROP TABLE

DROP TABLE table_name

The DROP TABLE statement allows you to delete a table definition from the data dictionary. Specify the name of the table to drop. Scalable SQL drops both the table definition and the corresponding data file.

Note: If you attempt to drop a table that contains a primary key, Scalable SQL returns a status code and does not drop the table. You must first drop the primary key and then drop the table.

Also, Scalable SQL commits the changes specified in a DROP TABLE statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

You cannot drop a table during a transaction if you have previously referred to that table during the transaction. You must first commit the work from the transaction and then drop the table.

You can drop a table only if no trigger refers to it within the triggered action; you cannot drop dependent elements for a defined trigger.

Note: If the associated data file has an owner name defined, use a SET OWNER statement to pass in the owner name before issuing a DROP TABLE statement.

Example

The following statement drops the Tuition table definition from the dictionary:

```
DROP TABLE Tuition;
```

DROP TRIGGER

DROP TRIGGER *trigger_name*

Use the DROP TRIGGER statement to delete a trigger.

When you drop a trigger, procedures, views, or tables that were flagged as dependent elements are no longer flagged; therefore, you can drop or alter them, provided they are not still dependent elements for other triggers.

Example

The following example drops the trigger CheckCourseLimit from the dictionary.

```
DROP TRIGGER CheckCourseLimit;
```

DROP VIEW

DROP VIEW view_name

The DROP VIEW statement allows you to delete a view definition from the data dictionary. Specify the name of the view to drop. Dropping a view definition does not affect the tables referenced in the view.

> **Note:** Scalable SQL commits the changes specified in a DROP VIEW statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

You can drop a view only if no trigger refers to it within the triggered action; you cannot drop dependent elements for a defined trigger.

You cannot drop a view during a transaction if you have previously referred to that view (or a table in that view) during the transaction. You must first commit the work from the transaction and then drop the view.

Example

The following statement drops the Phones view definition from the dictionary:

```
DROP VIEW Phones;
```

For an example of how this view was created, refer to the CREATE VIEW example on page 2-59.

FETCH

FETCH [[*fetch_orientation*] FROM] *cursor_name*

INTO *target [, target]* ...

where

fetch_orientation ::= < NEXT | PRIOR | FIRST |
LAST

| RELATIVE *simple_value* >

simple_value ::= < *literal* | *SQL_variable_name* |
parameter_name

| *column_reference* >

column_reference ::= [column_qualifier.]
column_name

column_qualifier ::= < *table_name* | *view_name* |
alias_name >

target ::= < *SQL_variable_name* | *parameter_name* >

A FETCH statement positions a SQL cursor on a specified row of a table and retrieves values from that row by placing them into the variables in the target list.

Rules for Using the FETCH Statement

The following rules apply to using FETCH statements.

- The cursor denoted by the cursor name must be declared in the current scope and must be in an open state.
- If you do not specify the fetch orientation, NEXT is the default.
- If you specify any option other than NEXT, you must have declared the cursor with SCROLL. The following table lists the behaviors resulting from different specifications of the fetch orientation:

Fetch Orientation	Current Cursor Position
NEXT	Moved to the next logical row if the view is not empty. For all other conditions, moved after the last row of the view and a No Data status returns.
PRIOR	Moved to the previous logical row if the view is not empty. For all other conditions, moved before the first row of the view and a No Data status returns.

FIRST	Moved to the first row of the view if the table is not empty. For all other conditions, moved before the first row of the view and a No Data status returns.
LAST	Moved to the last row of the view if the table is not empty. For all other conditions, moved after the last row of the view and a No Data status returns.

The following table lists the behavior of the RELATIVE fetch orientation and simple value.

Fetch Orientation	Simple Value	Current Cursor Position
RELATIVE	0	Not changed.
positive N		Moved to the *n*th row following the previous current position if such a row exists; otherwise, moved after the last row of the view and a No Data status returns.
negative N		Moved to the *n*th row before the previous current position if such a row exists; otherwise, moved before the first row of the view and a No Data status returns.

If setting the current cursor position does not produce Status Code 9 (no data), values from the current row are assigned to their corresponding targets that the fetch target list identifies.

- The data type of the simple value must be an integer.
- The number of the targets in the fetch target list must be the same as the degree of the table that the cursor specifies. The i-th target in the fetch target list corresponds with the i-th column of the view.
- The type of each target must match the type of the corresponding column in the view of the cursor.

Examples

Because FETCH statements retrieve data associated with SQL cursors, you use them in the same context as DECLARE CURSOR, OPEN CURSOR, and other SQL cursor-oriented statements, such as Positioned UPDATE and DELETE.

The FETCH statement in this example retrieves values from cursor c1 into the CourseName variable. The Positioned UPDATE statement in this example updates the row for Modern European History (HIS 305) in the Course table in the sample database:

```
DECLARE CourseName CHAR(7) = 'HIS 305';


DECLARE OldName CHAR(7);
```

```
DECLARE cursor1 CURSOR

FOR SELECT Name

FROM Course

WHERE Name = CourseName;


OPEN cursor1;


FETCH NEXT FROM cursor1 INTO OldName;


UPDATE SET name = 'HIS 306'

WHERE CURRENT OF cursor1;
```

The following example is the FETCH loop in the stored procedure LargeRooms2. (A full example of this stored procedure is available on page 2-101.) This example returns the total capacity of the largest rooms on campus.

```
FETCH_LOOP:

LOOP

 FETCH NEXT FROM cRooms INTO CurrentCapacity;

 IF (SQLSTATE = '02000' OR counter = NumRooms) THEN

   LEAVE FETCH_LOOP;

 END IF;

 SET counter = counter + 1;

 SET TotalCapacity = TotalCapacity +

   CurrentCapacity;

END LOOP;
```

GRANT (access rights)

GRANT

< < ALL | INSERT | DELETE | ALTER |
REFERENCES >

| < SELECT | UPDATE > [*column_name_list*]

>

ON table_name_list TO *user_list*

where

column_name_list ::= *column_name* [, *column_name*]
...

table_name_list ::= *table_name* [, *table_name*] ...

user_list ::= < user_name | group_name | PUBLIC >

[, < user_name | group_name >] ...

The GRANT (access rights) statement allows you to assign access rights to the following:

- A group or user.
- A list of groups or users.
- All users defined in the dictionary.

To assign access rights to all users in the dictionary, include the PUBLIC keyword to grant the rights to the PUBLIC group, as in the following example:

GRANT SELECT ON Course TO PUBLIC;

This statement assigns the Select right on the Course table to all users defined in the dictionary. If you later revoke the Select right from the PUBLIC group, only users who are granted the Select right explicitly can access the table.

When you grant the Select or Update right to a user, you can specify a list of columns on which the right applies. Otherwise, Scalable SQL assigns the right on all the columns in the table.

If you specify a list of columns in a statement that grants rights on more than one table, each column you name must be contained in every table in the table list. You cannot qualify the column names with a table name. The following statement is valid because the Building_Name column appears in both the Faculty table and the Department table:

GRANT SELECT Building_Name ON Faculty, Department TO
Laura;

However, the next statement is *not* valid because the column Head_of_Department does not appear in the Faculty table:

GRANT SELECT Building_Name, Head_of_Department

ON Faculty, Department TO Laura;

When you grant the All, Insert, Delete, Alter, or References right to a group or user, specify a table name or a list of table names on which to assign access rights. You cannot include a column list in this type of GRANT (access rights) statement because these rights apply to entire tables.

Note: Scalable SQL commits the changes specified in a GRANT (access rights) statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

The rights of a user who is part of a group are limited to those that are defined for the group. To change the rights of a user in a group, you must either change the rights for the entire group or delete the user from the group (using the REVOKE LOGIN statement) and assign the user individual rights.

If an error occurs and Scalable SQL is unable to grant rights to a user or group, it does not grant rights to any user or group in the list. For example, in the following statement, if Scalable SQL is unable to grant rights to Laura, then it does not grant rights to the Registrars group.

GRANT ALTER ON Billing TO Laura, Registrars;

All Right

A GRANT ALL statement grants the Insert, Update, Alter, Select, Delete, and References rights to the specified user or group. In addition, the user or group is granted the Create Table right for the dictionary.

Insert Right

A GRANT INSERT statement grants the Insert right to the specified user or group. The Insert right allows users to insert new rows into a table. If you grant the Insert right to a user, Scalable SQL also grants to that user the Select, Update, and Delete rights on the table.

The following statement gives John the right to insert, select, update, or delete data in the Person table:

GRANT INSERT ON Person TO John;

You cannot specify a column or a list of columns when you grant the Insert right. When you insert values, you insert an entire row; consequently, you cannot specify individual columns. For example, the following statement returns Status Code 524:

GRANT INSERT Last_Name, First_Name, Phone ON Person
TO John;

Delete Right

A GRANT DELETE statement grants the Delete right to the specified user or group. The Delete right allows users to delete rows from a table. If you grant the Delete right to a user, Scalable SQL also grants to that user the Select, Update, and Insert rights on the table.

As with a GRANT INSERT statement, you cannot specify a column or a list of columns in a GRANT DELETE statement.

Alter Right

A GRANT ALTER statement grants the Alter right to the specified user or group. The Alter right allows a user to modify the dictionary definition of a table. When you grant the Alter right, you cannot include a column list in the statement.

The following statement allows Anna to modify the definition of the Person table:

```
GRANT ALTER ON Person TO Anna;
```

When you grant the ALTER right to a user or group, Scalable SQL also grants them the INSERT, UPDATE, and DELETE rights on the table.

References Right

A GRANT REFERENCES statement grants the References right to the specified user or group. The References right allows a user to create foreign keys that refer to a specific table. For example, to create a foreign key that refers to the Person table, you must have the References right on the Person table.

Note: The creator of a table automatically has the References right on that table.

The following example grants user Lisa the right to create foreign keys that refer to the Person table:

```
GRANT REFERENCES ON Person TO Lisa;
```

When you grant the REFERENCES right to a user or group, Scalable SQL also grants them the ALTER right on the table.

Select Right

A GRANT SELECT statement grants the Select right to the specified user or group. The Select right allows a user to read the specified columns in the table.

The following statement grants Select rights to Sarah and James on the Last Name, First Name, and Phone columns in the Person table:

```
GRANT SELECT Last_Name, First_Name, Phone ON Person
TO Sarah, James;
```

To grant Select rights on all the columns in a table, do not include a column list in the statement. For example, the following statement grants Sarah and James the Select right on all the columns in the Person table:

```
GRANT SELECT ON Person TO Sarah, James;
```

Update Right

A GRANT UPDATE statement grants the Update right to the specified user or group. The Update right allows a user to modify the data in the specified columns in the tables. When you grant the Update right, Scalable SQL also grants the Select right on the specified columns.

The following statement allows Sarah to update only the Last Name, First Name, and Phone columns. She cannot update any other columns in the Person table, nor can she insert or delete rows.

```
GRANT UPDATE Last_Name, First_Name, Phone ON Person
TO Sarah;
```

The next statement allows Sarah to update all the columns in the Person table; consequently, she can insert and delete rows, too.

```
GRANT UPDATE ON Person TO Sarah;
```

GRANT CREATETAB

GRANT CREATETAB TO *user_list*

where

user_list ::= < user_name | group_name | PUBLIC >

[, < user_name | group_name >] ...

The GRANT CREATETAB statement allows you to grant a user, a group of users, or all users (the PUBLIC group) the right to create tables. When you grant the Create Table right to a user, that user has full access rights on any table or dictionary that he or she creates. However, you must explicitly grant that user access rights on any table that he or she did not create.

Note: When you include the ALL keyword in a GRANT (access rights) statement, Scalable SQL does not assign the Create Table right in addition to the access rights.

To grant rights to multiple users or groups, list the user or group names in your GRANT CREATETAB statement.

Note: Scalable SQL commits the changes specified in a GRANT CREATETAB statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Examples

The following statement grants the Create Table right to Jim and Brian:

```
GRANT CREATETAB TO Jim, Brian;
```

The following statement includes the PUBLIC keyword to grant the Create Table right to all the users defined in the dictionary:

```
GRANT CREATETAB TO PUBLIC;
```

GRANT LOGIN

GRANT LOGIN TO user_name [:] password

[, user_name [:] password] ...

[IN GROUP group_name]

The GRANT LOGIN statement allows you to assign the Login right to a user (to define a user in the dictionary). You can also add the user to a specific group.

When you grant the Login right to a user, Scalable SQL does not automatically assign to that user access rights to the tables in the dictionary. However, if you add the user to a group for which access rights are already defined, the user assumes those rights.

A GRANT LOGIN statement automatically adds the specified user to the group PUBLIC. If you have defined access rights for the group PUBLIC, the user assumes those rights.

When you issue a GRANT LOGIN statement, you must specify a username and a password for the user. To add the user to a group, you must also include an IN GROUP clause specifying a group name.

Note: Scalable SQL commits the changes specified in a GRANT LOGIN statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Example

The following statement grants the Login right to a user named Cathy and specifies her password as seniors. It also adds her to the group Registrar.

GRANT LOGIN TO Cathy : seniors

IN GROUP Registrar;

IF

IF *Boolean_value_expression*

THEN *SQL_statement_list*

[*if_statement_elseif_clause ...*]

[ELSE *SQL_statement_list*]

END IF

where

if_statement_elseif_clause ::=

ELSEIF *Boolean_value_expression*

THEN *SQL_statement_list*

SQL_statement_list ::= { *SQL_statement* ; } ...

Boolean_value_expression ::= see page 2-121

An IF statement provides conditional execution based on the truth value of a condition. You may use IF statements in the body of both a stored procedure and a trigger.

The keyword ELSEIF has exactly the same meaning as the two keywords ELSE IF. Both are permitted.

IF *Boolean_value_expression*

THEN *SQL_statement_list*

ELSEIF *Boolean_value_expression 1*

THEN *SQL_statement_list 1*

[*if_statement_elseif_clause ...*]

[ELSE *SQL_statement_list*]

The previous example is equivalent to the following:

IF *Boolean_value_expression*

THEN *SQL_statement_list*

ELSE

IF *Boolean_value_expression 1*

THEN *SQL_statement_list 1*

[*if_statement_elseif_clause ...*]

[ELSE *SQL_statement_list*]

```
END IF

END IF
```

Examples

The following example uses the IF statement to set the variable Negative to either 1 or 0, depending on whether the value of vInteger is positive or negative.

```
IF (vInteger < 0) THEN

SET Negative = '1'


ELSE

 SET Negative = '0'


END IF;
```

The following example uses the IF statement to test the loop for a defined condition (SQLSTATE = '02000'). If it meets this condition, then the WHILE loop is terminated.

```
FETCH_LOOP:

WHILE (counter < NumRooms) DO

 FETCH NEXT FROM cRooms into CurrentCapacity;


 IF (SQLSTATE = '02000') THEN

   LEAVE FETCH_LOOP;

 END IF;


 SET counter = counter + 1;

 SET TotalCapacity = TotalCapacity +

   CurrentCapacity;


END WHILE;
```

INSERT

INSERT INTO table_list

[(column_list)]

< { VALUES (value_list) } ...

| *select_statement*

>

where

table_list ::= table_reference [, table_reference]
...

table_reference ::= < *table_name* | *view_name* > [
alias_name]

column_list ::= column_reference [,
column_reference] ...

column_reference ::= [column_qualifier.]
column_name

column_qualifier ::= < table_name | view_name |
alias_name >

value_list ::= *expression* [, expression] ...

The INSERT statement allows you to insert column values into one or more tables. When you issue an INSERT statement, Scalable SQL validates the data you specify and inserts the values into the designated table or tables.

You must include an INTO clause to specify the name of the view, table, or tables to which to add data. If you are inserting data into more than one table, you may want to define aliases for the table names; you can then use the aliases to qualify the column names in the column list.

To specify the columns into which to insert values, list the columns after the table list. Enclose the list of columns in parentheses, and separate the column names with commas. If you are inserting data into multiple tables that contain columns with identical names, use a table name or alias to qualify each column name in the list. If you omit the column list, Scalable SQL assumes you are inserting values into all the columns, in the order they are defined in the table.

Use either of the following methods to specify the data values to insert:

- Specify the data values explicitly by including one or more VALUES clauses.
- Extract the data values from another table in the database by including a SELECT clause.

If the data values you specify (or those the SELECT clause returns) are invalid, Scalable SQL returns a status code or message indicating the problem. Any values that were inserted successfully prior to the error are rolled back; Scalable SQL either executes all data modification (DML) statements to successful completion or leaves the data files

in the same state they were in before execution of the statement.

If you define RI for the database, and if you define the column into which you are inserting values as a foreign key, Scalable SQL verifies that the parent table contains a primary key value that matches the foreign key value you are inserting. If the parent table does not contain a corresponding primary key value, the insert fails.

VALUES Clause

Include a VALUES clause to specify a list of data values; you must specify one value for each column in the column list. When you specify data values, the following rules apply:

- You can specify the values using substitution variables as well as strings or numeric constants. For more information on substitution variables, see the *Database Design Guide*.
- If you do not specify a value for a column of DATE, TIME, or TIMESTAMP, Scalable SQL does not insert the current date or time by default. If you want to insert the current date or time, use the CURDATE and CURTIME variables as values.

The values you define must correspond to the columns you specify in the column list. You must enclose in single quotation marks all string values (date and time values may be enclosed in single quotation marks but do not need to be). If you omit the column list, you must include a value for each column defined in the table or view into which you are inserting data.

If a constant appears in a VALUES clause in an INSERT statement it must not be too large for the column in which it is to be placed. For example, if a table consists of a single column of type DECIMAL (8,2), the following statement fails because the value 1.111 does not fit into the column.

```
INSERT INTO table

VALUES(1.111)
```

The following statement adds data to the Course table by directly specifying the values in three VALUES clauses:

```
INSERT INTO Course(Name, Description, Credit_Hours)

VALUES ('CHE 308', 'Organic Chemistry II', 4)

VALUES ('ENG 409', 'Creative Writing II', 3)

VALUES ('MAT 307', 'Probability II', 4);
```

If the column list does not contain all the columns in the table, Scalable SQL assigns default values to the columns not listed. Scalable SQL uses the following guidelines to determine the default value for a given column:

- If you have defined a default value for the column using the SET DEFAULT statement, Scalable SQL inserts that value.
- If you have issued a SET (global null value) statement to define a null value for all columns of the given column's data type, Scalable SQL inserts that value.
- If you have not defined a default value or a global null value, Scalable SQL assigns a default value based on the column's data type. Scalable SQL sets STRING data types to blanks, AUTOINC columns to the next valid increment, and all others to zero.

The Role of Data Types, Defined Masks, and Default Input Formats

A column's data type and default input format dictate how you can insert and update data using constant values. The following rules apply:

- You must specify constant values using a default input format. (For more information about default input formats, refer to Appendix A, "Data Types.") For data types INTEGER, AUTOINC, and UNSIGNED, Scalable SQL rounds constants to the nearest whole number value. For data types DATE and TIME, Scalable SQL does not implicitly align the constant to any default input format; therefore, embedded blanks are important.

- Constants can be either the operand of a cast expression or a direct operand of a non-cast expression. When the constant is the operand of a cast expression, Scalable SQL interprets the value according to the mask and data type specified for the CAST expression that contains the constant. Otherwise, Scalable SQL interprets the value according to a default input format for the target data type.

Note: For more information on the CAST function and syntax, refer to the *Database Design Guide*.

- If an expression contains a substitution variable, then the constant subsequently assigned to the substitution variable has the same requirements as a constant that would appear at the same position in the expression.

Note: In some cases, Scalable SQL determines a value to be of a certain type, such as a DATE constant or NULL value, and attempts to use the value as interpreted. If the attempt fails, Scalable SQL retries the value as a constant string. To force the engine to interpret the value as a constant string on the first attempt, enclose the value in single quotes.

SELECT Clause

By using a SELECT clause instead of a VALUES clause in an INSERT statement, you can retrieve data from one table or view and insert it into another table or view without any intermediate steps. (A SELECT clause in an INSERT statement is not a subquery, because the syntax does not require the clause to be closed within parentheses.)

The following INSERT statement uses a SELECT clause to retrieve from the Student table the ID numbers of students who have taken classes.

The statement then inserts the ID numbers into the Billing table.

```
INSERT INTO Billing (Student_ID)

SELECT ID

FROM Student

WHERE Cumulative_Hours > 0;
```

Data Type Compatibility

The data types of the columns in the view defined by the SELECT statement must be compatible with the columns of the table into which you are inserting data. If you insert data into a STRING column, then you can select from columns of any data type. If you insert data into a numeric or TIMESTAMP column, then you can select from columns of any data type except LOGICAL and BIT.

If you insert data into a DATE column, then you can select from columns of any data type except LOGICAL, BIT, and TIME. If you insert data into a TIME column, then you can select from columns of any data type except LOGICAL, BIT, and DATE. If you insert data into LOGICAL or BIT columns, then you can select from columns of data type LOGICAL and BIT or columns of any string data type.

Although you can select any data with the SELECT statement, you can only INSERT that data into a given column if the data types are compatible. Table 2-5 indicates which data types you can insert into a given type of column.

Table 2-5
Data Type Convertibility

Column Type (INSERT)	Expression Type (SELECT)					
	numeric	DATE	TIME	TIMESTAMP	string	boolean
numeric	Yes	Yes	Yes	Yes	Yes	No
DATE	Yes	Yes	No	Yes	Yes	No
TIME	Yes	No	Yes	Yes	Yes	No
TIMESTAMP	Yes	Yes	Yes	Yes	Yes	No
string	Yes	Yes	Yes	Yes	Yes	Yes
boolean	No	No	No	No	Yes	Yes

If you attempt to insert an invalid data type into a column, Scalable SQL returns Status Code 223.

The following rules apply to the conversion of data types via a SELECT statement in an INSERT or UPDATE statement:

- When converting a string type to a non-string type, the string value must match the defined mask, if present, on the column being inserted into or updated. If the column has no defined mask, then the string value must match the default mask of the column's data type. If the string value cannot be converted, Scalable SQL returns Status Code 330 (data not formatted according to mask) or 224 (invalid character in numeric data).

- When converting DATE to numeric, the resulting value is the number of days between January 00 1980 and the indicated date. If the indicated DATE is prior to January 00 1980, then the resulting numeric value is negative. For example, if the indicated DATE is January 02 1980, then the numeric value is 2.

- When converting TIME to numeric, the resulting value is the number of one hundredths of a second. For example, if the indicated TIME is 01:01:01:01, then the numeric value is 366101.

- When converting TIMESTAMP to numeric, the resulting value is determined according to the following

formulae:

The TIMESTAMP value is:

'year-month-day hour:minute:second'

The numeric result is:

= (second * 10,000,000)

+ ((minute + (60 * hour)) * 60 * 10,000,000)

+ ((A+B) * 24 * 60 * 60 * 10,000,000)

- ((C + (60 * D)) * 60 * 10,000,000)

where

A = # of days from Jan 1, 0001 to Jan 1, year
B = # of days from Jan 1, year to day, month, year
C = minute in the current time zone
D = hour in the current time zone

- When converting TIMESTAMP to DATE, the resulting value is the date portion of the TIMESTAMP value. For example, if TIMESTAMP is '1996-01-01 01:02:03', then the resulting DATE is 01/01/96.
- When converting TIMESTAMP to TIME, the resulting value is the time portion of the TIMESTAMP value. For example, if TIMESTAMP is '1996-01-01 01:02:03', then the resulting TIME is 01:02:03.
- When converting DATE to TIMESTAMP, the resulting value is the DATE with the time portion of the TIMESTAMP set to zero.
- When converting TIME to TIMESTAMP, the resulting value is the current date with the time portion of the TIMESTAMP set to the TIME value.

LEAVE

LEAVE *statement_label*

A LEAVE statement continues execution by leaving a block or loop statement. You can use it in the body of both a stored procedure and a trigger.

Rules for Using the LEAVE Statement

The following rules apply to using the LEAVE statement:

- The statement label must match the label of some labelled statement in the same scope as the LEAVE statement. This matching label is called the *corresponding label*.

  The body of a stored procedure that is a compound statement can contain a loop statement, and loop statements can be embedded; therefore, the statement label in a LEAVE statement can match the label of any of the embedded loops or the label of the body of the stored procedure.

- The labelled statement can be a labelled compound statement, which may be a procedure body or a trigger body. In all cases, the LEAVE statement causes execution of the labelled statement to terminate as though the last sequential statement had been executed.

Example

The following example increments the variable vInteger by 1 until it reaches a value of 11, when the loop is ended with a LEAVE statement.

```
TestLoop:

LOOP

 IF (vInteger > 10) THEN

   LEAVE TestLoop;

 END IF;


 SET vInteger = vInteger + 1;


END LOOP;
```

LOOP

[*beginning_label* :]

LOOP [*SQL_statement_list*]

END LOOP [*ending_label*]

where

SQL_statement_list ::= { *SQL_statement* ; } ...

A LOOP statement repeats the execution of a block of statements. You can use it in the body of both a stored procedure and a trigger.

If a LOOP statement has a beginning label, it is called a labelled loop statement. If you specify the ending label, it must be identical to the beginning label.

The SQL statement list executes repeatedly, and only a LEAVE statement, an exception condition, or the invocation of an EXIT handler can terminate it.

Examples

The following example increments the variable vInteger by 1 until it reaches a value of 11, when the loop is ended.

```
TestLoop:

LOOP

 IF (vInteger > 10) THEN

   LEAVE TestLoop;

 END IF;



SET vInteger = vInteger + 1;



END LOOP;
```

The following example returns the total capacity of the largest rooms on campus in the university database. (This example provides the same results as the LargeRooms procedure used for the WHILE example on page 2-174. The following example illustrates a different way of achieving the same results.)

```
CREATE PROCEDURE LargeRooms2

(IN NumRooms INT(4), OUT TotalCapacity INT(4));

BEGIN



DECLARE counter INT(2) = 0;

DECLARE CurrentCapacity INT(4) = 0;
```

```
DECLARE cRooms CURSOR
 FOR SELECT Capacity
 FROM Room
 ORDER BY Capacity DESC;

OPEN cRooms;

SET TotalCapacity = 0;

FETCH_LOOP:
 LOOP
   FETCH NEXT FROM cRooms
    INTO CurrentCapacity;

   IF (SQLSTATE = '02000' OR counter = NumRooms)
    THEN
    LEAVE FETCH_LOOP;
   END IF;

   SET counter = counter + 1;
   SET TotalCapacity = TotalCapacity +
    CurrentCapacity;

 END LOOP;

CLOSE cRooms;
END;
```

OPEN (cursor)

OPEN *cursor_name*

The OPEN (cursor) statement opens a cursor.

Rules for Opening a Cursor

The following rules apply to opening cursors:

- The cursor specified by cursor name must not already be open when you issue the OPEN statement.
- As a result of executing an OPEN cursor statement, the current position of the cursor is before the first row of the table.
- All table, column, and variable references in the cursor declaration of a cursor must be valid when you execute the OPEN statement for the cursor.
- If a cursor is opened inside a procedure, and it is not closed before the procedure execution is finished, the cursor is implicitly closed at the end of the procedure.

Example

The following example opens the declared cursor BTUCursor.

```
DECLARE BTUCursor CURSOR

FOR SELECT Degree, Residency, Cost_Per_Credit

FROM Tuition

ORDER BY ID;

OPEN BTUCursor;
```

RELEASE SAVEPOINT

RELEASE SAVEPOINT *savepoint_name*

To delete a savepoint, use the RELEASE SAVEPOINT statement.

The savepoint name must refer to a currently active savepoint in the current transaction.

If a procedure executes without releasing or rolling back a savepoint declared inside it, Scalable SQL automatically releases all savepoints declared within it. Therefore, a reference to the savepoint in a procedure that follows is not valid.

Example

The following example releases the savepoint SP1. For a full example of a transaction that uses the RELEASE SAVEPOINT statement, refer to page 2-115.

```
RELEASE SAVEPOINT SP1;
```

RESIGNAL

RESIGNAL [*signal_value]*

where

signal_value ::=

< *condition_name* | SQLSTATE [VALUE]
character_string_literal >

The RESIGNAL statement allows you to resignal an exception condition or a completion condition other than successful completion. You use this statement in conjunction with declared conditions and declared handlers.

When a handler executes, the statements within it affect the SQLSTATE value in the same way as statements in the main body of the compound statement. However, a handler that is intended to take specific action for a specific condition can optionally leave that condition unaffected, by resignalling that condition at its conclusion. This does not cause the handler to be reinvoked; that would cause a loop. Instead, Scalable SQL treats the exception condition as an unhandled exception condition, and execution stops.

Rules for Using the RESIGNAL Statement

The following rules apply to using the RESIGNAL statement:

- A RESIGNAL statement can only occur in a handler action or in a procedure that a handler invokes.
- If you specify the signal value as a condition name, you must declare the condition name in the scope of this statement with a DECLARE CONDITION statement.
- If you do not specify a signal value, the statement terminates the handler by resignalling the condition which originally caused the handler to be entered. This effectively allows the handler to remember the condition for which it was entered.
- If you specify the signal value, the statement has exactly the same effect as SIGNAL; i.e., it raises the specified exception condition or completion condition by setting the value of SQLSTATE to the specified character string literal or to the value associated with the condition name.

Examples

The following is an example of the RESIGNAL statement.

```
RESIGNAL Cond1
```

Any successful statement sets SQLSTATE back to the success value. So, even though the exit handler was signalled with a '02000', the successful completion of the handler causes the success value to be set and thus to be returned to the caller of this procedure.

The purpose of the RESIGNAL statement is to preserve an original SQLSTATE value after an attempt to handle the condition has changed the SQLSTATE value, either by failing or succeeding. Use of the RESIGNAL statement in the handler below causes the '02000' value to propagate back to the caller, thus indicating the cause of the exit.

```
CREATE PROCEDURE LargeRooms3

(IN NumRooms INT(4), OUT TotalCapacity INT(4));

BEGIN
```

```
DECLARE counter INT(2) = 0;
DECLARE CurrentCapacity INT(4) = 0;
DECLARE tempCapacity INT(4) = 0;

DECLARE cRooms CURSOR
 FOR SELECT Capacity
 FROM Room
 ORDER BY Capacity DESC;

DECLARE endData CONDITION
 FOR SQLSTATE '02000';

DECLARE EXIT HANDLER FOR endData
 BEGIN
   SET TotalCapacity = tempCapacity;
   RESIGNAL;
 END;

OPEN cRooms;
SET TempCapacity = 0;
LOOP
 FETCH NEXT FROM cRooms
   INTO CurrentCapacity;
 IF (counter = NumRooms) THEN
   SIGNAL endData;
 END IF;
 SET counter = counter + 1;
 SET tempCapacity = tempCapacity +
CurrentCapacity;
```

```
END LOOP;

CLOSE cRooms;

END
```

REVOKE (access rights)

REVOKE < < ALL | INSERT | DELETE | ALTER |
REFERENCES >

| < SELECT | UPDATE > [*column_name_list*]

>

ON table_name_list FROM *user_list*

where

column_name_list ::= *column_name* [, *column_name*]
...

table_name_list ::= *table_name* [, *table_name*] ...

user_list ::= < *user_name* | *group_name* | PUBLIC >

[, < *user_name* | *group_name* >] ...

The REVOKE (access rights) statement allows you to remove a group's or user's access rights on the specified tables. To revoke access rights, specify the keyword for the right to remove, and a table name or list of table names. You can use a column_name list to revoke the Select or Update right on specific columns within a table. For information about the access rights All, Insert, Delete, Alter, References, Select, and Update, refer to "GRANT (access rights)" on page 2-84.

Include a FROM clause to specify the group or user from whom you are revoking rights. You can specify a single name or a list of names, or you can include the PUBLIC keyword to revoke access rights from all users whose rights are not explicitly assigned.

When you revoke access rights from a group or user at a particular level, that group or user retains access at all lower levels.

Note: Scalable SQL commits the changes specified in a REVOKE (access rights) statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

You must revoke the Alter right on a table before you can restrict a user's rights on columns in that table.

Examples

Issue the following statement to assign user Sarah the Insert right (which includes the Select, Update, and Delete rights) to the Person table:

GRANT INSERT ON Person TO sarah;

To revoke Sarah's Select right on the ID column, use the following REVOKE statement:

```
REVOKE SELECT ID ON Person FROM sarah;
```

After you issue this REVOKE statement, Sarah no longer has the Insert or Delete right on the table, but she retains the Update and Select rights for all columns in the table except ID.

The following statement revokes from user George the right to create foreign keys that refer to the tables Billing and Tuition:

```
REVOKE REFERENCES ON Billing, Tuition FROM George;
```

REVOKE CREATETAB

REVOKE CREATETAB FROM *user_list*

where

user_list ::= < user_name | group_name | PUBLIC >

[, < user_name | group_name >] ...

The REVOKE CREATETAB statement allows you to revoke the right of one or more groups or users to create tables in a dictionary. Include a FROM clause to specify the user or group names. You can include the PUBLIC keyword in the FROM clause to revoke the Create Table right from all the users to whom the right was not explicitly assigned.

Note: Scalable SQL commits the changes specified in a REVOKE CREATETAB statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Example

The following statement revokes the Create Table right from users John and Carol:

```
REVOKE CREATETAB FROM John, Carol;
```

REVOKE LOGIN

REVOKE LOGIN FROM user_name [, user_name] ...

The REVOKE LOGIN statement allows you to revoke the Login right from a user (to remove a user from the dictionary). As long as security is active, that user can no longer access any tables defined in the dictionary.

You must also use a REVOKE LOGIN statement when removing a user from a group. You begin by revoking the user's Login right; you then issue a GRANT LOGIN statement to recreate the user without a group assignment.

Note: Scalable SQL commits the changes specified in a REVOKE LOGIN statement upon successful execution of the statement. Even if you include the statement in a transaction, you cannot roll back the changes from the statement.

Example

The following statement revokes the Login right from user Susan:

```
REVOKE LOGIN FROM Susan;
```

ROLLBACK WORK

ROLLBACK [WORK] [AND [NO] CHAIN] [
savepoint_clause]

where

savepoint_clause ::= TO SAVEPOINT *savepoint_name*

The ROLLBACK WORK statement allows you to undo all the changes made to your database since the beginning of a transaction, leaving your database in the state it was in before the transaction began. When you issue this statement, the keyword WORK is optional, and a SQL transaction must be active.

Rules for Using the ROLLBACK WORK Statement

The following rules apply to using the ROLLBACK WORK statement:

- If you do not specify AND CHAIN, the default is AND NO CHAIN.
- If you do not specify a savepoint clause, the following conditions occur:
    - All changes made during the current transaction are cancelled.
    - All savepoints defined during the current transaction are destroyed.
    - All open cursors are closed.
    - The current transaction is terminated.
    - If you specify AND CHAIN, a new transaction is issued.
- If you do specify a savepoint clause, the following rules apply:
    - The savepoint name must specify a currently active savepoint in the current SQL transaction.
    - Any changes made during the current transaction after establishing the specified savepoint are cancelled.
    - Any additional savepoints established subsequent to the savepoint identified in the *savepoint name* during the current transaction are deleted.
    - All cursors opened after you establish the savepoint are closed.
    - All cursors opened before you establish the savepoint remain open.
    - If you specify a savepoint clause, you cannot specify the AND CHAIN option.

Note: If you start a transaction and then log out of the dictionary before issuing a COMMIT WORK or ROLLBACK WORK statement, Scalable SQL automatically issues a ROLLBACK WORK statement before completing the logout.

Examples

The following statement undoes the changes made to the database since the beginning of a transaction.

```
ROLLBACK WORK;
```

The following statement undoes the changes made to the database since the last savepoint.

```
ROLLBACK TO SAVEPOINT SP1;
```

For a full example of a transaction that uses the ROLLBACK TO SAVEPOINT statement, see the SAVEPOINT example on page 2-115.

SAVEPOINT

SAVEPOINT *savepoint_name*

Use the SAVEPOINT statement to establish a savepoint. Savepoints are markers in a SQL transaction. Using savepoints allows you to undo a partial set of changes in a transaction and continue with additional changes before requesting the final commit or abort of the entire transaction.

Rules for Creating Savepoints

The following rules apply to creating savepoints.

- The savepoint name must conform to the rules of an identifier. Refer to the *Database Design Guide* for more information about naming database elements.
- You can only use the SAVEPOINT statement if a SQL transaction is currently active. See "START TRANSACTION" for more information about transactions.
- If you use a savepoint name that is associated with a currently active savepoint in the current SQL transaction, Scalable SQL returns Status Code 892 (invalid savepoint specification).
- The MicroKernel allows each transaction a total of 255 internal nesting levels. However, Scalable SQL uses some of these levels internally to enforce atomicity on INSERT, UPDATE, and DELETE statements. Therefore, a session can effectively define no more than 253 savepoints to be active at one time. This limit may be further reduced by triggers that contain additional INSERT, UPDATE, or DELETE statements. If your operation reaches this limit, you must reduce the number of savepoints or the number of atomic statements contained within it.

Example

This example enrolls a student, *student*, into a class *classNum*. Because the first section of the class may be full, this example uses a stored procedure that checks for alternate sections in which to enroll the student. To do so, the stored procedure follows these steps:

1. Establish a savepoint, SP1.
2. Insert the record into the Enrolls table.
3. Determine the current enrollment for the class by assigning the result of a query to *currentEnrollment*.
4. Obtain the maximum size for the class and assign this value to 'maxEnrollment'.
5. Compare this to the maximum size for the class.
6. If the comparison fails, it rolls back to SP1. If the comparison succeeds, it releases SP1.

The calling program or procedure must issue a START TRANSACTION statement before invoking this procedure, because savepoints are only permitted within a transaction. By the same token, however, the caller may issue a COMMIT WORK, regardless of the outcome of this procedure, because no class is ever allowed to exceed its enrollment limit.

```
CREATE PROCEDURE ENROLL_STUDENT

(student UNSIGNED(8), classNum INT(4));

BEGIN

DECLARE currentEnrollment INT(4);
```

```
DECLARE maxEnrollment INT(4);
SAVEPOINT SP1;
 INSERT INTO Enrolls
   VALUES (student, classNum);


 SET currentEnrollment =
   (SELECT COUNT (*) FROM Enrolls
   WHERE class_id = classNum);

 SET maxEnrollment =
   (SELECT Max_Size FROM Class
   WHERE ID = classNum);

 IF (currentEnrollment >= maxEnrollment) THEN
   ROLLBACK TO SAVEPOINT SP1;
 ELSE
   RELEASE SAVEPOINT SP1;
 END IF;
END
```

SELECT

SELECT [DISTINCT] < * | select_list >

FROM < table_name | view_name > [alias]

[, table_name [alias]] ...

[WHERE restriction_clause]

[GROUP BY control_column_list]

[HAVING aggregate_value_restriction_clause]

[ORDER BY < column_reference | position > [
DESC]

[, < column_reference | position > [DESC] ...
]

select_list ::= select_term [, select_term] ...

select_term ::= < column_reference | expression

| aggregate_value_function > [AS *item_name*]

column_reference ::=

[column_qualifier.] column_name [[edit_mask]
]

column_qualifier ::= < table_name | view_name |
alias_name >

The SELECT statement allows you to retrieve data from the database. To specify the columns to retrieve, replace *select_list* with one or more column names, expressions, or group aggregate functions, using commas to separate the items. To retrieve all the columns in a table, you can specify an asterisk (*) instead.

A SELECT statement creates a temporary view. To save a view in order to recall it later, create the view using a SELECT clause in a CREATE VIEW statement; Scalable SQL stores the view in the dictionary.

Example

The following SELECT statement retrieves all the data in the Person table:

SELECT * FROM Person;

The following statement retrieves only the first and last names from the Person table:

SELECT First_Name, Last_Name FROM Person;

DISTINCT

Include the DISTINCT keyword in your SELECT statement to direct Scalable SQL to remove duplicate values from the result. By using DISTINCT, you can retrieve all unique rows that match the SELECT statement's specifications.

The following rules apply to using the DISTINCT keyword:

- You can use DISTINCT in any statement that includes subqueries.
- The DISTINCT keyword is ignored if the selection list contains an aggregate; the aggregate guarantees that no duplicate rows will result.
- The following usage of DISTINCT is not allowed:

  SELECT DISTINCT column1, DISTINCT column2

Example

The following statement retrieves all the unique courses taught by Professor Beer (who has a Faculty ID of *111191115*):

```
SELECT DISTINCT Name

FROM Course c, class cl

WHERE c.name = cl.name AND cl.faculty_id =
'111191115';
```

The result of the query is as follows:

Course
CHE203
CHE205

Select List

The select term in the select list can be a column reference or expression. Because session variable names, local variable names, and column names can be the same, Scalable SQL uses the following order of precedence to determine the intent of the name:

1. Local variable names
2. Session variable names
3. Column names

For example, if you issue the statement SELECT C FROM T and C designates both a local variable and a column, Scalable SQL returns the value of the local variable C. Similarly, if C designates both a session variable and a local variable, Scalable SQL returns the value of the local variable.

To avoid naming conflicts with local and session variable names, you can use a qualified column name, as in the statement SELECT T.C FROM T.

AS

Include an AS clause to assign a name to a select term. You can use this name elsewhere in the statement to

reference the select term. When you use the AS clause on a non-aggregate column, you can reference the name in WHERE, ORDER BY, GROUP BY, and HAVING clauses. When you use the AS clause on an aggregate column, you can reference the name only in an ORDER BY clause.

The name you define must be unique in the SELECT list.

Example

The AS clause in the following statement instructs Scalable SQL to assign the name Total to the select term SUM (Amount_Paid) and order the results by the total for each student:

```
SELECT Student_ID, SUM (Amount_Paid) AS Total

FROM Billing

GROUP BY Student_ID

ORDER BY Total
```

FROM

Include a FROM clause to specify the table or view to query. The valid syntax for a FROM clause is as follows:

```
FROM < table_name | view_name > [ alias ]

[ , table_name [ alias ] ... ]
```

You can specify a single table or view, multiple tables, or a single view and multiple tables. When you specify more than one table in a FROM clause, the tables are said to be *joined*. For more information about joins, see the *Database Design Guide*.

If you are retrieving data from a view, you can apply restriction conditions to the query just as you would if you were querying a table. However, any restrictions you defined when you created the view apply in addition to the restrictions you specify in the SELECT statement.

You can assign an alternate name, or *alias*, to the view or tables from which you are retrieving data. When you refer to the view or tables elsewhere in the SELECT statement, you can use the alias instead of the actual table or view name. For example, you can assign a short name to a table and use that short name to qualify column names in the select list, in a WHERE clause, or in an ORDER BY clause.

Example

The FROM clause in the following statement instructs Scalable SQL to retrieve data from both the Person table and the Faculty table, using the aliases *p* and *f* to distinguish between the two tables:

```
SELECT p.id, f.salary

FROM Person p, Faculty f

WHERE p.id = f.id;
```

WHERE

Include a WHERE clause in your SELECT statement to define one or more search criteria that qualify the data returned. The valid syntax for a WHERE clause is as follows:

WHERE < Boolean_value_expression | expression
relational_operator

[< ALL | ANY | SOME >] subquery

| expression [NOT] IN subquery

| [NOT] EXISTS subquery

>

Boolean_value_expression ::= condition

[Boolean_operator condition] ...

condition ::= expression < relational_operator |
range_operator >

expression

expression ::= < column_reference | constant |
scalar_function >

[expression_operator

< column_reference

| constant

| scalar_function

>] ...

column_reference ::= [column_qualifier.]
column_name

column_qualifier ::= < table_name | view_name |
alias >

You can also use a WHERE clause in UPDATE and DELETE statements.

A restriction clause can define *restriction conditions*, *join conditions*, or both:

- A restriction condition compares an expression that references a column value either to a constant or to another expression that references a column value in the same table.
- A join condition compares an expression that references a column value from one table to an expression that references a column value from another table.

A restriction clause can also contain a SELECT subquery that allows you to base search criteria on the contents of other tables in the database. For information about how to include a subquery in a WHERE clause, refer to "Subqueries" on page 2-125.

Expressions

An expression can be a numeric value, a string, or any operation that evaluates to a numeric value or a string. The following rules apply to Scalable SQL expressions:

- The maximum length of a Scalable SQL expression is limited only by available memory.
- You must enclose strings in single quotes.
- You must enter all values as ASCII values.
- Parentheses define the precedence of operations in an expression. If none of the operations in the expression are enclosed within parentheses, Scalable SQL evaluates the expression according to ANSI standards.

  Arithmetic precedence is as follows:

  b. Unary operator

  c. Modulo, division, multiplication, append, and integer division

  d. Addition, subtraction, concatenate, and concatenate with spaces

  Boolean precedence is as follows:

  a. AND

  b. OR

  Operators of the same precedence cause left to right evaluation of the affected expressions, as follows:

  10 - 10 + 3 is evaluated the same as (10 - 10) + 3

- With the IN and NOT IN keywords, the second expression can be a subquery instead of a numeric value, a string, or an operation that evaluates to a numeric value or a string. (See "Subqueries" on page 2-125.)
- You must specify constant values using a default input format. (For more information about default input formats, refer to Appendix A, "Data Types.") For data types INTEGER, AUTOINC, and UNSIGNED, Scalable SQL rounds constants to the nearest whole number value. For data types DATE and TIME, Scalable SQL does not implicitly align the constant to any default input format; therefore, embedded blanks are important.
- Constants can be either the operand of a cast expression or a direct operand of a non-cast expression. When the constant is the operand of a cast expression, Scalable SQL interprets the value according to the mask and data type specified for the CAST expression that contains the constant. Otherwise, Scalable SQL interprets the value according to a default input format for the target data type.
- If an expression contains a substitution variable, then the constant subsequently assigned to the substitution variable has the same requirements as a constant that would appear at the same position in the expression.

Note: In some cases, Scalable SQL determines a value to be of a certain type, such as a DATE constant or NULL value, and attempts to use the value as interpreted. If the attempt fails, Scalable SQL retries the value as a constant string. To force the engine to interpret the value as a constant string on the first attempt, enclose the value in single quotes.

- When you specify a WHERE clause using an application that supports substitution variables, you can use substitution variables to specify constants.

For more information about expressions, refer to the *Database Design Guide*.

Operators

You can use three categories of operators in restriction clauses:

Boolean operator	Connects two conditions within a restriction clause. The Boolean operators are AND and OR.
Condition operator	Applies to two expressions to form a condition. A condition operator can be one of the relational operators (<, >, =, <=, >=, <>, !=) or one of the range operators (IN, NOT IN, BETWEEN, NOT BETWEEN, BEGINS WITH, CONTAINS, NOT CONTAINS, IS NULL, IS NOT NULL, LIKE, NOT LIKE).
Expression operator	Applies to two expressions to form another expression. An expression operator can be one of the arithmetic operators (+, –, * /, //, %) or one of the string operators (*, +, ++).

For more information about using operators in restriction clauses, refer to the *Database Design Guide*.

Subqueries

To base search criteria on the contents of a table other than the table you are querying, include a SELECT subquery in the WHERE clause.

Note: You can include a subquery in the WHERE clause of a SELECT, UPDATE, or DELETE statement, or in the HAVING clause of a SELECT statement. However, HAVING clauses cannot contain correlated subqueries.

The following rules apply when you create a subquery:

- You must enclose the subquery in parentheses.
- Unless you use the EXISTS or NOT EXISTS keyword in the WHERE clause of the outer query, the select list of the subquery can contain only one column name expression.
- The subquery cannot contain the UNION keyword.

Scalable SQL processes the subquery first and passes the results to the outer query, except in the case of *correlated subqueries*. A correlated subquery contains a WHERE or HAVING clause that references a table from the outer query's FROM clause. Use a correlated subquery to do either of the following:

- To test the results that a subquery returns against the results that the outer query returns.
- To test for the existence of a particular value in a query.

When you issue a statement with a correlated subquery, Scalable SQL tests the subquery against the result of each row being evaluated by the outer query. The statement returns a row only if the specified correlation exists between the returned values of the subquery and those of the outer query.

Scalable SQL allows you to nest several levels of subqueries in a statement. The amount of memory available to Scalable SQL determines the number of subqueries you can nest in a single statement.

The way Scalable SQL processes a statement containing a subquery depends on how you specify the subquery in the WHERE clause. If you precede the subquery with an expression and a relational operator, you can use the ALL, ANY, and SOME keywords to affect the result. Alternatively, you can precede the subquery with EXISTS or NOT EXISTS, or with an expression and the IN or NOT IN operator.

ALL

When you specify the ALL keyword before a subquery, Scalable SQL performs the subquery and uses the result to evaluate the condition in the outer query. If *all* the rows the subquery returns meet the outer query's condition for a particular row, Scalable SQL includes that row in the final result table of the statement.

Generally, you can use the EXISTS or NOT EXISTS keyword instead of the ALL keyword.

The following SELECT statement compares the ID column from the Person table to the ID columns in the result table of the subquery:

```
SELECT ID, Last_Name

FROM Person

WHERE ID <> ALL

 (SELECT ID FROM Faculty WHERE Dept_Name =
'Chemistry');
```

If the ID value from Person *does not equal any* of the ID values in the subquery result table, Scalable SQL includes the row from Person in the final result table of the statement.

ANY and SOME

The ANY and SOME keywords are equivalent. They work similarly to the ALL keyword except that Scalable SQL includes the compared row in the result table if the condition is true for *any* row in the subquery result table.

The following statement compares the ID column from Person to the ID columns in the result table of the subquery. If the ID value from Person is equal to *any* of the ID values in the subquery result table, Scalable SQL includes the row from Person in the result table of the SELECT statement.

```
SELECT ID, Last_Name

FROM Person p

WHERE p.id = ANY

 (SELECT ID FROM Faculty WHERE Dept_Name =
'Chemistry');
```

EXISTS and NOT EXISTS

Use the EXISTS keyword to test whether rows exist in the result of the subquery. For every row the outer query evaluates, Scalable SQL tests for the existence of a related row from the subquery. Scalable SQL includes in the statement's result table each row from the outer query that corresponds to a related row from the subquery.

Conversely, the NOT EXISTS keyword allows you to test whether rows *do not* exist in the result of the subquery. For every row the outer query evaluates, Scalable SQL tests for the existence of a related row from the subquery. Scalable SQL *excludes* from the statement's result table each row from the outer query that corresponds to a related

row from the subquery.

For example, the following statement returns a list containing only persons who have a 4.0 grade point average:

```
SELECT * FROM Person p WHERE EXISTS

(SELECT * FROM Enrolls e WHERE e.Student_ID = p.id

 AND Grade = 4.0);
```

The following statement returns a list of students who are not enrolled in any classes:

```
SELECT * FROM Person p WHERE NOT EXISTS

(SELECT * FROM Student s WHERE s.id = p.id

 AND Cumulative_Hours = 0);
```

IN and NOT IN

Use the IN operator to test whether the result of the outer query is included in the result of the subquery. The result table for the statement includes only rows the outer query returns that have a related row from the subquery.

Conversely, the NOT IN operator allows you to test whether the result of the outer query is *not* included in the result of the subquery. The result table for the statement includes only rows the outer query returns that *do not* have a related row from the subquery.

For example, the following statement lists the names of all students who have taken Chemistry 408:

```
SELECT First_Name * Last_Name

FROM Person p, Enrolls e

WHERE (p.id = e.student_id)

AND (e.class_id IN

 (SELECT ID FROM Class WHERE Name = 'CHE 408'));
```

Scalable SQL first evaluates the subquery to retrieve the ID for Chemistry 408 from the Class table. It then performs the outer query, restricting the results to only those students who have an entry in the Enrolls table for that course.

The preceding query returns these results:

Person.First_Name * Person.Last_Name
Erik Domaas
Cathy Duda
Dana Dyer
Herbert Eburne

Richard Egyud

Lee Ragin

William Rejincos

Craig Rideau

Ladislao Ruksenas

Rebecka Ryiz

Fabian Ipock

Bruno Ippolite

Ernest Ipsen

Donald Ittner

Tarmo Jaaskelainen

Often, you can perform IN queries more efficiently using either the EXISTS keyword or a simple join condition with a restriction clause. Unless the purpose of the query is to determine the existence of a value in a subset of the database, it is more efficient to use the simple join condition because Scalable SQL optimizes joins more efficiently than it does subqueries.

GROUP BY

Include a GROUP BY clause to group rows and determine the aggregate values for one or more columns in the group. The valid syntax for a GROUP BY clause is as follows:

```
GROUP BY control_column_list


control_column_list ::= < column_reference |
position >

[, < column_reference | position > ... ]


column_reference ::= [ column_qualifier. ]
column_name
```

column_qualifier ::= < table_name | view_name |
alias >

The aggregate values you can calculate for a column are minimum (MIN), maximum (MAX), average (AVG), sum (SUM), or count (COUNT). For information about using the group aggregate functions, refer to "Group Aggregate Functions" on page 2-135.

In the GROUP BY clause, specify one or more control columns (the columns by which to group rows). You must also specify the control columns in the select list of the statement. Otherwise, Scalable SQL returns Status Code 827.

You can specify a control column in either of two ways:

- Specify the name of the column (qualified by the table name, view name, or alias, if necessary).
- Specify the column by its position in the view (based on where it appears in the select list). The column positions are numbered from 1 to the number of columns returned in the view. The integer value you specify must be in that range.

If the select list contains a computed column, you must use the column position (not the column name) to specify the columns by which to group rows.

The select list must contain the name of any column you specified as a control column; all other select items in the list must be aggregate value functions. Also, the control columns *cannot* be group aggregate functions (but they can be computed columns).

You can apply the DISTINCT keyword to a group aggregate function. In response, Scalable SQL includes only the unique occurrences of a value when it calculates the result.

Example

You can determine the number of classes per student and the grade average for each with the following statement:

```
SELECT p.ID, COUNT(e.Class_ID), AVG(Grade)

FROM Person p, Enrolls e

WHERE (p.ID = e.Student_ID)

GROUP BY p.ID;
```

The preceding example uses these aggregate value functions:

```
COUNT(e.Class_ID)


AVG(Grade)
```

The DISTINCT keyword in the following statement causes Scalable SQL to retrieve only unique occurrences of the values in the Name column. Thus, the following statement retrieves the number of types of classes a faculty member teaches.

```
SELECT COUNT (DISTINCT Name), Class.Faculty_ID

FROM Class

GROUP BY Faculty_ID;
```

HAVING

Use a HAVING clause in conjunction with a GROUP BY clause (see page 2-129) to limit your view to groups whose aggregate values meet specific criteria.

Note: Although you can specify a HAVING clause without a GROUP BY clause, doing so causes Scalable SQL to treat the entire table as a single group.

The valid syntax for a HAVING clause is as follows:

HAVING aggregate_value_restriction_clause

aggregate_value_restriction_clause ::=

condition [< AND | OR > condition ...]

condition ::= aggregate_value_function <
relational_operator >

< value | subquery >

The value you specify for *aggregate_value_restriction_clause* can contain multiple conditions. The first expression in a condition must be an aggregate value function. (For information about specifying an aggregate value function, see "Group Aggregate Functions" on page 2-135.) The second expression can be a substitution variable, a string or numeric constant, or a subquery. Although HAVING clauses can contain subqueries, they cannot contain correlated subqueries.

Example

In the following statement, the HAVING clause limits the returned data to students who paid at least $100:

```
SELECT Student_ID, SUM(Amount_Paid)

FROM Billing

GROUP BY Student_ID

HAVING SUM(Amount_Paid) >= $100.00;
```

ORDER BY

Include an ORDER BY clause to specify the order in which Scalable SQL returns the rows you request. The valid syntax for an ORDER BY clause is as follows:

ORDER BY < column_reference | position > [< ASC
| DESC >]

[, < column_reference | position > [< ASC |
DESC >]] ...

column_reference ::= [column_qualifier.]
column_name

column_qualifier ::= < table_name | view_name |
alias_name >

If you do not specify an ORDER BY clause, Scalable SQL returns the rows in an undefined order. This is because Scalable SQL may change the order to optimize performance. An ORDER BY clause overrides any order Scalable SQL may otherwise use.

You can specify the column or columns by which to order the resulting view in either of two ways:

- Specify the name of the column (qualified by the table name, view name, or alias, if necessary).
- Specify the column by its position in the view (based on where it appears in the select list). The column positions are numbered from 1 to the number of columns returned in the view. The integer value you specify must be in that range.
    - If the ORDER BY clause is part of a SELECT clause in a union, or if the select list contains a computed column, you must use the column position (not the column name) to specify the columns by which to order rows.
    - The ORDER BY clause is ignored in all SELECT clauses except the last SELECT in the union. If the ORDER BY clause in the last SELECT clause contains a column name rather than a column position, then Scalable SQL returns Status Code 859.

When you specify an ORDER BY clause, its effect on performance depends on the table's defined indexes.

If the column is not defined as an index, Scalable SQL must build a temporary sort file to contain the new index. In this case, there may be a delay before Scalable SQL returns the row you request, depending on the total size of your data file. For any column you specify in an ORDER BY clause, you have the option of sorting the column in ascending (ASC) or descending (DESC) order.

Note: By default, an ORDER BY clause returns rows in ascending order, even if the index is in descending order. Specify the DESC keyword to retrieve rows in descending order.

Examples

The following statement returns a list of classes ordered by their instructor's name. All the classes of one instructor are listed, followed by all the courses of another instructor, and so forth.

```
SELECT DISTINCT Name, First_Name +' ' + Last_Name

FROM Class c, Person P

WHERE c.Faculty_ID = p.ID

ORDER BY Faculty_ID;
```

The next statement further sorts the data obtained by the preceding statement so that the courses for each instructor appear in alphabetic order:

```
SELECT DISTINCT Name, First_Name +' ' + Last_Name

FROM Class c, Person P

WHERE c.Faculty_ID = p.ID

ORDER BY Faculty_ID, Name;
```

The following statement uses the DESC keyword to list classes, beginning with the highest in credit hours:

```
SELECT Name, Description, Credit_Hours

FROM Course

ORDER BY Credit_Hours DESC;
```

You can specify a sort column by giving its position in the select list. Thus, the following two statements produce the same result table, because Credit Hours is the third column in the select list:

```
SELECT Name, Description, Credit_Hours

FROM Course

ORDER BY Credit_Hours DESC;
```

```
SELECT Name, Description, Credit_Hours

FROM Course

ORDER BY 3 DESC;
```

Specifying a sort column by giving its position in the select list is essential if the select item on which to sort is not a column name. For example, the following statement retrieves people's names in alphabetic order, by full name:

```
SELECT Last_Name ++ ', ' * First_Name

FROM Person

ORDER BY 1;
```

Group Aggregate Functions

The group aggregate functions allow you to determine the minimum (MIN), maximum (MAX), average (AVG), sum (SUM), or count (COUNT) of one or more columns. You can use a group aggregate function in a selection list or in a HAVING clause. For general information about using group aggregate functions, see the *Database Design Guide*.

AVG

```
AVG ( < [ DISTINCT ] column_name | expression > )
```

Use the AVG function in a SELECT statement to calculate the average value of a column containing numeric values, or the average value of a numeric expression. If you include the DISTINCT keyword, Scalable SQL calculates the average of distinct values only.

Example

The following statement finds the average grade of students:

```
SELECT AVG (Grade) FROM Enrolls;
```

COUNT

COUNT ([DISTINCT] column_name) *or* COUNT (*)

Use the COUNT function in a SELECT statement to count the number of values in the specified column. When you use the DISTINCT keyword with the COUNT function, Scalable SQL does not include duplicate values in the result.

If you use COUNT (*) in a selection list, Scalable SQL counts all the rows in the result table. However, if the SELECT statement contains a GROUP BY clause and you include COUNT (*) in the selection list, Scalable SQL counts all the values in the column specified in the GROUP BY clause.

Example

The following statement finds the number of different departments listed in the Course table:

```
SELECT COUNT (DISTINCT Dept_Name) FROM Course;
```

The result of this statement is 22.

MAX

MAX (< column_name | expression >)

You can use the MAX function in a SELECT statement to determine the largest value in a column or expression.

Example

The following statement finds the highest outstanding balance:

```
SELECT MAX (Amount_Owed) FROM Billing;
```

The result of this statement is $6000.00.

MIN

MIN (< column_name | expression >)

You can use the MIN function in a SELECT statement to determine the smallest value in a column or expression.

Example

The following statement finds the lowest outstanding balance:

```
SELECT MIN (Amount_Owed)
```

```
FROM Billing;
```

The result of this statement is $1200.00.

SUM

SUM (< [DISTINCT] column_name | expression >)

You can use the SUM function in a SELECT statement to calculate the sum of the values in a column containing numeric values, or the sum of the values of a numeric expression. If you include the DISTINCT keyword, Scalable SQL calculates the sum of distinct values only.

Example

The following example determines the total amount of money each student has paid:

```
SELECT Student_ID, SUM (Amount_Paid)
FROM Billing
GROUP BY Student_ID;
```

Combining Multiple Queries with UNION

The UNION keyword allows you to obtain a single result table from multiple SELECT queries. The valid syntax is as follows:

```
select_statement
UNION [ ALL ]
select_statement
[ UNION [ ALL ]
select_statement ...]
```

You can combine the result of any legal SELECT clause with that of another legal SELECT clause, provided the clauses return comparable columns.

If one of the SELECT clauses contains an ORDER BY clause, you must specify the ordering column by its position in the selection list, not by its column name.

Scalable SQL uses the column names from the first query as the name of the result columns. For example, the result of the following statement is a view of one column (all the values in Column1 and all the values in Column2); the column name of the result column is Column1.

```
SELECT column1 FROM table1
UNION
SELECT column2 FROM table2
```

By default, the result column does not include duplicate values. Even if a value appears in more than one of the

columns you are combining, the value appears only once in the result column. To include duplicate values in the result column, follow the UNION keyword with ALL.

Scalable SQL can create a result table of combined columns only if the SELECT clauses return comparable columns. If you attempt to combine columns whose data types are not compatible, Scalable SQL returns Status Code 839. Although the columns can be of different specific data types and sizes, their data types must be of the same category. For example, you cannot combine a column that is a string data type (like LSTRING) with a column that is a numeric data type (like MONEY).

Scalable SQL determines the format of the result column based on the characteristics of the columns you combine. The following basic types of combinations are possible:

- Columns of the same data type and size.
- String columns with other string columns.
- Numeric columns with other numeric columns.
- Boolean columns with other boolean columns.
- Date columns with other date columns.
- Time columns with other time columns.

Note: Scalable SQL can apply an edit mask to the result column only if the mask is defined in the first SELECT clause in the union. Scalable SQL ignores an edit mask specified in any SELECT clause other than the first.

Combining Same Type and Size Columns

If the columns you are combining are of the same type and size, the result column has that type and size, too.

If the columns you are combining are of type MONEY, DECIMAL, NUMERIC, NUMERICSA, or NUMERICSTS, they must also have the same number of decimal places. If they do not, the result column's size is determined using the formula discussed in "Determining the Size of the Result Column".

Examples of Combining Same Type and Size Columns

The following examples use the tables in the sample database.

Unions with Distinct Rows

The following statement lists the ID numbers of each student whose last name begins with *M* or who has a 4.0 grade point average. The result table does not include duplicate rows.

```
SELECT ID FROM Person WHERE Last_Name begins with 'M'

UNION SELECT ID FROM Student WHERE Cumulative_GPA
= 4.0;
```

The following statement lists the names of all classes that begin with *P* or that were taught by Professor Vugrinac (Faculty_ID = 193644951). Scalable SQL orders the result table by the first column in the result table (the Name column) and does not include duplicate rows.

```
SELECT Name FROM Class WHERE Faculty_ID =
'193644951'

UNION SELECT Name FROM Class WHERE Name BEGINS
WITH 'P' ORDER BY 1;
```

Union with Duplicate Rows

The following statement also lists the names of all classes that begin with *P* or that were taught by Professor Vugrinac (Faculty_ID = 193644951). However, because the statement includes the ALL keyword, the result table includes duplicate rows.

```
SELECT Name FROM Class WHERE Faculty_ID =
'193644951'

UNION ALL

 SELECT Name FROM Class WHERE Name BEGINS WITH 'P'
ORDER BY 1;
```

Combining String Type Columns

The string data types are divided into two groups: fixed length (CHARACTER, LSTRING, ZSTRING), and variable length (LVAR and NOTE). Scalable SQL combines columns of these data types as follows:

- If all the columns you are combining are of variablelength string types, the result column has the type, size, and mask of the column that contains the longest data item.
- If one of the columns is variable length, the result column has the type of the variablelength column, and it has the size and mask of the column that contains the longest data item.
- If none of the columns are variable length, the result column has the type and size of the longest column.

Combining Numeric Type Columns

The numeric data types are divided into three groups: integer (AUTOINC and INTEGER), decimal (DECIMAL, NUMERIC, NUMERICSA, NUMERICSTS, and MONEY), and float (FLOAT and BFLOAT). Scalable SQL allows you to combine columns of these data types as follows:

- If the data types of all the columns you are combining are of the integer group, the result column has the same type, size, and mask as the largest column.
- If all the columns' data types are of the FLOAT group, the result column has the same type, size, and mask as the largest column.
- If the columns are of different types and one column's data type is of the FLOAT group, the result column has that column's type, size, and mask.
- If the data type of one column is of the DECIMAL group and the data type of another column is of the INTEGER group, the result column is of the same type as the DECIMAL column. The result column's size and mask are based on the sizes of the columns you are combining.
- If all the columns' data types are of the decimal group, the result column's size, type, and mask are determined as described in the following sections, "Determining the Size of the Result Column" and "Determining the Mask of the Result Column".

Determining the Size of the Result Column

In some cases, Scalable SQL must calculate the size of the result column based on the types and sizes of the columns you are combining. This occurs in either of the following situations:

- When you are combining columns whose data types are of the decimal group but whose sizes are different.
- When you are combining a column whose data type is of the decimal group with one whose data type is of the integer group.

The result column must be large enough to contain the largest possible number of digits to the left and right of the decimal point in all columns. In addition, Scalable SQL limits the length of the result column to 19 digits. Scalable SQL uses a formula to determine the exact size of the result column so that it meets these constraints.

In the formula, *precision* is the total number of digits possible in the column, and *scale* is the number of digits to the right of the decimal point. Consequently, the number of digits to the left of the decimal point is *precision minus scale*. The formula uses these variables:

p1	Precision of the first column you are combining.
p2	Precision of the second column you are combining.
p3	Precision of the result column.
s1	Scale of the first column you are combining.
s2	Scale of the second column you are combining.
s3	Scale of the result column.

The formula is as follows:

IF (maximum (*p1 – s1*, *p2 – s2*) + maximum (*s1*, *s2*)) <= 19

p3 = maximum (*p1 – s1*, *p2 – s2*) + maximum (*s1*, *s2*)

s3 = maximum (*s1*, *s2*)

ELSE

p3 = 19

s3 = 19 – maximum (*p1 – s1*, *p2 – s2*)

Using this formula, the result column has a size of (p3/2 + 1), with s3 decimal places.

Determining the Mask of the Result Column

The mask Scalable SQL uses for the result column varies depending on the data types of the columns you are combining and certain characteristics of the result column. For example, assume Scalable SQL combines two columns whose data types are of the decimal group:

- If the result column has the same size and number of decimal places as one of the two columns, the result column has the same type and mask as that column.
- If the result column does not have the same size and number of decimal places as one of the two columns, the result column has the same type and mask as the column that has the largest number of places to the left of the decimal point.

In contrast, if Scalable SQL combines a column whose data type is of the decimal group with a column whose data type is of the integer group, the following rules apply:

- If the number of places to the left of the decimal point in the result column is the same as that for the decimal type column, the result column has the decimal type column's mask.
- If the number of places to the left of the decimal point in the result column is not the same as that for the decimal type column, the result column uses the default mask for its data type.

Combining Boolean Type Columns

The Boolean data types are LOGICAL and BIT. Scalable SQL combines columns of these data types as follows:

- If one column is LOGICAL and one is BIT, the data type of the result column is LOGICAL, and the result column has the same size and mask as the LOGICAL type column.
- If both columns are LOGICAL, the result column has the same size and mask as the larger of the columns you are combining.
- If both columns are BIT, the result column has the same mask as the column returned by the first query of the union.

Combining Date and Time Columns

If one of the columns you are combining is of type DATE or TIME, the other columns must be of the same type and size. Otherwise, Scalable SQL returns a Status Code 839.

Examples of Combining Comparable Columns

The examples in this section use the following tables:

Table 1

Int2 (2 bytes)	Zstring11 (11 bytes)	Zstring15 (15 bytes)	Float8 (8 bytes)
1	zstring11a	zstring15a	1.11000E+000
2	zstring11b	zstring15b	2.22000E+000
3	zstring11c	zstring15c	3.33000E+000

Table 2

Float4 (4 bytes)	Char15 (15 bytes)	Char11 (11 bytes)	Decimal104 (10 bytes)

1.110000E+00	char15a	char11a	4.44400
2.220000E+00	char15b	char11b	5.55500
3.330000E+00	char15c	char11c	6.66600

Union Combining Comparable Data Types

The following union combines several comparable data types:

SELECT * FROM table1

UNION

SELECT * FROM table2

The result table for this union is as follows:

Int2 **float (4 bytes)**	**Zstring11** **character (15 bytes)**	**Zstring15** **zstring (15 bytes)**	**Float8** **float (8 bytes)**
1.000000E+00	**zstring11a**	**zstring15a**	**1.110**
1.110000E+00	**char15a**	**char11a**	**4.444**
2.000000E+00	**zstring11b**	**zstring15b**	**2.220**
2.220000E+00	**char15b**	**char11b**	**5.555**
3.000000E+00	**zstring11c**	**zstring15c**	**3.330**
3.330000E+00	**char15c**	**char11c**	**6.666**

Note: Scalable SQL uses the column names from the first query in the union as the column names in the result table.

The statement combines Table1.Int2 with Table2.Float4, Table1.Zstring11 with Table2.Char15, and so forth. The result table demonstrates the following:

- When combining the Int2 column and the Float4 column, Scalable SQL converts the data from the Int2 column to type FLOAT, size 4.

- When combining the Zstring15 column and the Char11 column, Scalable SQL converts the data from the Char11 column to type ZSTRING, size 15 because the longer string column determines the type and size of the result. For the same reason, Scalable SQL changes the format of data in the Zstring11 column to CHARACTER, size 15.
- When combining the Float8 column and the Decimal104 column, Scalable SQL converts the data from the Decimal104 column to type FLOAT, size 8.
- Scalable SQL displays the results using the default masks for the Float4, Char15, Zstring15, and Float8 columns.

Union Using Defined Masks

The union in this example combines columns and uses a mask for one of the columns:

```
SET MASK table1.FLOAT8 = 'ZZZZ.ZZZ';

SELECT FLOAT8 FROM table1

UNION

SELECT DECIMAL104 FROM table2;
```

The result table combines the data from both tables and displays it using the mask for Table1.FLOAT8:

FLOAT8
1.110
2.220
3.330
4.444
5.555
6.666

Union Showing an Integer-to-Decimal Conversion

The following statement combines Table1.FLOAT8 with Table2.FLOAT4, and Table1.INT2 with Table2.DECIMAL104:

```
SELECT FLOAT8, INT2 FROM table1

UNION SELECT FLOAT4, DECIMAL104 FROM table2
```

The result table is as follows:

FLOAT8	INT2

1.110	1.0000
2.220	2.0000
3.330	3.0000
1.110	4.4440
2.220	5.5550
3.330	6.6660

The result table demonstrates the following:

- When combining the FLOAT8 and FLOAT4 columns, Scalable SQL converts the data from the FLOAT4 column to type FLOAT, size 8.
- When combining the INT2 column and the DECIMAL104 column, Scalable SQL converts the data from the INT2 column to type DECIMAL, size 10, with 4 decimal places.
- Scalable SQL displays the results using the FLOAT8 column's defined mask and the DECIMAL104 column's default mask.

Union Using Temporary Edit Masks

The following union defines two temporary masks:

```
SELECT INT2 [ZZZZ.ZZZ], FLOAT8 FROM table1

UNION SELECT FLOAT4, DECIMAL104 [ZZZZ.ZZZZZ] FROM
table2
```

The result table is as follows:

INT2	**FLOAT8**
1.0000	1.110
2.0000	2.220
3.0000	3.330
4.4440	1.110
5.5550	2.220
6.6660	3.330

This table demonstrates the following:

- When combining the INT2 column and the FLOAT4 column, Scalable SQL converts the data from the INT2 column to type FLOAT, size 4. Scalable SQL displays the results using the temporary mask for INT2 instead of the default mask for FLOAT4.
- Similarly, when combining the FLOAT8 column and the DECIMAL104 column, Scalable SQL converts the data from the DECIMAL104 column to type FLOAT, size 8.
- Scalable SQL displays the result using the FLOAT8 column's defined mask (set in "Union Using Defined Masks") instead of the temporary mask defined for DECIMAL104. The temporary mask for DECIMAL104 does *not* override FLOAT8's defined mask, nor would it override FLOAT8's default mask if FLOAT8 did not have a defined mask. This is because the temporary mask must be defined in the *first* query of the union.

Read-Only Views

Generally, you can update and delete rows in views like those in a table. However, you cannot update or delete rows in *readonly* views. A read-only view is one that contains data from a table that has been opened in readonly mode, or one whose SELECT clause meets certain criteria (for more information about read-only views, refer to the).*Database Design Guide*

*T*he following rules apply to read-only views:

- If the open mode is Read Only due to executing a SET OPENMODE = READONLY statement, then you cannot update the view.
- You cannot update a UNION.
- If an aggregate appears in the selection list, you cannot update the view.

> **Note:** If you specify GROUP BY or HAVING, an aggregate must appear in the selection list.

- If the DISTINCT keyword is present, you cannot update the view.
- An INSERT, UPDATE, or DELETE statement cannot contain a system table. However, a subquery in the statement may contain a system table. Tables in subqueries are not a target for an update in a statement.
- An INSERT, UPDATE, or DELETE statement cannot contain a non-mergeable stored view. For more information about mergeable views, refer to the *Database Design Guide*.
- You cannot perform an operation on a table or column if you do not have the necessary rights.

Scalable SQL allows you to update views with the following characteristics:

- Contains a self join.
- Contains a cartesian product.
- Contains an inequality join condition.

- Contains a correlated subquery.

SET

SET *SQL_variable_name* = < *value_expression* | NULL >

Use the SET statement to initialize or change the value of SQL variables.

The value expression may be a SELECT statement or a computed expression involving constants, operators, and this or other SQL variables. For more information about computed expressions, refer to the *Database Design Guide*.

Example

The following example sets the variable Negative to either 1 or 0, depending on whether vInteger is a positive or negative number.

```
IF (vInteger < 0) THEN SET Negative = '1'

ELSE SET Negative = '0'

END IF;
```

SET CHAR

SET CHAR [table_name.] column_name =

< char_value [, char_value] ... | NULL >

where

char_value ::= < '*single_char*' | *char_range* >

char_range ::= '*single_char*' - '*single_char*'

The SET CHAR statement allows you to define valid input characters for a string column. Unless you specify NULL, Scalable SQL stores the definition in the X$Attrib system table.

When you issue a SET CHAR statement, specify the name of the column for which to define input validation criteria. If the column name is unique in the dictionary, you can specify just the name. If the column name is used in more than one table definition, be sure to qualify the column name in the following way:

table_name.column_name

Replace *value* with the string to define as valid input. You can specify a range or specific characters. Enclose each value in single quotation marks; if you are specifying a range, enclose in quotation marks both characters that define the range, and include a hyphen between them. If you specify multiple values, separate them with commas.

To remove a validation criteria from the dictionary, set the validation criteria for that column to NULL.

Example

The following statement specifies that only uppercase and lowercase alphabetic characters, the digits 0 through 9, and the pound sign (#) are valid as input to the Address column:

SET CHAR Address = 'A'-'Z', 'a'-'z', '0'-'9', '#';

SET DATAPATH

> **Note:** This statement is provided for support of existing applications. New applications should use database names to provide locations of files. See the *Database Design Guide* for more information about database names.

SET DATAPATH = < '*pathname*' | NULL >

The SET DATAPATH statement is valid only when you log in using a path, and not a named database.

The SET DATAPATH statement specifies the prefix to the file path that is defined for the data files in the dictionary. If the definition of the data files in the dictionary does not contain a complete pathname to the files, use the SET DATAPATH statement to specify a prefix to the file path. You can also specify a list of directories by separating each prefix with a semicolon. Enclose the entire list in quotation marks.

If you set a data path, Scalable SQL sequentially searches through the directory list when it is locating a data file. If you are creating a data file, Scalable SQL creates the file in the first directory listed in the SET DATAPATH statement.

When you specify NULL, Scalable SQL looks for the data files only in the current directory.

The SET DATAPATH statement is in effect only for the current login session. Scalable SQL does not store the new path in the data dictionary.

Examples

The following examples illustrate the use of the SET DATAPATH statement:

```
SET DATAPATH = 'f:\bti\win\demodata';

SET DATAPATH = '\\servername\sys:demodata';
```

SET DATEFORMAT

SET DATEFORMAT [=] <format>

where

format ::= [mdy | myd | dmy | dym | ymd | ydm | **NULL**
| **DEFAULT**]

The SET DATEFORMAT statement allows you to change the default input and display format for DATE values. The default input format is used to interpret DATE constants and DATE data values; the default display format is used to display DATE constants and DATE data values for which there is no mask defined.

> **Note:** The SET DATEFORMAT statement changes the ordering of the three DATE value parts: day, month, and year. This statement does not define masks for date values. For information about defining masks, refer to "SET MASK" .

The format parameter is case-insensitive.

NULL and DEFAULT revert the input format to month, day, year and revert the display format to two-digit month, two-digit day, and two-digit year. (For all other formats, the display format contains a four-digit year.)

All input values can include a two- or four-digit year value, a one- or two-digit month value, and a one- or two-digit day value, regardless of the ordering you specify (even if you do not use the SET DATEFORMAT statement or if you specify NULL or DEFAULT).

Example

To specify that all date values and date constants must indicate the year value, then the month value, and then the day value, issue the following statement:

```
SET DATEFORMAT = ymd;
```

SET DEFAULT

SET DEFAULT [table_name.] column_name = <
literal | NULL >

The SET DEFAULT statement allows you to define the value Scalable SQL uses for a certain column if you insert a row without specifying a value for that column. Unless you specify NULL, Scalable SQL stores the definition in the X$Attrib system table.

The values in SET DEFAULT statements can match either the column's user-defined edit mask or the column's data type default mask, except in the case of types DATE, TIME, and TIMESTAMP. The values must match the column's user-defined edit mask for columns of type DATE, TIME, and TIMESTAMP.

When you issue a SET DEFAULT statement, specify the name of the column for which to define a default value. If the column name is unique in the dictionary, you can specify just the name. If the column name is used in more than one table definition, be sure to qualify the column name in the following way:

table_name.column_name

Replace *value* with the default value you want to define for the column. If you are specifying a default value for a string type column, enclose the value in single quotes.

To cancel a previously defined default value, specify NULL instead of a value.

Example

To direct Scalable SQL to insert the characters TX into the State column if you do not specify another value in an INSERT statement, issue the following statement:

```
SET DEFAULT Person.State = 'TX';
```

SET (global null value)

SET < STRINGNULL = '*single_char*' | < BINARYNULL
| DECIMALNULL >

= numeric_literal >

The SET (global null value) statement allows you to specify a null value for a particular group of data types. You can define the null value for all columns of certain data types. Scalable SQL does not store the setting in the dictionary; the null value you specify is in effect only for the current login session.

Depending on the keyword you include in a SET (global null value) statement, Scalable SQL sets the null value as follows:

- SET STRINGNULL defines a null value for all columns of data type CHARACTER.
- SET DECIMALNULL defines a null value for all columns of data type CURRENCY, DECIMAL, MONEY, NUMERIC, NUMERICSA, or NUMERICSTS.
- SET BINARYNULL defines a null value for columns of all other data types (AUTOINC, BFLOAT, BIT, DATE, FLOAT, INTEGER, LOGICAL, LSTRING, TIME, TIMESTAMP, UNSIGNED, and ZSTRING).

Note: A SET (global null value) statement has no effect on NOTE and LVAR type columns.

To specify a value with a SET STRINGNULL statement, enclose the value in single quotes. However, do not use quotation marks around the values in SET DECIMALNULL and SET BINARYNULL statements.

Example

The following statement directs Scalable SQL to insert the asterisk character into each byte of any string column for which you do not specify a value in an INSERT statement and for which you have not defined a default value:

```
SET STRINGNULL = '*';
```

SET ISOLATION

SET ISOLATION = < CS | EX >

The SET ISOLATION statement allows you to control a session's *isolation level*. The isolation level is the level of MicroKernel data locking that Scalable SQL employs to provide isolation from changes to the data during a transaction. In other words, it is the degree to which a user's transaction affects the ability of other users to read and update data in the same table.

Scalable SQL does not store the isolation level setting in the dictionary; the isolation level you specify is in effect only for the current login session.

You can choose either of the following isolation levels:

- CS – Cursor Stability
  Scalable SQL causes the MicroKernel Database Engine to lock individual pages during a transaction instead of locking the entire file, thus allowing concurrent updates. When the MicroKernel locks a data page during a transaction, users can still read and update the other pages in the file. The MicroKernel locks the pages as follows:

  - Each page from which you read data is locked until your next read operation or until you end the transaction, whichever occurs first.
  - If you update, delete, or insert data, each affected page is locked until the end of the transaction (regardless of subsequent read operations).

  Users within a transaction cannot update or read data from any locked page until the transaction that locked the page ends. This is because the user's transaction attempts to lock pages that are already locked. Users outside a transaction can read data from the locked pages, but they cannot write data to them.

- EX – Exclusive Access
  Scalable SQL causes the MicroKernel Database Engine to use the entire data file as the locking unit for transactions. Users within a transaction cannot update or read data from any locked file until the transaction that locked the file ends. Users outside a transaction can read data from the locked files, but they cannot write data to them.

If you do not issue a SET ISOLATION statement, Scalable SQL uses the default isolation level that was defined when Scalable SQL was loaded.

Note: If you change the isolation level during a transaction, the change does not take effect until the transaction ends.

Example

The following statement instructs Scalable SQL to use cursor stability:

```
SET ISOLATION = cs;
```

SET MASK

SET MASK [table_name.] column_name = < edit_mask
| NULL >

The SET MASK statement allows you to define the display format of the specified column. You can specify either a mask or, for string columns, the display length of the column. Unless you specify NULL, Scalable SQL stores the definition in the X$Attrib system table.

> **Note:** You cannot define a display format for columns of data type NOTE or LVAR. These data types are variable-length, user-defined types and their internal format is unknown.

When you issue a SET MASK statement, specify the name of the column for which to define a display format. If the column name is unique in the dictionary, you can specify just the name. If the column name is used in more than one table definition, qualify the column name.

To define a mask for the column, replace *edit_mask* with the desired mask. For information about the correct method of specifying masks for the various data types, see the *Database Design Guide*.

To specify the display length of a string column, replace *edit_mask* with *xn*, where *n* is the desired display length. For example, the following statement limits the display length of the City column to 10 characters:

```
SET MASK Person.City = 'x10';
```

If a column value is longer than the display length you specify, Scalable SQL truncates the value from the right. To cancel a previously defined display format, specify NULL instead of a mask or display length.

Example

The following example defines a mask for the Amount Paid column in the billing table of the sample database:

```
SET MASK Billing.Amount_Paid = '$Z,ZZZ.99';
```

SET OPENMODE

SET OPENMODE =

< NORMAL | ACCELERATED | READONLY | VERIFY
| EXCLUSIVE >

The SET OPENMODE statement allows you to specify the file open mode for the data files associated with the tables in your database. The open modes you can specify through Scalable SQL correspond directly to the MicroKernel Database Engine open modes.

Scalable SQL does not store the file open mode setting in the dictionary; the open mode you specify is in effect only for the current login session.

Note: If you change the file open mode during a transaction, the change does not take effect until the next file is opened.

Normal Mode

Including the NORMAL keyword in the statement sets the data files' open mode to Normal, which is the default open mode. Using a server-based MicroKernel Database Engine, Normal mode allows shared read/write access to data files. In Normal mode, the MicroKernel performs its standard integrity processing when it updates the data files.

Accelerated Mode

Including the ACCELERATED keyword in the statement sets the data files' open mode to Accelerated.

Using a post-v5.*x* and pre-v7.0 MicroKernel Database Engine, Accelerated mode is equivalent to Normal mode, except that opening a data file in Accelerated mode with the server-based MicroKernel cancels the effect of flagging a file as transactional.

With the v7.*x* MicroKernel Database Engine, Accelerated mode provides improved response time over Normal mode when updating data files. However, you may lose durability if a crash occurs while you have a file open in Accelerated mode. In this event, changes to a single data file are durable, but the file may then be out of sync with other files in the database.

ReadOnly Mode

Including the READONLY keyword in the statement sets the data files' open mode to ReadOnly, which means Scalable SQL can read data files but cannot write to them. After you set this open mode, you cannot issue an INSERT, UPDATE, or DELETE statement for any of the tables in your database until you reset the open mode to Normal, Accelerated, Verify, or Exclusive.

Verify Mode

Including the VERIFY keyword in the statement sets the data files' open mode to Verify. This enables operating system verification during each write operation to the data files. After each write operation, the operating system rereads the data to ensure that it has been recorded correctly on the disk. The operating system does not support verification on a network disk; files on a network disk are opened in Normal mode.

Exclusive Mode

Including the EXCLUSIVE keyword in the statement sets the data files' open mode to Exclusive, which means access to the data files is restricted to only one task. If a file is opened in Exclusive mode, no other task can open that file until the file is closed.

Example

The following statement sets the file open mode to Normal:

```
SET OPENMODE = NORMAL;
```

SET OWNER

SET OWNER = owner [, owner] ...

where

owner ::= ['] *owner_name* [']

The SET OWNER statement allows you to specify a list of owner names for data files that have owner names assigned through the MicroKernel Database Engine. Scalable SQL passes the owner names to the MicroKernel, enabling it to open the data files.

If you have data files that were created with file owner names, you may be required to specify the owner names to Scalable SQL before accessing the files, depending on the type of access required and the owner access restriction specified for the files. (For more information, refer to "Specifying the Owner Access Restriction")

You can specify as many as eight owner names with a SET OWNER statement. If an owner name begins with a nonalphabetic character, you must enclose the name in single quotes (' ').

A SET OWNER statement cancels the effect of any previous SET OWNER statement. Also, a SET OWNER statement provides owner names for the current session only. The next time you log in, you must specify the owner names again.

Note: If you set the owner name during a transaction, the change does not take effect until the next file is opened.

Example

The following statement specifies three data file owner names to Scalable SQL:

```
SET OWNER = George, Marie, '123xx';
```

SET RANGE

SET RANGE [table_name.] column_name =

< value_range [, value_range] ... | NULL >

where

value_range ::= *literal - literal*

The SET RANGE statement allows you to define one or more acceptable ranges for the specified column. Unless you specify NULL, Scalable SQL stores the definition in the X$Attrib system table.

When you issue a SET RANGE statement, specify the name of the column for which to define acceptable ranges. If the column name is unique in the dictionary, you can specify just the name. If more than one table definition uses the column name, be sure to qualify the column name in the following way:

table_name.column_name

Replace *value* with a range specification consisting of two values separated by a dash (–). Surround the dash with spaces to distinguish it from a negative sign. Separate multiple range specifications with commas.

The values in SET RANGE statements can match either the column's user-defined edit mask or the column's data type default mask, except in the case of types DATE, TIME, and TIMESTAMP. The values must match the column's user-defined edit mask for columns of type DATE, TIME, and TIMESTAMP.

To cancel a previous range definition, specify NULL instead of a range.

Example

The following statement sets ranges for the AcctNum column:

SET RANGE Acct_Num = 1000 - 1999, 5000 - 5999;

This statement tells Scalable SQL to accept account number values from 1,000 through 1,999 and from 5,000 through 5,999 only.

SET SECURITY

SET SECURITY = < master_password | NULL >

The SET SECURITY statement allows you to enable and disable security for the database to which you are currently logged in.

To enable security for a dictionary, you must first log in to that dictionary. Then, issue a SET SECURITY statement specifying a master password for the dictionary. Scalable SQL creates a user named Master, gives the master user full rights to the dictionary, and assigns the password you specify to the master user. (Both the master password and the Master username are case-sensitive.) Once the master user exists, no other users have access to the dictionary until they are granted rights. For more information about granting rights, see "GRANT (access rights)", "GRANT CREATETAB", and "GRANT LOGIN".

To disable security for a dictionary, specify NULL instead of a password.

Example

The following statement enables security for a dictionary and defines Secure as the master password:

```
SET SECURITY = Secure;
```

The following statement disables security:

```
SET SECURITY = NULL;
```

SET VALUES

SET VALUES [table_name.] column_name =

< literal [, literal] ... | NULL >

The SET VALUES statement allows you to specify all acceptable values for a column. Unless you specify NULL, Scalable SQL stores the definition in the X$Attrib system table.

When you issue a SET VALUES statement, specify the name of the column for which to define acceptable values. If the column name is unique in the dictionary, you can specify just the name. If more than one table definition uses the column name, be sure to qualify the column name in the following way:

table_name.column_name

Replace *value* with the acceptable value, enclosed in single quotes. If you specify multiple values, separate them with commas.

The values in SET VALUES statements can match either the column's user-defined edit mask or the column's data type default mask, except in the case of types DATE, TIME, and TIMESTAMP. The values must match the column's user-defined edit mask for columns of type DATE, TIME, and TIMESTAMP.

To cancel a previously acceptable value definition, specify NULL instead of a value.

Example

The following statement tells Scalable SQL to accept only the two-character abbreviation for Texas and Louisiana in the State column:

SET VALUES State = 'TX','LA';

SIGNAL

SIGNAL *signal_value*

where

signal_value ::= < *condition_name*

| SQLSTATE [VALUE] *character_string_literal*
>

The SIGNAL statement allows you to signal an exception condition or a completion condition other than successful completion.

Rules for Using the SIGNAL Statement

The following rules apply to using the SIGNAL statement:

- You must declare the condition name in the scope of this statement with a DECLARE CONDITION statement.
- The SIGNAL statement raises the specified exception condition or completion condition by setting the value of SQLSTATE to the specified character string literal or to the value that is associated with the condition name.

Example

The following examples use the SIGNAL statement to signal a SQLSTATE condition and a declared a condition, respectively.

```
SIGNAL '01111'
```

```
SIGNAL Cond1
```

For more examples using the SIGNAL statement, refer to the CREATE PROCEDURE example on page 2-37 and the DECLARE HANDLER example on page 2-69. Both examples use the SIGNAL statement to signal declared conditions.

START TRANSACTION

START TRANSACTION

The START TRANSACTION statement allows you to specify the beginning of a transaction. All subsequent statements you issue are part of the transaction until you end it with either a COMMIT WORK or a ROLLBACK WORK statement. Scalable SQL executes either all the statements or none of them.

After issuing a START TRANSACTION statement, issue the statements to execute as a logical unit.

Note: If you issue a SET ISOLATION statement during a transaction, the change does not take effect until the transaction ends.

Statements that change the dictionary files are not affected by transactions. These include the following types of statements: ALTER TABLE; CREATE TABLE; CREATE INDEX; CREATE PROCEDURE; CREATE VIEW; and statements containing the DROP, GRANT, or REVOKE keywords. If Scalable SQL executes one of these types of statements within a transaction, you cannot roll back the changes from that statement.

Note: If you start a transaction and then log out of the dictionary before issuing a COMMIT WORK or ROLLBACK WORK statement, Scalable SQL automatically issues a ROLLBACK WORK statement before completing the logout.

Examples

To signify the beginning of a logical transaction, issue the following statement:

```
START TRANSACTION;
```

The following example begins a transaction which updates the Amount_Owed column in the Billing table. This work is committed; the AND CHAIN clause begins another transaction that updates the log in the Billing table. The final COMMIT WORK statement ends the second transaction.

```
START TRANSACTION;

UPDATE Billing B

SET Amount_Owed = Amount_Owed - Amount_Paid

WHERE Student_ID IN

 (SELECT DISTINCT E.Student_ID

 FROM Enrolls E, Billing B

 WHERE E.Student_ID = B.Student_ID);

COMMIT WORK AND CHAIN;

UPDATE Billing B
```

```
SET Amount_Paid = 0

WHERE Student_ID IN

 (SELECT DISTINCT E.Student_ID

 FROM Enrolls E, Billing B

 WHERE E.Student_ID = B.Student_ID;

COMMIT WORK;
```

UPDATE

UPDATE table_list

SET set_clause [, set_clause] ...

[WHERE Boolean_value_expression]

where

table_list ::= table_reference [, *table_reference*]
...

table_reference ::= < *table_name* | view_name > [
alias_name]

set_clause ::= column_reference = expression

column_reference ::= [column_qualifier.]
column_name

column_qualifier ::= < table_name | view_name |
alias_name >

Boolean_value_expression ::= see page 2-121

The UPDATE statement allows you to modify column values in a database. Scalable SQL modifies only the columns you list in the SET clause.

If you do not want to change all the values in the specified column, you can include a WHERE clause (see page 2-121 for the valid syntax) to define which rows in the table to modify. When you issue an UPDATE statement, Scalable SQL applies any input validation that you specified in the dictionary for the table. It rejects statements containing invalid data.

If you define referential integrity (RI) for the database and you update the value of a foreign key, Scalable SQL verifies that the parent table contains a corresponding primary key whose value matches the new foreign key value. If the parent table does not contain a corresponding primary key, the update fails.

You can use either of the following methods to specify the values to use in the update:

- Specify the data values directly in the SET clause.
- Extract the data values from the database by including a SELECT clause in the SET clause.

Specifying Data Values Directly

To specify the new data values directly, use the SET clause to define the name of each column to modify and the new value to assign to the column. The value you specify can be a constant or an expression. For example, the following statement changes the credit hours for Economics 305 in the Course table from 3 to 4:

UPDATE Course SET Credit_Hours = 4 WHERE Name = 'ECO
305'

In the following statement, an expression specifies the new data value. The statement sets the start date in the Class table to 10 days after January 1, 1996.

```
UPDATE Class SET Start_Date = (01/01/96 + 10)
```

If you specify the data values directly, you can use an UPDATE statement to update columns in more than one table. You may want to assign aliases to the table names and use the aliases when you refer to the columns in the SET clause. For more information about specifying constant values, refer to "The Role of Data Types, Defined Masks, and Default Input Formats"

When you use an UPDATE statement with an application that supports substitution variables, you can use variables to specify the values. For example, the following statement uses substitution variables to determine both the course to update and the new credit hours:

```
UPDATE Course SET Credit_Hours = @credithours

WHERE Course = @course;
```

For more information about using substitution variables, see the *Database Design Guide*.

Extracting Data Values with a SELECT Clause

Including a SELECT clause in an UPDATE statement allows you to update data based on a column value in another table. Instead of specifying a new data value directly in the SET clause, you can use the SELECT clause to retrieve the value from the database.

A SELECT clause included in an UPDATE statement usually contains a WHERE clause that references the table you are updating.

When you use the results of a SELECT statement to update a column, the data types of the columns in the view defined by the SELECT statement must be compatible with the data types of the columns you are updating. (These same rules also apply to the use of a SELECT clause with an INSERT statement.) For more information about data type compatibility rules, refer to "Data Type Compatibility" .

Example

The following statement updates the address for a person in the Person table:

```
UPDATE Person p

SET p.Street = '123 Lamar',

 p.zip = '78758',

 p.phone = 5123334444

WHERE p.ID = 131542520;
```

UPDATE: positioned

UPDATE [*table_list*]

SET *set_clause* [, *set_clause*] ...

WHERE CURRENT OF *cursor_name*

where

table_list ::= table_reference [, table_reference]
...

table_reference ::= < table_name | view_name > [
alias_name]

set_clause ::= *column_reference = value*

column_reference ::= [*column_qualifier.*]
column_name

column_qualifier ::= < *table_name* | *view_name* |
alias_name >

The Positioned UPDATE statement updates the current row of a view associated with a SQL cursor.

Rules for Using the Positioned UPDATE Statement

The following rules apply to using the Positioned UPDATE statement.

- The specified cursor must be an updatable cursor, and it must be open. For more information about declaring cursors, see “DECLARE CURSOR”.
- Each specified column name must identify a column in one of the tables in the view declared by the specified cursor.
- If the declaration of the specified cursor contains a sort specification, then no column name in the Positioned UPDATE statement can identify a column used in the sort specification.
- The same column name must not appear more than once.
- If you specify a query expression in a value expression, the table defined by the query expression must have a degree and cardinality of 1 or 0. If the cardinality is 0, the column is set to NULL.
- The table reference clause is optional; the underlying tables to be updated are specified by the declaration of the cursor.
- You must establish a valid position with the FETCH statement before executing a positioned DELETE. If a cursor is not positioned to a row, Scalable SQL returns Status Code 8 (invalid positioning).
- All concurrency controls and rules apply to positioned DELETE statements, including isolation levels, locking, and passive control.

- All data constraints are enforced when the columns are updated.
- If the expression used to determine the new value refers to a column from the table being updated, the value of the column before the update is used. This allows SET clauses such as the following:

  SET IntColumn = IntColumn + 1
- The value of each specified column in the object row is replaced by the result of the evaluation of the associated value expression.
- The cursor remains positioned on the current row, even if an error has occurred during the execution of the statement.

Example

The following sequence of statements provide the setting for the Positioned UPDATE statement. The required statements for a positioned UPDATE are DECLARE CURSOR, OPEN CURSOR, and FETCH FROM *cursorname*.

The Positioned UPDATE statement in this example updates the name of the course HIS 305 to HIS 306.

```
DECLARE CourseName CHAR(7) = 'HIS 305';

DECLARE OldName CHAR(7);

DECLARE cursor1 CURSOR

FOR SELECT Name

FROM Course

WHERE Name = CourseName;

OPEN cursor1;

FETCH NEXT FROM cursor1 INTO OldName;

UPDATE SET name = 'HIS 306'

WHERE CURRENT OF cursor1;
```

WHILE

[*beginning_label* :]

WHILE *Boolean_value_expression* DO

SQL_statement_list

END WHILE [*ending_label*]

where

SQL_statement_list ::= { *SQL_statement* ; } ...

Boolean_value_expression ::= see page 2-121

A WHILE statement repeats the execution of a block of statements while a specified condition is true.

If a WHILE statement has a beginning label, it is called a labeled WHILE statement. If you specify an ending label, it must be identical to the beginning label. WHILE statements can appear only in the body of stored procedures and triggers.

The following occurs in a WHILE statement:

- The Boolean value expression is evaluated.
- If the Boolean value expression is true, the SQL statement list executes, and if each statement in the SQL statement list executes without error and no LEAVE statement is encountered, the WHILE statement is repeated until the Boolean expression is false.
- If the Boolean value expression is false or unknown, the WHILE statement is terminated.

Examples

The following example increments the variable vInteger by 1 until it reaches a value of 10, when the loop is ended.

```
WHILE (vInteger < 10) DO

SET vInteger = vInteger + 1;


END WHILE
```

The following example returns the total capacity of the largest rooms on campus in the university database. If NumRooms equals 10, then the result is the capacity of the 10 largest rooms.

```
CREATE PROCEDURE LargeRooms

(IN NumRooms INT(4), OUT TotalCapacity INT(4));

BEGIN


DECLARE counter INT(2) = 0;
```

```
DECLARE CurrentCapacity INT(4) = 0;

DECLARE cRooms CURSOR
 FOR SELECT Capacity
 FROM Room
 ORDER BY Capacity DESC;

OPEN cRooms;

SET TotalCapacity = 0;

FETCH_LOOP:
 WHILE (counter < NumRooms) DO
   FETCH NEXT FROM cRooms INTO CurrentCapacity;

   IF (SQLSTATE = '02000') THEN
    LEAVE FETCH_LOOP;
   END IF;

   SET counter = counter + 1;
   SET TotalCapacity = TotalCapacity +
    CurrentCapacity;

 END WHILE;

CLOSE cRooms;
END
```

Data Types

Scalable SQL uses several data types that are divided into two categories: fixed-length and variable-length. All the Scalable SQL data types are fixed-length except NOTE and LVAR, which are variable length.

- *Fixed-length* data types have a designated stored length that does not vary from row to row within a table. Each table must contain at least one column that has a fixed-length data type and the fixed-length column sizes must combine to at least 4 bytes. See "FixedLength Data Types".

- *Variable-length* data types have a stored length that can vary from row to row in a table, and are limited only by the amount of data the MicroKernel can store. A table or view cannot contain more than one column that has a variable-length data type. See "VariableLength Data Types".

You must specify a data type for a column anytime you create or alter a column definition. Following are the types of statements with which you can create or alter column definitions:

- CREATE TABLE
- ALTER TABLE

Table A-1 lists the code, default display mask, and default input format for each data type. For more information about masks, refer to the *Programmer's Guide*.

Scalable SQL uses the data type codes (rather than the keywords) internally when you create a table. These codes also appear in the X$Column system table, and you can use them to determine the data type of a particular column.

Table A-1
Data Types, Codes, Default Masks, and Default Input Formats

Data Type	Code	Default Display Mask	Default Input Formats
AUTOINC	15	–ZZZZZ**1	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
BFLOAT	9	–Z.ZZZZZZE+ZZ	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
BIT	16	true–false	true–false 1–0
CHARACTER	0	none	none

CURRENCY	19	–$ZZZZZZZZZZZZZZZ.ZZZZ	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
DATE	3	mm/dd/yy	mm/dd/yy mm/dd/yyyy mm/d/yy mm/d/yyyy m/dd/yy m/dd/yyyy m/d/yy m/d/yyyy yyyy-mm-dd
DECIMAL	5	–ZZZZZZZZZZZ.ZZZ**2	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
FLOAT	2	–Z.ZZZZZZE+ZZ	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
INTEGER	1	–ZZZZZ1	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
LOGICAL	7	true–false	true–false 1–0
LSTRING	10	none	none
LVAR	13	none	none

MONEY	6	–$ZZZZZZZZZ.ZZ1	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
NOTE	12	none	none
NUMERIC	8	–ZZZZZZ.ZZZ2	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
NUMERICSA	18	–ZZZZZZ.ZZZ2	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
NUMERICSTS	17	–ZZZZZ.ZZZ2	[+-]$<number>[.<digit><digit>] [+-]<number>[.<number>] [+-]<number>[. [<number>]]e[+-]<number>
TIME	4	hh:mm:ss	hh:mm:ss hh:mm:ss:uu
TIMESTAMP	20	yyyy-mm-ddBhh:tt:ss[.fffffff]**3	yyyy-mm-ddBhh:tt:ss[.fffffff]
UNSIGNED	14	ZZZZZ1	[+]$<number>[.<digit><digit>] [+]<number>[.<number>] [+]<number>[. [<number>]]e[+-]<number>
ZSTRING	11	none	none

***1*The number of Zs in the mask depends on the size of the value.
***2*The number of Zs in the mask depends on the size and scale of the value.
***3*The number of fs in the mask depends on the precision of the value.

Table A-2 lists the default length, valid length, and valid value range for each data type.

Table A-2
Data Type Lengths and Ranges

Data Type Keyword	Default Length (in bytes)	Valid Length (in bytes)	Valid Value Range
AUTOINC	2	2	-32768 – +32767
4	-2147483648 – 2147483647		
BFLOAT	4	4	±(5.8774718E-39 – 1.70141173E+38)
8	±(5.8774718E-39 – 1.70141183E+38)		
BIT	1	1	0 or 1
CHARACTER	1	1–255	N/A
CURRENCY	8	8	-922337203685477.5808 – 922337203685477.5807
DATE	4	4	01-01-0001 – 12-31-9999
DECIMAL	6, 0	1–10	Depends on the length and number of decimal places.
FLOAT	4	4	±(1.17549E-38 – 3.402823E+38)
8	±(2.2250738585072E-380 – 1.79769313486232E+308)		
INTEGER	2	1	0 – 255
2	-32768 – 32767		
4	-2147483648 – 2147483647		
8	-9223372036854775808 – 9223372036854775807		
LOGICAL	2	1, 2	0 or non-zero

LSTRING	2	2–255	N/A
LVAR	5	5–32761	N/A
MONEY	6, 2	1–10	Depends on the length and number of decimal places.
NOTE	2	2–32761	N/A
NUMERIC	6, 0	1–15	Depends on the length and number of decimal places.
NUMERICSA	6, 0	1–15	Depends on the length and number of decimal places.
NUMERICSTS	6,0	2–15	Depends on the length and number of decimal places.
TIME	4	4	00:00:00:00 – 23:59:59:99
TIMESTAMP	8	8	0001-01-01 00:00:00.0000000 – 9999-12-31 23:59:59.9999999 UTC
UNSIGNED	2	1	0 – 255
2	0 – 65535		
4	0 – 4294967295		
8	0 – 18446744073709551615		
ZSTRING	2	2–255	N/A

FixedLength Data Types

The stored length of a fixedlength data type does not vary from row to row within a table. The Scalable SQLfixedlength data types correspond to the most useful data types that many programming languages recognize. The following sections discuss each fScalable SQLixedlength data type.

AUTOINC

The AUTOINC data type represents a special form of signed 2 or 4byte integers. If you specify an AUTOINC column value of binary zeros when inserting a row, Scalable SQLreplaces the specified value by automatically incrementing the highest existing value in the column and using the result value in the row you are inserting. Specify a nonzero value only when replacing or updating a row within the existing order.

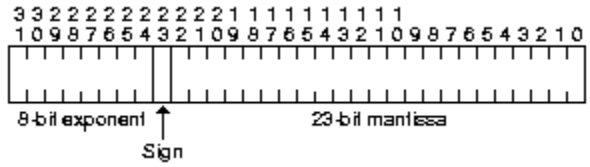
To automatically increment values in an AUTOINC column, define the column as a unique index. An autoincrement index can only be part of a segmented index if the index number of the autoincrement index is less than the index number of the segmented index. If you do not define the column as an index, Scalable SQL treats it as an INTEGER column.

BFLOAT

The BFLOAT data type represents either a single-precision (4byte) or a double-precision (8byte) real number stored in a format that the Microsoft BASIC programming language defines. A single-precision real number is stored with a 23bit mantissa, an 8bit exponent biased by 128, and a sign bit.

Note: This Microsoft Binary Format (MBF) is no longer supported by Microsoft Visual Basic; it was supported in older versions of Quick Basic only.

The internal layout for a 4byte BFLOAT column is as follows:



The representation of an 8-byte BFLOAT column is the same as that for a 4byte BFLOAT column except the mantissa is 55 bits instead of 23 bits. The least significant 32 bits are stored in the first 4 bytes.

Note: In the same way that some values (such as 1/3) cannot be expressed accurately in decimal terms, some values (such as 1/10) cannot be stored accurately in FLOAT or BFLOAT format. Therefore, when storing values that represent precise numbers (such as monetary values), use the CURRENCY, DECIMAL, MONEY, NUMERIC, NUMERICSA, NUMERICSTS, or UNSIGNED data types instead. Restrictions using float values may be more effective using a range (such as column >= 1.501 and column <= 1.502) rather than an equality (such as column = 1.501).

BIT

The BIT data type is a 1-byte value that allows Scalable SQL to compress logical values into bit flags. The column

values of adjacent BIT columns are packed into as few bytes as possible; up to eight values can be stored in a single byte. However, if two BIT columns are separated by one or more non-bit columns, the BIT values cannot be stored in the same byte.

For example, if a table contains two adjacent BIT columns, their values are packed into a single byte. If a third BIT column in the table is preceded by a non-bit column, the value of the third BIT column cannot be stored in the byte containing the other two BIT column values.

Note: A column of the BIT data type may not be defined as an index or index segment.

CHARACTER

The CHARACTER data type in Scalable SQL represents a sequence of characters ordered from left to right, of fixed length, and does not have a terminator character. Each character is represented in ASCII format in a single byte. Scalable SQL pads CHARACTER values with blanks to the defined length of the column.

In comparisons between CHARACTER values and constants, Scalable SQL pads constants with blanks to the length of the value. Trailing blanks in a CHARACTER value are considered significant for comparisons in restriction clauses and expressions and when the LENGTH function returns the length of a string column. Use the LTRIM, RTRIM, or SUBSTRING function to ensure that the lengths of strings you compare are the same.

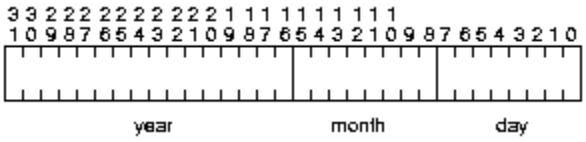
CURRENCY

The CURRENCY data type represents an 8-byte signed quantity, sorted and stored in Intel binary integer format; therefore, its internal representation is the same as an 8 byte INTEGER data type. The CURRENCY data type has an implied four digit scale of decimal places, which represents the fractional component of the currency data value.

DATE

Column values of data type DATE are stored internally as 4byte values in Intel integer format. Day and month values are each stored in 1byte binary format. Year values are stored as 2byte binary numbers. The day is stored in the first byte, the month in the second byte, and the year in the two bytes following the month.

The internal layout for a DATE column is as follows:

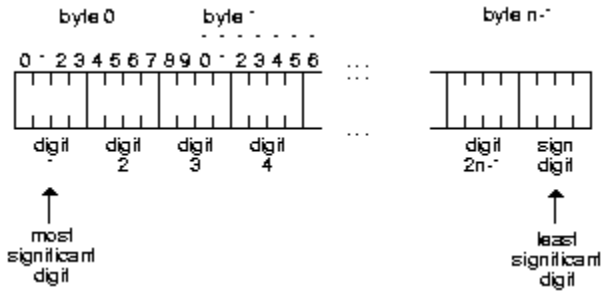


DECIMAL

Column values of data type DECIMAL are signed or unsigned real numbers stored internally as packed decimal numbers, with two decimal digits per byte. This format is consistent with the COMP3 data type in ANSI74 standard COBOL programming language.

For DECIMAL values, each digit takes one half byte of storage and the sign takes one half byte. The internal size is M/2 + 1, where M is the number of significant displayable digits. Therefore, any DECIMAL value that has an even number of displayable digits has an extra one half byte of internal storage.

The internal representation for an *n*-byte DECIMAL column is as follows. (The high-order bit is on the left side.)



The sign digit is either C or F for positive numbers and D for negative numbers. The decimal point is not stored with the data; its location in the column value is fixed according to the column definition in the dictionary.
Example

The DECIMAL column value 12345678.1234 requires a minimum of 7 bytes (12/2 + 1). The internal storage format for the value is as follows:

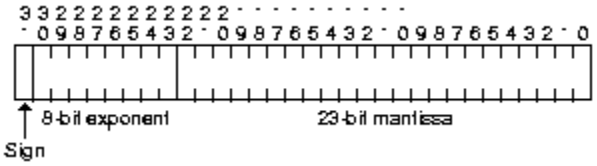
Byte	0	1	2	3	4	5	6
Internal Value (Hexidecimal)	01	23	45	67	81	23	4F

The DECIMAL column value 123456789.1234 also requires a minimum of 7 bytes (13/2 + 1). The internal storage format for the value is as follows:

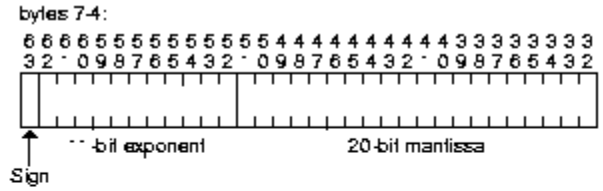
Byte	0	1	2	3	4	5	6
Internal Value (Hexidecimal)	12	34	56	78	91	23	4F

FLOAT

The FLOAT data type is consistent with the IEEE standard for single and double-precision real numbers. The internal format for a 4byte FLOAT consists of a sign bit, an 8bit exponent biased by 127, and a 23bit mantissa. The internal layout for a 4byte FLOAT value is as follows:



A FLOAT data type column with 8 bytes has a sign bit, an 11bit exponent biased by 1023, and a 52bit mantissa. The internal layout for an 8byte FLOAT value is as follows:



Note: In the same way that some values (such as 1/3) cannot be expressed accurately in decimal terms, some values (such as 1/10) cannot be stored accurately in FLOAT or BFLOAT format. Therefore, when storing values that represent precise numbers (such as monetary values), use the CURRENCY,

DECIMAL, MONEY, NUMERIC, NUMERICSA, NUMERICSTS, or UNSIGNED data types instead. Restrictions using float values may be more effective using a range (such as column >= 1.501 and column <= 1.502) rather than an equality (such as column = 1.501).

INTEGER

The INTEGER data type represents a signed or unsigned whole number. Internally, INTEGER values are stored in Intel binary integer format. The 1 byte INTEGER uses the character extended data type and the 2, 4, and 8 byte INTEGER use the unsigned extended data type when indexed.

INTEGER columns store values in the following ranges:

Length in Bytes	Value Ranges
1	0 – 255
2	-32768 – 32767
4	-2147483648 – 2147483647
8	-9223372036854775808 – 9223372036854775807

LOGICAL

The LOGICAL data type is stored as a 1 or 2 byte binary value representing either the logical value true or the logical value false. In an INSERT statement, a value of 1 or TRUE are acceptable input values for a true value. A value of 0 or FALSE are acceptable input values for a false value. These input values are valid when the default masks are set for this LOGICAL column. When a user-defined mask is set on a LOGICAL column, the insert values must match the mask values.

LSTRING

The LSTRING data type in Scalable SQL corresponds to a Pascal string. It has the same characteristics as the CHARACTER data type except the first byte of the string contains the binary representation of the string's length. Bytes beyond the specified end of the string are undefined.

An LSTRING value may contain significant blanks at the end of the string. The length byte indicates the true length.

Example

For an 8-byte LSTRING column, the value Hello is stored as follows:

MONEY

The data type MONEY has the same characteristics as the DECIMAL data type except the number of decimal digits

is fixed at two. The internal size requirements are the same as for the DECIMAL data type. By default, Scalable SQL displays MONEY column values with a dollar sign, which increases the display width by one.

NUMERIC

The total display size required for a NUMERIC column value is equal to the maximum number of significant digits, plus 1 for a decimal point and 1 for a possible sign. Column values of the NUMERIC data type are signed or unsigned real numbers stored internally as ASCII strings that are right justified and padded with leading zeros. Each digit occupies 1 byte internally. The rightmost byte of the number can include an embedded sign, which does not require any additional space. Thus, the internal size of a NUMERIC value is equal to the number of significant digits.

Table A-3 indicates how the rightmost digit is represented when it contains an embedded sign for positive and negative numbers.

Table A-3
Numeric Embedded Sign Characters

Number	Positive	Negative
1	A	J
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	G	P
8	H	Q
9	I	R
0	{	}

For positive numbers, you can represent the rightmost digit with either 1 through 0 or A through {. When Scalable SQL returns positive numbers, the rightmost digit is always represented by 1 through 0. Scalable SQL accepts input from either range.

Example 1

The NUMERIC column value 12345678.4567 requires a minimum of 12 bytes. The storage format is as follows:

Byte	0	-	2	3	4	5	6	7	8	9	-0	--
Internal Value (Hexidecimal)	3-	32	33	34	35	36	37	38	34	35	36	47
Display Value	-	2	3	4	5	6	7	8	4	5	6	7

The value 47 in the last byte represents the embedded sign character G, a display value of 7 and a positive sign for the column value.
Example 2

The NUMERIC column value –12345678.4567 also has an internal size of 12 bytes. The storage format is as follows:

Byte	0	-	2	3	4	5	6	7	8	9	-0	--
Internal Value (Hexidecimal)	3-	32	33	34	35	36	37	38	34	35	36	50
Display Value	-	2	3	4	5	6	7	8	4	5	6	7

The value 50 in the last byte represents the embedded sign character P, a display value of 7 and a negative sign for the column value.

NUMERICSA

The total display size required for a NUMERICSA column value is equal to the maximum number of significant digits, plus 1 for a decimal point and 1 for a possible sign. Column values of the NUMERICSA data type are signed or unsigned real numbers stored internally as ASCII strings that are right justified and padded with leading zeros. Each digit occupies 1 byte internally. The rightmost byte of the number can include an embedded sign, which does not require any additional space. Thus, the internal size of a NUMERICSA value is equal to the number of significant digits.

A NUMERICSA column is consistent with a numeric signed ASCII column in COBOL. The NUMERICSA data type is identical to the NUMERIC data type except it uses a different embedded sign for negative values.

Table A-4 indicates how the rightmost digit is represented when it contains an embedded sign for negative numbers.

Table A-4
Numericsa Embedded Sign Characters

Number	Positive	Negative
0	p	p
1	Q	q
2	R	r
3	S	s
4	T	t
5	U	u
6	V	v

7	W	w
8	X	x
9	Y	y

For positive numbers, you can represent the rightmost digit with either 0 through 9 or P through Y. When Scalable SQL returns positive numbers, the rightmost digit is always represented by 1 through 0. Scalable SQL accepts input from either range.

NUMERICSTS

The total display size required for a NUMERICSTS column value is equal to the maximum number of significant digits, plus 1 for a decimal point and 1 for a possible sign. The NUMERICSTS data type is consistent with a numeric sign trailing separate column in COBOL. NUMERICSTS values are signed or unsigned real numbers stored internally as ASCII strings that are right justified and padded with leading zeros. Each digit occupies 1 byte internally.

> **Note:** Unlike the NUMERIC data type, the NUMERICSTS data type does not embed the sign. Instead, the sign is stored in its own byte. Since the sign uses one byte, the number of digits your application can place in a NUMERICSTS column is one less than the length you specify when you create the table.

The sign is stored in the rightmost byte. Thus, the internal size for a NUMERICSTS column value is equal to the number of significant digits, plus one byte to store the sign.

Example

The NUMERICSTS column value 12345678.4567 requires a minimum of 13 bytes. The storage format is as follows:

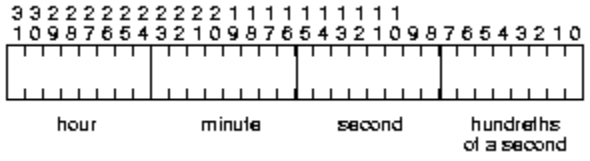
Byte	0	1	2	3	4	5	6	7	8	9	10	11	12
Internal Value (Hexidecimal)	31	32	33	34	35	36	37	38	34	35	36	37	2B
Display Value	1	2	3	4	5	6	7	8	4	5	6	7	

The value 2B in the last byte indicates a positive sign for the column value.

TIME

Column values of data type TIME are stored internally as 4byte values. Hundredths of a second, seconds, minutes, and hours are each stored in 1byte binary format. The hundredths of a second value is stored in the first byte, followed by the second, minute, and hour values.

The internal layout for a TIME column is as follows:



TIMESTAMP

The TIMESTAMP data type represents a time and date value. You use this data type to stamp a record with the time and date of the last update to the record. The column value is stored in 8-byte unsigned values representing septa seconds (10-7 second) since January 1, 0001 in a Gregorian calendar.

You specify the TIMESTAMP data type in the following format using a CREATE TABLE or ALTER TABLE statement:

TIMESTAMP [(*timestamp precision*)]

The timestamp precision is the number of fractional digits in seconds. Valid values of precision are between 0 and 7 inclusive. The following table indicates the valid values of each component of TIMESTAMP:

YEAR	0001 to 9999
MONTH	01 to 12
DAY	01 to 31, constrained by the value of MONTH and YEAR in the Gregorian calendar.
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59

The value is stored internally as a TIMESTAMP MicroKernel key, an 8-byte long field that contains the complete date and time value converted to the fractions of seconds that timestamp precision specifies. For example, it is converted to microseconds when timestamp precision is 6, and it is converted to milliseconds when timestamp precision is 3. You provide the value of a TIMESTAMP in local time and the Scalable SQL engine converts it to Coordinated Universal Time (UTC), formerly Greenwich Mean Time (GMT), before storing it in a MicroKernel record. When you request the value of a TIMESTAMP, Scalable SQL converts it from UTC to local time before returning the data.

Note: It is critical that you set time zone information correctly. If you move across time zones or change time zone information, the returned data will change when it is converted from UTC to local time. The local time/UTC conversions occur in Scalable SQL using the time zone information where the Scalable SQL engine is running. The time zone information for sessions that are in different time zones than the Scalable SQL engine are not used in the local time/UTC conversions.

UNSIGNED

The UNSIGNED data type represents an unsigned quantity of 1, 2, 4, or 8 bytes. Internally, INTEGER values are stored in Intel binary integer format. The 1 byte UNSIGNED uses the character extended data type and the 2, 4 and 8 byte UNSIGNED use the unsigned extended data type when indexed.

Following are the values each column accepts:

Length in Bytes	Value Range
1	0 – 255
2	0 – 65535
4	0 – 4294967295
8	0 – 18446744073709551615

ZSTRING

The ZSTRING data type in Scalable SQL corresponds to a C string. It has the same characteristics as the CHARACTER data type except it is terminated by a byte containing a binary 0. Bytes beyond the specified end of the string are undefined. A ZSTRING value may contain significant blanks immediately preceding the termination character.

Example

For an 8-byte ZSTRING column, the value Hello is stored as follows:

Byte	0	1	2	3	4	5	6	7
Internal Value (Hexadecimal)	48	65	6C	6C	6F	00	?	?
ASCII Value	H	e	l	l	o			

VariableLength Data Types

Scalable SQL allows you to define columns of variable length using the NOTE and LVAR data types. Columns of data types NOTE and LVAR can contain any type of data, and the column values can vary in length from row to row within a table; they are limited only by the maximum amount of data that the MicroKernel is able to store in a data file record.

Five restrictions apply when you use a variablelength column in a table:

- You cannot insert data in a variablelength column if the data is longer than the length you defined for the column in the dictionary definition. The size of a column is stored in a 2-byte integer value. You can specify sizes up to 32,761, or use the default size, which is unlimited.

> **Note:** *Developers only*: If you specify an unlimited size, you must use the chunk operations to fetch or update existing rows. To interact with data in an unlimited sized column, use the buffer format described in the *Programmer's Guide*.

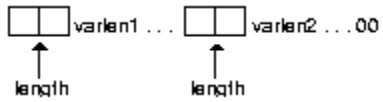
- You cannot include more than one variablelength column when you create a table or a view.
- If you include a variablelength column in a table or a view, you must define the column as the last column in the table or view.
- You cannot define a variablelength column as an index or sort column.
- A table or a view must contain at least one fixedlength data type.
- Data from a variable-length column is always specified and returned in internal format.

LVAR

The maximum record length in Scalable SQL is 32,765 bytes. The data you insert into an LVAR column must not exceed the column length you specify. A record can contain a variable length field defined as the last column; the maximum column length is 32,761 bytes minus the length of any fixed-length data.

The LVAR data type uses length words in conjunction with an end-of-column delimiter. Each variablelength data segment in the column is preceded by a 2byte length word that indicates the length of that segment. The end-of-column delimiter is always 2 bytes of binary zeros.

An LVAR column with multiple data segments is structured as follows:



NOTE

The maximum record length in Scalable SQL is 32,765 bytes. The data you insert into a NOTE column must not exceed the column length you specify. A record can contain a variable length field defined as the last column; the maximum column length is 32,761 bytes minus the length of any fixed-length data.

The NOTE data type uses a binary zero as a delimiter to denote the end of the column value. To determine the length of a NOTE column value, Scalable SQL scans from the end of the value until it reaches the delimiter. It then scans until it reaches a non-delimiter. The last delimiter immediately after the non-delimiter is interpreted as the end of the data.

Scalable SQL Keywords

Scalable SQL supports the following keywords. Keywords that are marked with an asterisk (*) are Scalable SQL extensions to the ANSI and IBM SAA SQL standards. Keywords in parentheses are acceptable abbreviations of the preceding keyword.

List of Keywords

ABSOLUTE	ACCELERATED	ADD	AFTER
ALL	ALTER	AND	ANY
AS	ASC	ATOMIC	AUTOINC*
AVG	BEGIN	BEGINS*	BEFORE
BETWEEN	BFLOAT*	BINARYNULL	BIND
BIT*	BLANK*	BY	CALL
CASCADE	CASE*	CAST	CHAIN
CHARACTER	(CHAR)	CLOSE	COMMIT
CONDITION	CONTAINS*	CONTINUE	COUNT
CREATE	CREATETAB*	CS*	CURDATE*
CURRENCY	CURRENT	CURRENT_TIMESTAMP	CURSOR
CURTIME*	DATABASE	DATE*	DATEFORMAT
DAY*	DCOMPRESS*	DECIMAL	(DEC)
DECIMALNULL	DECLARE	DEFAULT*	DELETE
DESC	DICTIONARY*	DISTINCT	DO
DROP	EACH	ELSE	ELSEIF
END	EX*	EXCLUSIVE*	EXEC
EXECUTE	EXISTS	EXIT	EXTERNAL
FETCH	FIRST	FLOAT	FOR
FOREIGN	FROM	GRANT	GROUP

HANDLER	HAVING	HOUR*	IF
IN	INCLUDE	INDEX*	INOUT
INSERT	INTEGER	(INT)	INTO
IS	ISOLATION	KEY	LAST
LEAVE	LEFT*	LENGTH*	LIKE
LOGICAL	LOGIN*	LOOP	LOWER*
LSTRING	LTRIM*	LVAR*	MASK*
MAX	MILLISECOND*	MIN	MINUTE*
MOD*	MODIFY	MONEY*	MONTH*
NEW	NEXT	NO	NORMAL*
NOT	NOTE*	NOTFOUND	NULL
NUMERIC*	NUMERCSA*	NUMERICSTS*	OF
OLD	ON	ONLY	OPEN
OPENMODE*	OR	ORDER	OUT
OWNER*	OWNERACCESS*	PAGESIZE*	PREALLOCATE*
PREPARE	PRIMARY	PRIOR	PROCEDURE
(PROC)	PUBLIC	RANGE*	READ
READONLY*	REFERENCES	REFERENCING	RELATIVE
RELEASE	REPLACE*	RESIGNAL	RESTRICT
REVOKE	RIGHT*	ROLLBACK	ROW

RTRIM*	SAVEPOINT	SCROLL	SECOND*
SECURITY*	SEG*	SELECT	SET
SIGNAL	SOME	SQLERROR	SQLEXCEPTIONS
SQLSTATE	SQLWARNINGS	START*	STATEMENT
STRINGNULL	SUBSTRING*	(SUBSTR)	SUM
TABLE	THEN	THRESHOLD*	TIME*
TIMESTAMP	TO	TRANSACTION	TRIGGER
TYPE	UNDO	UNION	UNIQUE
UNSIGNED*	UPDATABLE	UPDATE	UPPER*
USER	USING*	VALUE	VALUES
VERIFY*	VIEW	WEEKDAY*	WHEN
WHERE	WHILE	WITH	WORK
YEAR*	ZSTRING*		

System Tables

This appendix describes the Scalable SQL system tables. For each system table, the following table indicates the name of the associated file and briefly describes the system table's contents.

> **Note:** Some data in the system tables cannot be displayed. For example, information about stored views and procedures, other than their names, is available only to Scalable SQL. In addition, some data (such as user passwords) displays in encrypted form.

Table C-1
System Tables

System Table	Dictionary File	Contents
"X$File"	FILE.DDF	Names and locations of the tables in your database.
"X$Field"	FIELD.DDF	Column and named index definitions.
"X$Index"	INDEX.DDF	Index definitions.
"X$Attrib"	ATTRIB.DDF	Column attributes definitions.
"X$View"	VIEW.DDF	View definitions.
"X$Proc"	PROC.DDF	Stored procedure definitions.
"X$User"	USER.DDF	Usernames, group names, and passwords.
"X$Rights"	RIGHTS.DDF	User and group access rights definitions.
"X$Relate"	RELATE.DDF	Referential integrity (RI) information.
"X$Trigger"	TRIGGER.DDF	Trigger information.
"X$Depend"	DEPEND.DDF	Trigger dependencies such as

tables, views, and procedures.

When you issue a CREATE DICTIONARY statement, Scalable SQL creates the X$File, X$Field, and X$Index system tables and the associated dictionary files. Scalable SQL creates the other system tables as follows:

- X$Attrib—When you define column attributes, Scalable SQL creates this table and stores the definitions.
- X$View—When you define views, Scalable SQL creates this table and stores the definitions.
- X$Proc—When you define stored procedures, Scalable SQL creates this table and stores the definitions.
- X$User and X$Rights—When you set up data security on the database, Scalable SQL creates these two tables. In X$User, Scalable SQL stores information about user names, group names, and passwords. In X$Rights, Scalable SQL stores information about the access rights assigned to users and groups. When you disable security, Scalable SQL deletes these two tables.
- X$Relate—When you define RI constraints for the database, Scalable SQL creates this table and stores information about foreign key references.
- X$Trigger and X$Depend—When you define triggers for tables in the database, Scalable SQL creates these two tables. In X$Trigger, Scalable SQL stores information about the triggers. In X$Depend, Scalable SQL stores information about the trigger dependencies.

Because the system tables are part of the database, you can query them to retrieve information about the database. However, to update the system tables, you must use data definition statements. You cannot update them with data manipulation statements as you would standard data tables; this may corrupt the dictionary.

Following is a brief discussion of installing system tables and dictionary files; the remainder of this appendix discusses each system table individually. Each discussion describes the columns and indexes defined for the designated system table.

Installing System Tables and Data Dictionary Files

The system tables included with Scalable SQL contain the data dictionary files for the sample database. When you install Scalable SQL, you can copy this data dictionary to the appropriate device on your system and log in to the sample database. After logging in, you can create a new data dictionary in another directory of your choice. Alternatively, you can create a new data dictionary using the Setup utility to define a bound or unbound named database.

X$File

The X$File system table is associated with the file FILE.DDF. For each table defined in the database, X$File contains the table name, the location of the associated table, and a unique internal ID number that Scalable SQL assigns. The structure of X$File is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xf$Id	UNSIGNED	2	N/A	Internal ID Scalable SQL assigns.
Xf$Name	CHARACTER	20	Yes	Table name.
Xf$Loc	CHARACTER	64	No	File location (pathname).
Xf$Flags	UNSIGNED	1	N/A	File flags. If bit 4=1, the file is a dictionary file. If bit 4=0, the file is user-defined.
Xf$Reserved	CHARACTER	10	No	Reserved.

Two indexes are defined for the X$File table.

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xf$Id	No	N/A	No
1	0	Xf$Name	No	Yes	No

Note: Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Field

The X$Field system table is associated with the file FIELD.DDF. X$Field contains information about all the columns and named indexes defined in the database. The structure of X$Field is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xe$Id	UNSIGNED	2	N/A	Internal ID Scalable SQL assigns.
Xe$File	UNSIGNED	2	N/A	ID of table to which this column or named index belongs. It corresponds to Xf$Id in X$File.
Xe$Name	CHARACTER	20	Yes	Column name or index name.
Xe$DataType	UNSIGNED	1	N/A	Column data type (range 0–20), or 0xFF for a named index.
Xe$Offset	UNSIGNED	2	N/A	Column offset in table; index number if named index.
Xe$Size	UNSIGNED	2	N/A	Column size.
Xe$Dec	UNSIGNED	1	N/A	Column decimal place (for DECIMAL, NUMERIC, NUMERICSA, NUMERICSTS, MONEY, or CURRENCY types). Relative bit positions for contiguous bit columns**.** Fractional seconds for TIMESTAMP data type.
Xe$Flags	UNSIGNED	2	N/A	Flags word. Bit 0 is the case flag for string data types.

The column Xe$File corresponds to the column Xf$Id in the X$File system table and is the link between the tables and the columns they contain.

The integer values in column Xe$DataType are codes that represent the Scalable SQL data types. Five indexes are defined for the X$Field table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xe$Id	No	N/A	No

1	0	Xe$File	Yes	N/A	No
2	0	Xe$Name	Yes	Yes	No
3	0	Xe$File	No	N/A	Yes
3	1	Xe$Name	No	Yes	No
4	0	Xe$File	Yes	N/A	Yes
4	1	Xe$Offset	Yes	N/A	Yes
4	2	Xe$Dec	Yes	N/A	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Index

The X$Index system table is associated with the file INDEX.DDF. X$Index contains information about all the indexes defined on the tables in the database. The structure of X$Index is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xi$File	UNSIGNED	2	N/A	Unique ID of the table to which the index belongs. It corresponds to Xf$Id in X$File.
Xi$Field	UNSIGNED	2	N/A	Unique ID of the index column. It corresponds to Xe$Id in X$Field.
Xi$Number	UNSIGNED	2	N/A	Index number (range 0–119).
Xi$Part	UNSIGNED	2	N/A	Segment number (range 0–119).
Xi$Flags	UNSIGNED	2	N/A	Index attribute flags.

The Xi$Flags column contains integer values that define the index attributes. The following table describes how Scalable SQL interprets each bit position when the bit has the binary value of 1. Bit position 0 is the rightmost bit in the integer.

Bit Position	Decimal Equivalent	Description
0	1	Index allows duplicates.
1	2	Index is modifiable.
2	4	Indicates an alternate collating sequence.
3	8	Null values are not indexed.
4	16	Another segment is concatenated to this one in the index.
5	32	Index is case-insensitive.
6	64	Index is collated in descending order.
7	128	Index is a named index.
8	256	Index is a Btrieve extended key type.

13	8192	Index is a foreign key
14	16384	Index is a primary key referenced by some foreign key

The value in the Xi$Flags column for a particular index is the sum of the decimal values that correspond to that index's attributes. Three indexes are defined for the X$Index table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xi$File	Yes	N/A	No
1	0	Xi$Field	Yes	N/A	No
2	0	Xi$File	No	N/A	Yes
2	1	Xi$Number	No	N/A	Yes
2	2	Xi$Part	No	N/A	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Attrib

The X$Attrib system table is associated with the file ATTRIB.DDF. X$Attrib contains information about the column attributes of each column in the database; there is an entry for each column attribute you define. The structure of X$Attrib is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xa$Id	UNSIGNED	2	N/A	Corresponds to Xe$Id in X$Field.
Xa$Type	CHARACTER	1	No	C for character, D for default, H for heading, M for mask, R for range, or V for value.
Xa$ASize	UNSIGNED	2	N/A	Length of text in Xa$Attrib.
Xa$Attrs	NOTE	<=2048	N/A	Text that defines the column attribute.

When you define multiple attributes for a single column, the X$Attrib system table contains multiple entries for that column ID—one for each attribute you define. If you do not define column attributes for a particular column, that column has no entry in the X$Attrib table. The text in the Xa$Attrs column appears exactly as you define it with Scalable SQL. One index is defined for the X$Attrib table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xa$Id	No	N/A	Yes
0	1	Xa$Type	No	No	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$View

The X$View system table is associated with the file VIEW.DDF. X$View contains view definitions, including information about joined tables and the restriction conditions that define views. You can query the X$View table to retrieve the names of the views that are defined in the dictionary.

The first column of the X$View table contains the view name; the second and third columns describe the information found in the LVAR column, Xv$Misc. The structure of X$View is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xv$Name	CHARACTER	20	Yes	View name.
Xv$Ver	UNSIGNED	1	N/A	Version ID.
Xv$Id	UNSIGNED	1	N/A	Sequence number.
Xv$Misc	LVAR	<=2000	N/A	Scalable SQL internal definitions.

Two indexes are defined for the X$View table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xv$Name	Yes	Yes	No
1	0	Xv$Name	No	Yes	Yes
1	1	Xv$Ver	No	N/A	Yes
1	2	Xv$Id	No	N/A	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Proc

The X$Proc system table is associated with the file PROC.DDF. X$Proc contains the compiled structure information for every stored procedure defined. The structure of X$Proc is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xp$Name	CHARACTER	30	Yes	Stored procedure name.
Xp$Ver	UNSIGNED	1	N/A	Version ID.
Xp$Id	UNSIGNED	2	N/A	0-based Sequence Number.
Xp$Flags	UNSIGNED	1	N/A	1 for stored statement, 2 for stored procedure or 3 for external procedure.
Xp$Misc	LVAR	990	N/A	Internal representation of stored procedure.

One index is defined for the X$Proc table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xp$Name	No	Yes	Yes
0	1	Xp$Id	No	N/A	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$User

The X$User system table is associated with the file USER.DDF. X$User contains the name and password of each user and the name of each user group. Scalable SQL uses this table only when you enable the security option. The structure of X$User is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xu$Id	UNSIGNED	2	N/A	Internal ID assigned to the user or group.
Xu$Name	CHARACTER	30	Yes	User or group name.
Xu$Password	CHARACTER	9	No	User password.
Xu$Flags	UNSIGNED	2	N/A	User or group flags.

Note: For any row in the X$User system table that describes a user group, the column value for Xu$Password is null.

The Xu$Flags column contains integer values whose rightmost 8 bits define the user or group attributes. The following table describes how Scalable SQL interprets each bit position when the bit has the binary value of 1. Bit position 0 is the rightmost bit in the integer.

Bit Position	Decimal Equivalent	Description
0	1	Reserved.
1	2	Reserved.
2	4	Reserved.
3	8	Reserved.
4	16	Reserved.
5	32	Reserved.
6	64	Name is a group name.
7	128	User or group has the right to define tables in the dictionary.

The value in the Xu$Flags column for a particular user or group is the sum of the decimal values corresponding to the attributes that apply to the user or group.

Two indexes are defined for the X$User table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xu$Id	Yes	N/A	No
1	0	Xu$Name	No	Yes	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Rights

The X$Rights system table is associated with the file RIGHTS.DDF. X$Rights contains access rights information for each user. Scalable SQL uses this table only when you enable the security option. The structure of X$Rights is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xr$User	UNSIGNED	2	N/A	User ID
Xr$Table	UNSIGNED	2	N/A	Table ID
Xr$Column	UNSIGNED	2	N/A	Column ID
Xr$Rights	UNSIGNED	1	N/A	Table or column rights flag

The Xr$User column corresponds to the Xu$Id column in the X$User table. The Xr$Table column corresponds to the Xf$Id column in the X$File table. The Xr$Column column corresponds to the Xe$Id column in the X$Field table.

Note: For any row in the system table that describes table rights, the value for Xr$Column is null.

The Xr$Rights column contains integer values whose rightmost 8 bits define the users' access rights. The following table describes how Scalable SQL interprets each bit position when the bit has the binary value of 1. Bit position 0 is the rightmost bit in the integer.

Bit Position	Decimal Equivalent	Description
0	1	Reorganization in progress.
1	2	Reserved.
2	4	Reserved.
3	8	Reserved.
4	16	References rights to table.
5	32	Alter Table rights.
6	64	Select rights to table or column.

7	128	Update, Insert, and Delete rights to table or column.

A decimal equivalent of 0 implies no rights.

The value in the Xr$Rights column for a particular user is the sum of the decimal values corresponding to the access rights that apply to the user.

Three indexes are defined for the X$Rights table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xr$User	Yes	N/A	No
1	0	Xr$User	No	N/A	Yes
1	1	Xr$Table	No	N/A	Yes
1	2	Xr$Column	No	N/A	No
2	0	Xr$Table	Yes	N/A	Yes
2	1	Xr$Column	Yes	N/A	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Relate

The X$Relate system table is associated with the file RELATE.DDF. X$Relate contains information about the referential integrity (RI) constraints defined on the database. The structure of X$Relate is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xr$PId	UNSIGNED	2	N/A	Primary table ID.
Xr$PIndex	UNSIGNED	2	N/A	Index number of primary key in primary table.
Xr$FId	UNSIGNED	2	N/A	Dependent table ID.
Xr$FIndex	UNSIGNED	2	N/A	Index number of foreign key in dependent table.
Xr$Name	CHARACTER	20	Yes	Foreign key name.
Xr$UpdateRule	UNSIGNED	1	N/A	1 for restrict.
Xr$DeleteRule	UNSIGNED	1	N/A	1 for restrict, 2 for cascade.
Xr$Reserved	CHARACTER	30	No	Reserved.

Five indexes are defined for the X$Relate table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xr$PId	Yes	N/A	No
1	0	Xr$FId	Yes	N/A	No
2	0	Xr$Name	No	Yes	No
3	0	Xr$PId	No	N/ANo	Yes
3	1	Xr$Name	No	Yes	No
4	0	Xr$FId	No	N/A	Yes

4	1	Xr$Name	No	Yes	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Trigger

The X$Trigger system table is associated with the file TRIGGER.DDF. X$Trigger contains information about the triggers defined for the database. The structure of X$Trigger is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xt$Name	CHARACTER	30	Yes	Trigger name.
Xt$Version	UNSIGNED	2	N/A	Trigger version; 4 is for Scalable SQL v4.0.
Xt$File	UNSIGNED	2	N/A	File on which trigger is defined. Corresponds to Xf$Id in X$File.
Xt$Event	UNSIGNED	1	N/A	0 for INSERT, 1 for DELETE, 2 for UPDATE.
Xt$ActionTime	UNSIGNED	1	N/A	0 for BEFORE, 1 for AFTER.
Xt$ForEach	UNSIGNED	1	N/A	0 for ROW (default), 1 for STATEMENT.
Xt$Order	UNSIGNED	2	N/A	Order of execution of trigger.
Xt$Sequence	UNSIGNED	2	N/A	0-based sequence number.
Xt$Misc	LVAR	<=4054	N/A	Internal representation of trigger.

Three indexes are defined for the X$Trigger table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xt$Name	No	Yes	Yes
0	1	Xt$Sequence	No	N/A	No
1	0	Xt$File	No	N/A	Yes
1	1	Xt$Name	No	Yes	No
2	0	Xt$File	Yes	N/A	Yes

2	1	Xt$Event	Yes	N/A	Yes
2	2	Xt$ActionTime	Yes	N/A	Yes
2	3	Xt$ForEach	Yes	N/A	Yes
2	4	Xt$Order	Yes	N/A	Yes
2	5	Xt$Sequence	Yes	N/A	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

X$Depend

The X$Depend system table is associated with the file DEPEND.DDF. X$Depend contains information about trigger dependencies such as tables, views, and procedures. The structure of X$Depend is as follows:

Column Name	Type	Size	Case Insensitive	Description
Xd$Trigger	CHARACTER	30	Yes	Name of trigger. It corresponds to Xt$Name in X$Trigger.
Xd$DependType	UNSIGNED	1	N/A	1 for Table, 2 for View, 3 for Procedure.
Xd$DependName	CHARACTER	30	Yes	Name of dependency with which the trigger is associated. It corresponds to either Xf$Name in X$File, Xv$Name in X$View, or Xp$Name in X$Proc.

Two indexes are defined for the X$Depend table, as follows:

Index Number	Segment Number	Column Name	Duplicates	Case Insensitive	Segmented
0	0	Xd$Trigger	No	Yes	Yes
0	1	Xd$DependType	No	N/A	Yes
0	2	Xd$DependName	No	Yes	No
1	0	Xd$DependType	Yes	N/A	Yes
1	1	Xd$DependName	Yes	Yes	No

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

SQLSTATE Classes and Values

Scalable SQL 4.0 has a special session variable called SQLSTATE. Based on proposed ANSI SQL3 standards, the SQLSTATE variable is a five-byte character string that reflects the status of the last statement executed. Previous versions of Scalable SQL returned a status code only. Scalable SQL 4.0 returns a status code to maintain backward compatibility, and sets the SQLSTATE value, as well.

Within a stored procedure or trigger, the logic may respond to given SQLSTATE values either via condition handlers or via explicit SQLSTATE tests, such as:

```
IF (SQLSTATE='00000')

THEN action
```

This appendix contains the following topics:

For more information about stored procedures, triggers, conditions, and condition handlers, refer to Chapter 2, "Scalable SQL Syntax."

Example Use of SQLSTATE

The following example uses the SQLSTATE session variable in a stored procedure. For other examples that use SQLSTATE, see “Examples for CREATE PROCEDURE” on page 2-37 and “Examples for DECLARE HANDLER” on page 2-69.

```
CREATE PROCEDURE c1 (INOUT xx INT (2));

BEGIN

DECLARE c1 CONDITION FOR SQLSTATE '03000';

DECLARE c2 CONDITION FOR SQLSTATE '09000';

DECLARE c3 CONDITION FOR SQLSTATE '01234';


DECLARE CONTINUE HANDLER FOR c1

 BEGIN

   SET xx = xx + 1;

   SIGNAL c2;

 END;


DECLARE CONTINUE HANDLER FOR c2

 BEGIN

   SET xx = xx + 2;

   SIGNAL c3;

 END;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

 BEGIN

   SIGNAL c3;

   SET xx = xx + 4;

 END;


IF (xx < 10) THEN

 SIGNAL c1;

ELSE

 IF (xx = 10) THEN

   SIGNAL c2;
```

```
 ELSE
   SIGNAL c3;
 END IF;
END IF;

IF (SQLSTATE = c3) THEN
 SET xx = 2000;
END IF;
END
```

When the input parameter equals 5, 10, and 12, respectively, the procedure executes as follows:

- When xx = 5:

  Because this value is less than 10, condition c1 is immediately signalled.

  The handler for c1 is invoked. This increments *xx* to 6 and signals condition c2.

  The handler for c2 is invoked. This increments *xx* to 8 and signals condition c3.

  Because c3 is a warning level condition, execution proceeds to the explicit test of SQLSTATE = c3, which succeeds, and *xx* is set to 2000.

- When *xx* = 10:

  Since this value equals 10, condition c2 is immediately signalled.

  The handler for c2 is invoked. This increments xx to 12 and signals c3.

  Since c3 is a warning level condition, execution proceeds to the explicit test of SQLSTATE = c3, which succeeds, and xx is set to 2000.

- When *xx* = 12:

  Because this value is greater than 10, condition c3 is immediately signalled.

  Because c3 is a warning level condition, execution proceeds to the explicit test of SQLSTATE = c3, which succeeds, and *xx* is set to 2000.

The input and output for stored procedure c1 can be summarized as follows:

xx Input	**xx Output**
5	2000
10	2000
12	2000

Types of SQLSTATE Values

The SQLSTATE variable is a five-byte character string. The first two bytes of the SQLSTATE value form a class value; the last three bytes form a subclass value.

The ANSI standard reserves class values that begin with 0 through 4 and A through H for standard defined classes. Class values that begin with any other character are implementation-defined. Within these standard classes, the ANSI standard reserves subclass values that begin with 0 through 4 and A through H for standard defined subclasses. Subclass values that begin with any other character are implementation-defined.

SQLSTATE values can indicate success, no data, warning, or exception. The following table lists the types of SQLSTATE values.

Table D-1
Types of SQLSTATE Values

SQLSTATE		ANSI-level Classification	DBMS-level Classification
Class	**Subclass**		
00	000	Success	
01	000 through 4ZZ	Warning	
500 through 9ZZ		Warning	
A00 through HZZ	Warning		
I00 through ZZZ		Warning	
02	000	No data	
03 through 4Z	000 through 4ZZ	Exception	
A00 through HZZ	Exception		
500 through 9ZZ		Exception	
I00 through ZZZ		Exception	
A0 through HZ	000 through 4ZZ	Exception	
A00 through HZZ	Exception		

500 through 9ZZ		Exception	
I00 through ZZZ		Exception	
50 through 9Z	000	Non-subclassed, user-defined exception	
I0 through ZZ	000	Non-subclassed, user-defined exception	
50 through 9Z	001 through ZZZ		Subclassed, user-defined exception
I0 through ZZ	001 through ZZZ		Subclassed, user-defined exception

By default, success, warning, and no data values allow execution to continue, and exception values cause execution to halt. You can use condition handlers to change this default behavior.

SQLSTATE Classes

The following table summarizes the standard-defined classes that Scalable SQL uses. If a class or range of classes is not listed, then Scalable SQL does not provide a status code that would map to the missing class or the class is reserved for future use by the ANSI SQL4 standard.

Table D-2
Standard SQLSTATE Classes Used by Scalable SQL

Class	Classification	Description
00	Success	All statuses less than or equal to 0, except -109.
01	Warning	Status Code 80, record conflict. (Defined by the standard as “Cursor operation conflict.)
02	No data (such as end-of-file)	Status Code 9 or -109.
08	Connection exception	Login and communication errors.
09	Triggered action exception	Status Code 911.
0A	Feature not supported	Status Code 362 or 902.
0B	Transaction start exception	Start transaction errors.
21	Cardinality violation	Status Code 844.
22	Data exception	Data conversion and constraint errors, mostly status codes in the 200 and 300 range.
23	Integrity constraint violation exception	Referential integrity errors.
24	Invalid cursor state exception	Declared cursor errors.
2D	Transaction termination exception	Disallowed COMMIT/ROLLBACK errors.
34	Invalid cursor name exception	Status Code 893.
38	External routine exception	Inscribe errors.

39	External routine invocation exception	Inscribe errors.
3B	Savepoint exception	Savepoint errors.
40	Transaction rollback exception	Deadlock and lock conflicts.
42	Syntax error or access violation exception	Mostly status codes in the 500 and 800 range.

The following table summarizes the classes defined by the standard that Scalable SQL does not use.

Table D-3
Standard SQLSTATE Classes Not Used by Scalable SQL

Class	Description
03	SQL statement not yet complete
0D	Invalid target type specification
0E	Invalid role specification
0F	Locator exception
0G	Reference to null table value
25	Transaction state exception
26	Invalid SQL statement name
27	Triggered data change violation
28	Invalid authorization specification
2B	Dependent privilege descriptors still exist
2C	Invalid character set name
2E	Invalid connection name

2F	SQL routine exception
30	Invalid SQL statement
31	Invalid target specification value
33	Invalid SQL descriptor
35	Invalid condition number
36	Cursor sensitivity exception
3C	Ambiguous cursor name
3D	Invalid catalog name
3F	Invalid schema name
3G	Invalid ADT instance
44	With check option violation
H1 through H5	SQL/MM
HZ	Remote database access (protocol)

For each status code that does not map to a standard defined SQLSTATE class, Scalable SQL uses the following process to assign a SQLSTATE value for the status code. First, Scalable SQL makes the last three digits of the SQLSTATE value equal to the last three digits of the status code. Next, Scalable SQL makes the first two digits of the SQLSTATE value equal to one of the classes shown in Table D-4.

The following table lists classes defined by Scalable SQL.

Table D-4
Classes Defined by Scalable SQL

Class	Description	Associated Status Codes
K0	MicroKernel status codes	1 through 199

S0	Scalable SQL status codes	200 through 999
M0	MicroKernel status codes for Windows and OS/2 workstations	1000 through 1999
R0	Requester status codes	2000 through 2999
X0	Uncategorized status codes	All others

SQLSTATE Subclasses

The following table lists the subclasses—within the standard classes—that Scalable SQL uses for status codes that do not map directly to a complete, standard SQLSTATE value.

Table D-5
SQLSTATE Subclasses Defined by Scalable SQL

Subclass	Description
I01 through I99	Status Codes 1 through 99
N00 through N99	Status Codes 100 through 199
O00 through O99	Status Codes 200 through 299
P00 through P99	Status Codes 300 through 399
Q00 through Q99	Status Codes 400 through 499
500 through 999	Status Codes 500 through 999
J00 through J99	Status Codes 1000 through 1999
L00 through L99	Status Codes 2000 through 2999
U00 through U99	Status Codes 3000 through 5999 and 7000 through 9999

The first digit of the subclass indicates the range of the status code, and the last two digits of the subclass indicate the last two digits of the status code.

Application-Defined SQLSTATE Values

Your application can define its own SQLSTATE classes that begin with the letters W, Y, and Z. For example, your application could use the SQLSTATE class *W9* for exceptions. As with all SQLSTATE values, each application-specific SQLSTATE value must include a subclass value. Your application can use any subclass value except 000, which is reserved by the ANSI standard to mean *no subclass*. If your application uses OOO, ANSI defines this to mean 000.

SQLSTATE Values

The following table provides a comprehensive list of status codes and their associated SQLSTATE values.

Status Code	SQLSTATE Value
0	00000
1	K0001
2	K0002
3	K0003
4	K0004
5	22028
6	K0006
7	K0007
8	24I08
9	02000
10	42I10
11	K0011
12	K0012
13	K0013
14	K0014
15	K0015
16	K0016
17	K0017
18	K0018

19	K0019
20	K0020
21	K0021
22	K0022
23	K0023
24	K0024
25	K0025
26	K0026
27	K0027
28	K0028
29	K0029
30	K0030
31	K0031
32	K0032
33	K0033
34	K0034
35	K0035
36	25I36
37	25I37
38	K0038

39	25I39
40	K0040
41	K0041
42	K0042
43	K0043
44	K0044
45	K0045
46	K0046
47	K0047
48	K0048
49	K0049
50	K0050
51	K0051
52	K0052
53	K0053
54	K0054
55	K0055
56	K0056
57	K0057
58	K0058
59	K0059

60	K0060
61	K0061
62	K0062
63	K0063
64	K0064
65	K0065
66	K0066
67	K0067
68	23I68
69	K0069
71	23I71
72	23I72
73	23I73
74	K0074
75	K0075
76	K0076
77	K0077
78	40001
79	K0079
80	K0080

81	K0081
82	K0082
83	K0083
84	40I84
85	40I85
86	K0086
87	K0087
88	K0088
89	K0089
90	K0090
91	08I91
92	K0092
93	K0093
94	K0094
95	08I95
96	08I96
97	K0097
98	K0098
99	K0099
100	K0100
101	K0101

102	K0102
103	K0103
104	K0104
105	K0105
106	K0106
107	K0107
109	K0109
110	K0110
111	K0111
112	K0112
113	K0113
114	K0114
115	K0115
130	K0130
132	K0132
133	K0133
134	K0134
135	K0135
136	K0136
139	K0139

140	K0140
143	K0143
147	K0147
148	K0148
149	K0149
151	K0151
160	K0160
161	K0161
162	K0162
200	S0200
201	S0201
202	S0202
203	08O03
204	42O04
205	08O05
206	S0206
207	42O07
208	S0208
209	08O09
210	S0210
211	S0211

212	S0212
213	42O13
214	42O14
215	S0215
218	S0218
219	S0219
220	S0220
221	42O21
222	42O22
223	22O23
224	22O24
225	42O25
226	S0226
227	S0227
228	S0228
229	42O29
230	S0230
231	S0231
232	S0232
234	S0234

235	S0235
236	42O36
237	42O37
238	S0238
239	42O39
240	42O40
241	42O41
242	42O42
243	42O43
244	42O44
245	42O45
246	42O46
247	42O47
248	22O48
249	S0249
250	42O50
251	42O51
252	42O52
253	42O53
254	42O54
255	S0255

256	S0256
257	42O57
258	42O58
259	42O59
260	42O60
261	42O61
262	42O62
263	42O63
264	S0264
265	08O65
266	08O66
269	42O69
270	42O70
271	42O71
272	42O72
273	42O73
274	42O74
275	42O75
276	S0276
277	S0277

278	S0278
279	42O79
280	42O80
281	S0281
282	42O82
283	42O83
284	42O84
285	S0285
286	42O86
287	S0287
288	S0288
289	S0289
290	42O90
291	42O91
292	22O92
293	22O93
294	22O94
295	S0295
296	S0296
297	S0297
298	S0298

299	S0299
300	S0300
301	S0301
302	S0302
303	S0303
304	S0304
305	S0305
306	42P05
307	S0307
308	42P08
309	S0309
310	42P10
311	S0311
312	S0312
313	0B000
314	S0314
315	S0315
316	S0316
317	S0317
318	42P18

319	22P19
320	22P20
321	22P21
322	22P22
323	42P23
324	S0324
325	42P25
326	42P26
327	S0327
328	S0328
329	S0329
330	22P30
331	42P31
332	S0332
333	S0333
334	42P34
335	S0335
336	S0336
337	42P37
338	42P38
339	S0339

340	42P40
341	42P41
342	42P42
343	42P43
344	S0344
345	S0345
346	22P46
347	S0347
348	S0348
349	22P49
350	42P50
351	25P51
352	S0352
353	S0353
354	S0354
357	S0357
358	S0358
359	S0359
360	S0360
361	S0361

362	0AP62
363	42P63
364	42P64
365	42P65
366	42P66
367	42P67
368	42P68
369	42P69
370	S0370
371	S0371
373	S0373
380	S0380
381	S0381
501	42501
502	42502
503	S0503
504	42504
505	S0505
506	42506
507	42507
508	42508

509	42509
510	42510
511	42511
512	42512
513	42513
514	42514
515	42515
516	42516
517	42517
518	42518
519	42519
520	42520
521	42521
522	42522
523	42523
524	42524
525	42525
526	42526
527	42527
528	42528

529	42529
530	42530
531	42531
532	42532
533	42533
534	S0534
535	42535
536	42536
537	42537
538	42538
539	S0539
540	42540
541	42541
542	42542
543	S0543
544	42544
545	42545
546	42546
547	42547
548	42548
549	42549

550	S0550
551	42551
552	42552
553	42553
554	42554
555	42555
556	42556
557	42557
558	42558
559	42559
560	42560
561	42561
562	42562
563	S0563
564	42564
565	42565
566	42566
567	42567
568	42568
569	42569

570	S0570
571	S0571
800	S0800
802	S0802
803	S0803
804	S0804
805	42805
806	42806
807	S0807
808	42808
809	S0809
810	S0810
811	22811
812	22812
813	22813
815	S0815
816	42816
818	42818
819	S0819
820	S0820
821	S0821

822	S0822
823	42823
824	S0824
825	42825
826	S0826
827	42827
828	S0828
829	S0829
830	42830
831	42831
832	S0832
833	42833
835	S0835
836	42836
837	S0837
838	42838
839	42839
840	S0840
841	S0841
842	42842

843	S0843
844	21000
845	42845
846	42846
847	42847
848	42848
849	S0849
850	S0850
851	42851
852	S0852
853	S0853
854	S0854
856	42856
857	S0857
858	S0858
859	42859
860	S0860
861	S0861
862	42862
863	S0863
864	42864

865	S0865
866	S0866
867	S0867
868	S0868
869	42869
870	42870
871	42871
872	42872
873	42873
874	42874
875	42875
876	42876
877	S0877
878	42878
879	42879
880	42880
881	42881
882	42882
883	42883
884	42884

885	42885
886	S0886
887	42887
888	42888
889	42889
890	42890
891	42891
892	3B001
893	S0893
894	24894
895	24895
896	42896
897	42897
898	42898
899	24899
900	S0900
901	42901
902	0A902
903	42903
904	42904
905	42905

906	42906
907	42907
908	42908
909	42909
910	S0910
911	09000
913	S0913
914	S0914
915	S0915
1001	M1001
1002	M1002
1003	M1003
1004	M1004
1005	M1005
1006	M1006
1007	M1007
1008	M1008
1009	M1009
1010	M1010
1011	M1011

1012	M1012
1013	M1013
1014	M1014
1015	M1015
1016	M1016
1017	M1017
1018	M1018
1019	M1019
1020	M1020
1021	M1021
2000	S0910
2001	S0910
2002	S0910
2003	S0910
2004	S0910
2005	S0910
2006	S0910
2007	S0910
2008	S0910
2009	S0910
2010	S0910

2011	S0910
2012	S0910
2101	08L01
2102	08L02
2103	08L03
2104	08L04
2105	08L05
2106	08L06
2107	08L07
2108	08L08
2109	08L09
2110	08L10
2111	08L11
2112	08L12
2113	08L13
2114	08L14
2115	08L15
2116	08L16
2117	08L17
2118	08L18

2119	08L19
2120	08L20
2121	08L21
2122	08L22
2200	S0910
2201	S0910
2202	S0910
2203	S0910
2204	S0910
2205	S0910
2206	S0910
2300	S0910
2303	S0910
2305	S0910
2306	S0910
2307	S0910
2309	S0910
2310	S0910
2311	S0910
2312	S0910
2313	S0910

2314	S0910
2315	S0910
2316	S0910
2317	S0910
2318	S0910
2319	S0910
2320	S0910
2321	S0910
2322	S0910
2323	S0910
2324	S0910
2325	S0910
2326	S0910
2327	S0910
2328	S0910
2329	S0910
2330	S0910
2900	38L00
2901	38L01
2902	38L02

2903	38L03
2904	39L04
2905	39L05
2906	39L06
2907	39L07
2908	39L08
2909	39L09
2910	39L10
2911	38L11
2912	38L12
2913	38L13
2915	39L15
2916	39L16
2917	38L17
2918	38L18
2919	38L19
3000	08U00
3001	08U01
3002	08U02
3003	08U03
3004	08U04

3005	08U05
3006	08U06
3007	08U07
3008	08U08
3009	08U09
3010	08U10
3011	08U11
3012	08U12
3013	08U13
3014	08U14
3015	08U15
3016	08U16
3017	08U17
3018	08U18
3019	08U19
3020	08U20
3021	08U21
3022	08U22
3023	08U23
3024	08U24

3100	08U00
3101	08U01
3102	08U02
3103	08U03
3104	08U04
3105	08U05
3106	08U06
3107	08U07
3108	08U08
3109	08U09
3110	08U10
3111	08U11
3112	08U12
3113	08U13
3114	08U14
3115	08U15
3116	08U16
3117	08U17
3118	08U18
3119	08U19
3120	08U20

3121	08U21
3122	08U22
7002	X0002
7003	X0003
7004	X0004
7005	X0005
7006	X0006
7007	X0007
7008	X0008
7009	X0009
7010	X0010
7011	X0011
7012	X0012
7013	X0013
7014	X0014
7016	X0016
7017	X0017
7018	X0018
7019	X0019
7020	X0020

7021	X0021
7022	X0022
7023	X0023
7026	X0026
7028	X0028
7029	X0029
7030	X0030
7031	X0031
7032	X0032
7033	X0033
7035	X0035
7036	X0036
7037	X0037
7038	X0038
7039	X0039
7040	X0040
7041	X0041
7042	X0042
7046	X0046
7048	X0048
7049	X0049

7050	X0050
7051	X0051
7052	X0052
7053	X0053
7054	X0054
7061	X0061
7062	X0062
7063	X0063
7064	X0064
7065	X0065
8001	X0001
8002	X0002
8003	X0003
8004	X0004
8005	X0005
8006	X0006
8007	X0007
8008	X0008
8009	X0009
8010	X0010

8011	X0011
8012	X0012
8013	X0013
8014	X0014
8015	X0015
8016	X0016
8017	X0017
8018	X0018
8019	X0019
8020	X0020
8021	X0021
8022	X0022
8023	X0023
8024	X0024
8025	X0025
8026	X0026
8027	X0027
8028	X0028
8029	X0029
8030	X0030
8031	X0031

8032	X0032
8033	X0033
8034	X0034
8035	X0035
8036	X0036
8037	X0037
8038	X0038
8039	X0039
8040	X0040
8041	X0041
8200	X0200
8201	X0201
8202	X0202
8203	X0203
8204	X0204
8205	X0205
8206	X0206
8207	X0207
8208	X0208
8209	X0209

8210	X0210
8211	X0211
8212	X0212
8213	X0213
8214	X0214
8215	X0215
8216	X0216
8217	X0217
8218	X0218