23 SQL

The SQL proc lets you execute SQL statements that operate on SAS data. SQL, Structured Query Language, is the closest thing there is to a standard language for retrieving data from databases. SQL is not exactly a programming language, because it does not describe a sequence of actions. Instead, an SQL statement describes an outcome.

The simplest and most common action in SQL is to extract a subset of data from a table. The SQL statement to accomplish this is a description of the data to extract. SQL statements can often accomplish the same results as SAS statements, but because SQL's approach is so different, the SQL statements may look nothing like the SAS statements that do the same thing. For some tasks that involve combining data from several tables, the SQL approach can be more direct and concise than the SAS approach. There are other reasons why you might use SQL code in a SAS program. If you use SAS/ACCESS to connect to a database management system, you can write SAS SQL statements to access a database or to combine database data with SAS data. Or, if you move an existing project into the SAS environment, you can incorporate the project's existing SQL code into the new SAS programs.

IBM developed SQL in the 1970s as a database interface based on the relational database model, which IBM also developed around that same time. SQL has been used in nearly every relational database management system beginning with the original release of Oracle in 1979. It has been the subject of a series of ANSI and ISO standards since 1986. However, no implementation of SQL has ever followed any standard exactly. SAS SQL is mostly compliant with the 1992 SQL standard.

Lexicon

The prerelease version of SQL was called *SEQUEL* (for Structured English Query Language), and some people still pronounce SQL as "sequel."

SAS was originally developed around the same time as SQL, and both use many of the same concepts of data organization. However, some of the most important data objects are called different names in the SAS and SQL environments. The following table translates between SAS and SQL terminology.

SAS Term	SQL Term	Description
SAS data file	table	a file containing organized data
variable	column	a distinct data element
observation	row	an instance of data, including one value for each variable or column
missing	null	a special value of a data element that indicates that a value is not available
integrity constraint	constraint	a specific restriction on the value of a data element

Lexicon

The SAS and SQL terms for data objects differ because of the different purposes for which they were originally envisioned. The SAS terms *variable*, *observation*, and *missing* are the terms used in the field of statistics. The SQL terms *table*, *column*, and *row* are based on relational database theory.

The `PROC SQL` step starts with the `PROC SQL` statement, which indicates that the statements that follow are SQL statements. The `PROC SQL` statement can set options that affect the execution of the SQL statements. These options can be changed between SQL statements using the RESET statement. SQL uses an interpretive style, executing each statement as soon as it reaches it. You can write global statements to execute between the SQL statements; the settings of a TITLE statement or other global statement take effect for the following SQL statement. Use the QUIT statement, if necessary, to mark the end of the `PROC SQL` step. Because the proc executes each statement immediately, it ignores the RUN statement.

The syntax of the `PROC SQL` step is summarized as:

```
PROC SQL options;
statement
. . .
QUIT;
```

Each statement after the `PROC SQL` statement can be an SQL statement, a RESET statement, or a global statement. The QUIT statement that marks the end of the step is optional.

Among the various differences in style between SAS and SQL syntax, there is one difference that is especially important to notice. In SQL, a reference to a table, column, or any other data object is an expression. Therefore, in any list of objects in an SQL statement, you must write commas as separators between list items. This contrasts with the SAS style, in which a reference to an object is merely a name, and in lists, the names are separated only by spaces.

Query Expressions

SQL syntax is built around specific kinds of expressions, the most important of which is the query expression. A query expression defines a set of data in the shape of a table. The data of a query expression could be the entire contents of a table, but more often, it is a smaller amount of data extracted from one or more tables.

Lexicon

Strictly speaking, the simple form of the query expression that is described here is a *table expression*. More complicated query expressions use set operators to combine two or more table expressions.

The SELECT and WHERE Clauses

A minimal query expression contains the word SELECT, a list of columns, the word FROM, and a table. These terms form a SELECT clause, to indicate what table to draw data from, what columns to use, and the order of the columns. This is an example of a SELECT clause:

```
SELECT ADDRESS, TITLE, SIZE FROM MAIN.PAGES
```

The table is the SAS dataset MAIN.PAGES; the columns that are selected are the variables ADDRESS, TITLE, and SIZE in that SAS dataset.

This kind of query expression returns all the rows from the indicated table. To return only selected rows, add a WHERE clause with the condition for selecting rows. The syntax for the WHERE condition is the same as for any WHERE condition in SAS software. This is an example of a query expression with a WHERE clause:

```
SELECT ADDRESS, TITLE, SIZE FROM MAIN.PAGES
WHERE SIZE <= 81920
```

With the WHERE condition, the query expression returns only those rows where the value of the column SIZE is no more than 81,920.

A SELECT clause and a WHERE clause are sufficient to form most query expressions. The following table summarizes the syntax of these clauses:

Terms	Meaning
`SELECT` *column*, *column*, . . .	Selects columns and indicates their order
`FROM` *table*, *table*, . . .	The table that contains the columns
`WHERE` *condition*	Condition for selecting rows (optional)

The order of the clauses in a query expression is important. It is a syntax error if the clauses are in the wrong order.

SELECT Statement

A query expression can be used by itself as a statement. This kind of SQL statement is called a SELECT statement. It produces a table report of the data that the query expression selects. This is an example of a `PROC SQL` step that executes a SELECT statement:

```
PROC SQL;
SELECT ADDRESS, TITLE, SIZE FROM MAIN.PAGES
  WHERE SIZE <= 81920;
```

As with any SAS statement, an SQL statement ends in a semicolon.

A SELECT statement can also contain an ORDER BY clause to indicate the order of the rows. Write the sort order with commas between the columns. Use the modifier DESC after a column to sort in descending order of that column. This is an example of a SELECT statement with an ORDER BY clause:

```
SELECT ADDRESS, TITLE, SIZE FROM MAIN.PAGES
  WHERE SIZE <= 81920
  ORDER BY TITLE, SIZE DESC;
```

Print Output

Like the PRINT and REPORT procs, the SELECT statement produces print output in table form. This example demonstrates the print output of the SELECT statement:

```
PROC SQL;
SELECT * FROM MAIN.CITYS;
QUIT;
```

```
STATE                    POP     URBAN
----------------------------------
Ohio                10847040      81.4
Michigan             9295895      82.8
```

The SQL table output is similar to the output that the PRINT proc produces with the NOOBS and WIDTH=FULL options. It is similar to the output of the REPORT proc with the HEADLINE and WRAP options. PROC PRINT and PROC REPORT steps and output are shown here for comparison:

```
PROC PRINT DATA=MAIN.CITYS NOOBS WIDTH=FULL;
RUN;
PROC REPORT DATA=MAIN.CITYS NOWD HEADLINE;
RUN;
```

```
STATE                          POP            URBAN

Ohio                      10847040             81.4
Michigan                   9295895             82.8
```

```
  STATE                    POP      URBAN
  ----------------------------------
  Ohio                10847040       81.4
  Michigan             9295895       82.8
```

List of Values

In a SELECT clause, use the keyword DISTINCT before the list of columns to eliminate duplicates from the selected data. The result is a list of the distinct combinations of values of the selected columns. If you list only one column with the DISTINCT option, it produces a list of the different values of that column.

The following example shows the effect of the DISTINCT keyword in a query. The first query shows the value of TYPE for every row in the table. The second query, with the word DISTINCT added, shows only the two different values of TYPE, and it arranges them in sorted order.

```
SELECT TYPE FROM MAIN.LETTER;
SELECT DISTINCT TYPE FROM MAIN.LETTER;
```

```
TYPE
---------
Vowel
Consonant
Consonant
Consonant
Vowel
Consonant
...
```

```
TYPE
---------
Consonant
Vowel
```

Combining Tables

In relational database theory, it is expected that related columns are often stored in separate tables. Queries often combine columns from two or more tables. To write a query that draws columns from multiple tables:

- List the tables, separated by commas, after the keyword FROM.
- After each table name, write an alias. An alias is a one-word name for the table that is used in the query. Programmers usually use single letters for table aliases — often the letters A, B, and so on.
- In the query, identify columns with two-level names that combine the table alias and the column name. For example, the column STATE in the table whose alias is A is referred to as A.STATE.
- In the WHERE expression, include the condition for combining rows between the two tables. Most often, the requirement for combining rows is that the key variables match. For example, if the key variables that connect two tables are DAY and STATE, the WHERE condition could be `A.DAY = B.DAY AND A.STATE = B.STATE`. Writing the WHERE condition correctly is critical for a multi-table query (assuming the query uses tables that have multiple rows). If the WHERE condition is incorrect, the result of the query could be an enormous number of rows or no rows at all.

The query below is an example. It draws the columns NAME, AGE, and SUBSPECIES from the table MAIN.TIGER and adds the column NATIVE from the table MAIN.TSUBS. It shows only rows for which the values of SUBSPECIES match between the two tables. The selected column names appear as the column headings in the output.

```
SELECT I.NAME, I.AGE, I.SUBSPECIES, T.NATIVE
FROM MAIN.TIGER I, MAIN.TSUBS T
WHERE I.SUBSPECIES = T.SUBSPECIES;
```

```
NAME                 AGE  SUBSPECIES        NATIVE
-----------------------------------------------------------
Leah                   7  Bengal            Bay of Bengal
Max                    5  Indochinese       SE Asia
Pierce                 2  Siberian          SE Siberia
Princess               3  Bengal            Bay of Bengal
Stick                  8  Sabertooth        N Eurasia
```

Creating Files From Query Expressions

In a SELECT statement, the results of a query are converted to an output object. Query results can also be stored as data. The CREATE TABLE statement creates a table with the results of a query. The CREATE VIEW statement stores the query itself as a view. Either way, the data identified in the query can be used in later SQL statements or in other SAS steps.

A SELECT statement is converted to a CREATE TABLE statement by adding terms to the beginning of the statement. The added words are CREATE TABLE, the name of the table, and AS. This is an example of a CREATE TABLE statement:

```
CREATE TABLE WORK.PAGE80 AS
  SELECT ADDRESS, TITLE, SIZE FROM MAIN.PAGES
  WHERE SIZE <= 81920
  ORDER BY TITLE, SIZE DESC;
```

```
NOTE: Table WORK.PAGE80 created, with 25 rows and 3 columns.
```

The new table WORK.PAGE80 is a SAS data file. The log note that describes it is the similar to the note that ordinarily describes a new SAS data file, but it uses the SQL words `table`, `columns`, and `rows` in place of the SAS words `SAS data set`, `variables`, and `observations`.

The CREATE VIEW statement creates an SQL view. It is the same as the CREATE TABLE statement except that the word VIEW replaces the word TABLE. A view can be used the same way a table is used. However, a view does not operate the same way as a table. When a table is created, the query is executed and the resulting data is stored in a file. When a view is created, the query itself is stored in the file. The data is not accessed at all in the process of creating a view. The log note says only that a new view has been stored:

```
NOTE: SQL view WORK.CABLES has been defined.
```

The query is executed, and the number of rows and columns determined, only when a later program reads from the view.

Query expressions depend on librefs to identify the tables from which they draw data. The libref definitions of an SQL view can be stored in the view itself, so that the view is self-contained. At the end of the CREATE VIEW statement, write a USING clause that contains LIBNAME statements. Write the LIBNAME statements as that statement is normally written, but with commas, rather than semicolons, separating multiple LIBNAME statements.

Column Expressions and Modifiers

Most columns in queries are simply columns of a table. However, this is only one of the possibilities for writing a column expression in a query expression. The table below lists various ways that a column expression can be formed.

Expression	Description
column	A column of a table.
table alias.column	A column in the indicated table.
*	All columns of all tables in the query.
*table alias.**	All columns of the indicated table.
constant	A constant value.
expression	An expression that computes a value using columns, constants, operators, and functions.

When SAS variables are used as columns in a query, the SQL proc uses the format and label attributes of the columns in the resulting output object, table, or view. However, the SQL proc does not use the FORMAT or LABEL statements that other procs use to change these attributes. Instead, to change the appearance of a column, use the FORMAT= and LABEL= column modifiers. List the modifiers after the column name or expression (not separated by commas). This is an example of a SELECT statement that uses column modifiers:

```
SELECT
  START FORMAT=DATE9. LABEL='Start Date',
  END FORMAT=DATE9. LABEL='End Date'
  FROM MAIN.EVENTS
  ORDER BY START, END;
```

The INFORMAT= and LENGTH= column modifiers can also be used to set those two attributes. These attributes are especially useful when creating a table.

When a query column is not a table column, or when the same column name is used more than once in the same query, it is useful to assign an alias to it. The alias can be used as a name for the query column anywhere else in the same query. For example, aliases can be used in column expressions and WHERE expressions. The alias is also the name of the column in the print output or in the table or view created from the query. To assign an alias, write the keyword AS and the alias at the end of the column definition (after the column modifiers, if there are any). This example includes two column definitions with aliases:

```
SELECT
  TEMPERATURE/1.8 + 32 AS TEMPERATURE_FAHRENHEIT,
  PRECIPITATION AS RAIN
  FROM MAIN.WEATHER
```

```
WHERE RAIN > 0 AND TEMPERATURE_FAHRENHEIT >= 32
;
```

If you do not provide an alias for a column that is computed as an expression, the column is displayed without column headings. If you create a table or view that includes unnamed columns, the SQL proc generates names for the columns based on the column numbers.

SQL Expressions

Column expressions and WHERE expressions are two examples of SQL expressions that result in single values. These SQL expressions are also used in several other places in SQL syntax. The rules of syntax for SQL expressions are mostly the same as that of SAS expressions, with these differences:

- The WHERE operators are used.
- Comparison operators do not use the : modifier.
- The NOT operator has the lowest priority of any SQL operator.
- SQL expressions cannot use the queue functions or those that refer to specific data step objects.
- The INPUT and PUT functions do not use error control terms.
- SQL also supports the COALESCE function. This function returns the first non-null (nonmissing) value among its arguments. The arguments can be any number of columns of the same data type.
- The CASE operator of SQL allows logic within an expression that resembles the SELECT block of the data step. This is an example of a column definition that uses a CASE expression and an alias:

```
CASE
   WHEN AMOUNT < 0 THEN 'Credit'
   WHEN AMOUNT > 0 THEN 'Debit'
   ELSE 'No Balance'
   END
AS BALANCE
```

Dataset Options and Reserved Words in SQL Statements

When SAS datasets are used as tables in SQL statements, you can use any of the usual dataset options. As always, write the dataset options in parentheses after the SAS dataset name.

The SQL standards restrict the use of all words that SQL uses as keywords. In standard SQL, these reserved words cannot be used as the names of SQL objects. SAS SQL reserves only a few words. The names CASE and USER cannot be used as column names. If these are names of SAS variables that you want to use in a query, use the RENAME= dataset option to change their names.

Do not use SQL keywords as table aliases. Most programmers use single letters as table aliases; no single letter is a reserved word. It is also safe to use a letter or word with a numeric suffix, such as T1, as a table alias; no reserved word has a numeric suffix.

Summary Statistics

SQL can calculate summary statistics for a column. Write the statistic, then the column name in parentheses. For example, SUM(TRAFFIC) calculates the sum of the column TRAFFIC. The syntax is the same as a function call, but it is computed as a column statistic as long as the function name is a statistic and the argument is a column or column alias.

The available statistics are the standard set of SAS statistics, other than SKEWNESS and KURTOSIS. In addition to the usual SAS names for the statistics, AVG can be used as an alias for MEAN and COUNT and FREQ as aliases for N. Use the expression COUNT(*) to count the rows in a table.

Use aliases to create column names for statistic columns. Without aliases, statistic columns are displayed without column headings. This is an example of a query that uses summary statistics as columns:

```
TITLE1 'RIAA Yearend Statistics (Units Shipped)';
SELECT
  MIN(CD) FORMAT=COMMA14. AS MinCD,
  MIN(CASSETTE) FORMAT=COMMA14. AS MinCassette,
  MIN(LP_EP) FORMAT=COMMA14. AS MinLP_EP,
  MIN(SINGLE) FORMAT=COMMA14. AS MinSingle,
  MEAN(CD) FORMAT=COMMA14. AS AvgCD,
  MEAN(CASSETTE) FORMAT=COMMA14. AS AvgCassette,
  MEAN(LP_EP) FORMAT=COMMA14. AS AvgLP_EP,
  MEAN(SINGLE) FORMAT=COMMA14. AS AvgSingle,
  MAX(CD) FORMAT=COMMA14. AS MaxCD,
  MAX(CASSETTE) FORMAT=COMMA14. AS MaxCassette,
  MAX(LP_EP) FORMAT=COMMA14. AS MaxLP_EP,
  MAX(SINGLE) FORMAT=COMMA14. AS MaxSingle,
  COUNT(*) AS Years
  FROM RIAA.UNITS;
```

```
RIAA Yearend Statistics (Units Shipped)

         MinCD     MinCassette        MinLP_EP      MinSingle
         AvgCD     AvgCassette        AvgLP_EP      AvgSingle
         MaxCD     MaxCassette        MaxLP_EP      MaxSingle     Years
---------------------------------------------------------------------
   286,500,000     123,600,000       1,200,000     75,400,000
   622,560,000     280,620,000       3,600,000    117,130,000
   938,900,000     442,200,000      11,700,000    191,500,000        10
```

A statistic can use the keyword DISTINCT before the column name. With DISTINCT, the statistic is applied to the set of distinct values in the column, rather than the values of all the rows. For example, COUNT(PLACE) counts rows in which PLACE has a value, but COUNT(DISTINCT PLACE) counts the different values of PLACE. As another example, MEAN(X) is the mean of X for all rows in the table, but MEAN(DISTINCT X) is the mean of the set of distinct values of X.

If all columns in a query are summary statistics, the result is one row that contains a summary of the rows that the query reads. If a query contains a combination of summary statistics and other expressions, the statistics are repeated in each row of the result. Combining summary statistics and detail data in this way is called *remerging*. A log note indicates the process of remerging in case you wrote a remerging query without realizing it.

Summary statistics and detail data can be used together in column expressions. The most common use of this is to calculate percents or relative frequencies, as in this example:

```
SELECT BROWSER, HIT FORMAT=COMMA9. AS HITS,
  HIT/SUM(HIT)*100 FORMAT=7.3 AS SHARE
  FROM MAIN.BROWSERS ORDER BY BROWSER;
```

```
BROWSER                     HITS    SHARE
-----------------------------------------
AOL 3                        811    2.117
Internet Explorer 2        2,667    6.962
Internet Explorer 3        1,085    2.832
Internet Explorer 4        8,515   22.229
Internet Explorer 5        5,377   14.037
Lynx                          89    0.232
Netscape 1                   163    0.426
Netscape 2                   707    1.846
Netscape 3                 4,098   10.698
Netscape 4                 5,950   15.533
Netscape 5                 8,182   21.360
Other                        532    1.389
Prodigy                       95    0.248
Sega Saturn                   35    0.091
```

Grouping

The GROUP BY clause in a query expression lets you divide the rows of the query into groups and apply summary statistics within those groups. The GROUP BY clause follows the WHERE clause, if there is one. It usually lists one or several columns. This is an example of a GROUP BY clause:

```
  GROUP BY STATE, YEAR
```

The GROUP BY columns are essentially the same as class variables. They organize the data into groups. Statistics are calculated within the groups instead of being calculated for the entire set of data. In this example, statistics are calculated separately for each state and year.

GROUP BY items are usually table columns, but they can also be column expressions. If you use an integer as a GROUP BY item, it is used as a column number, and that column of the query is used for grouping. This makes it possible to group by computed columns that do not have names. A better approach, however, is to use aliases for the computed columns.

If there is a GROUP BY clause and all of the columns are summary statistics or GROUP BY items, the query results in only one row for each group. If the query contains a combination of summary statistics and other expressions, the summary statistics are repeated in each row of the group. If summary statistics are used to calculate percents, they are percents of the total for the group, rather than percents of the total for the entire set of data.

The simplest use of grouping is to create a frequency table, as shown in this example:

```
SELECT SYMBOL, NAME, COUNT(*) AS FREQUENCY
  FROM MAIN.SPLIT GROUP BY SYMBOL, NAME;
```

```
SYMBOL NAME                            FREQUENCY
------------------------------------------------
AAPL   Apple Computer Inc                      1
AOL    America Online                          5
BRCM   Broadcom Corp                           2
CPB    Campbell Soup                           1
CRA    PE Corp - Celera Genomics Grp           1
HGSI   Human Genome Sciences Inc               1
IBM    Intl Business Machines                  2
K      Kellogg Co                              1
QCOM   Qualcomm Inc                            2
RHAT   Red Hat Inc                             1
SUNW   Sun Microsystems                        4
TWX    Time Warner Inc                         1
YHOO   Yahoo Inc                               4
```

You cannot use the WHERE clause to select groups or rows based on summary statistics of a group. That is because the WHERE clause is always evaluated separately for each individual row. Instead, when a condition contains summary statistics, write it in a HAVING clause. Write the HAVING clause after the GROUP BY clause. In a query that has no GROUP BY clause, a HAVING clause is applied to all the rows as one group.

The HAVING condition is used instead of the WHERE condition in queries that combine summary rows of tables. The criterion for matching summary rows must be written in the HAVING clause, rather than the WHERE clause, because the WHERE clause applies only to individual rows, not to the summary rows of groups.

Other Queries

SQL has several more features that make a much wider range of queries possible.

- *Table join operators* represent alternate ways to combine two tables: LEFT JOIN, RIGHT JOIN, and FULL JOIN.
- *Set operators* combine the results of two query expressions. The set operators are UNION, OUTER UNION, EXCEPT, and INTERSECT. The INTERSECT operator can modified by the CORRESPONDING and ALL modifiers.
- A *database query* is an expression in the form CONNECTION TO *database* (*query expression*). The query is passed to a database, and the results that are returned from the database are used within the SAS query expression. This *SQL pass-through* feature requires SAS/ACCESS and other statements within the PROC SQL step to connect to the database.
- *Subqueries* are queries written in parentheses and used as values or as tables within a query expression. This is an example of the use of

a subquery to determine the number of different combinations of values for two columns:

```
SELECT COUNT(*) AS N
  FROM (SELECT DISTINCT DEPT, VENDOR FROM CORP.SOURCE);
```

- An *INTO clause* in a query expression assigns the results of the query to macro variables.

DICTIONARY Tables

The special libref DICTIONARY contains tables that can only be used in SQL queries. These tables list objects in the SAS environment:

DICTIONARY.OPTIONS	System options
DICTIONARY.TITLES	Title and footnote lines
DICTIONARY.EXTFILES	Filerefs
DICTIONARY.MEMBERS	SAS files
DICTIONARY.CATALOGS	Catalogs
DICTIONARY.MACROS	Macros
DICTIONARY.TABLES	SAS data files
DICTIONARY.VIEWS	Views
DICTIONARY.COLUMNS	Variables in SAS datasets
DICTIONARY.INDEXES	Indexes

You can query these tables in a SAS program to get information about objects and settings in the SAS session. For example, this query returns information on the PAGENO system option, including its current value:

```
SELECT * FROM DICTIONARY.OPTIONS WHERE OPTION='PAGENO';
```

```
optname         setting         optdesc                           level
-------------------------------------------------------------------------------
PAGENO          1               Beginning page number for         Portable
                                the next page of output
                                produced by the SAS System
```

To get a list of the columns in a DICTIONARY table, use a `DESCRIBE TABLE` statement, such as:

```
DESCRIBE TABLE DICTIONARY.COLUMNS;
```

Database Management Actions

Other SQL statements are designed to manage data. These statements can create, describe, update, modify, and delete various objects, including tables and views and the indexes and integrity constraints of a table. The following table summarizes the database management actions that are available in SAS SQL.

Action	Object	Statement

Create	Table	CREATE TABLE *table* (*definition*, . . .);
		 CREATE TABLE *table* LIKE *table*;
		 CREATE TABLE *table* AS *query expression*;

	View	CREATE VIEW *view* AS *query expression*;
	Index	CREATE INDEX *index* ON TABLE (*column*, . . .)
	Constraint	ALTER TABLE *table* ADD CONSTRAINT *constraint rule*;
Describe	Table	DESCRIBE TABLE *table*;
	View	DESCRIBE VIEW *view*;
	Constraint	DESCRIBE TABLE CONSTRAINTS *table*;

Update	Table	UPDATE *table* SET *column*=*value*, . . . WHERE *condition*;
		 INSERT INTO *table* SET *column*=*value*, . . . *or* VALUES (*value*, . . .) *or query expression*;
		 DELETE FROM *table* WHERE *condition*;

Modify	Table	ALTER TABLE *table action*;
Delete	Table	DROP TABLE *table*;
	View	DROP VIEW *view*;
	Index	DROP INDEX *index* FROM *table*;
	Constraint	ALTER TABLE *table* DROP CONSTRAINT *constraint*;

Indexes and integrity constraints and the SQL statements for creating them are described in chapter 11, "Options for SAS Datasets."

Describing

The DESCRIBE statement creates a log note that provides a description of a view or table or the constraints of a table. The DESCRIBE VIEW statement shows the query program that is stored in a view, with a log note similar to the one shown in this example:

```
DESCRIBE VIEW SASHELP.VTABLE;
```

```
NOTE: SQL view SASHELP.VTABLE is defined as:

        select *
          from DICTIONARY.TABLES;
```

The DESCRIBE TABLE generates a log note in the form of a CREATE TABLE statement that might have originally defined the table. This is an example:

```
DESCRIBE TABLE RIAA.YEAREND;
```

```
NOTE: SQL table RIAA.YEAREND was created like:

create table RIAA.YEAREND( bufsize=4096 )
  (
   YEAR num,
   CD num,
   CASSETTE num,
    LP_EP num,
    SINGLE num
  );
```

For a table that has constraints, the DESCRIBE TABLE and DESCRIBE TABLE CONSTRAINTS statements generate the same "Alphabetic List of Integrity Constraints" that the CONTENTS proc generates.

Creating a Table

The CREATE TABLE statement creates a new table. Define the columns for the new table in a list of column definitions in parentheses after the table name in the CREATE TABLE statement:

CREATE TABLE *table* (*column definition*, . . .);

This form of the CREATE TABLE statement creates a table with no rows. You can then add rows to the table using other SQL statements.

This example creates the table MAIN.STOCK with the columns SYMBOL, DATE, and CLOSE:

```
CREATE TABLE MAIN.STOCK (SYMBOL CHAR(8), DATE DATE, CLOSE NUM);
```

```
NOTE: Table MAIN.STOCK created, with 0 rows and 3 columns.
```

Use the DESCRIBE TABLE statement with existing tables, as shown above, to see more examples of this kind of CREATE TABLE statement.

A column definition consists of the column name, its data type, and any column modifiers. Column modifiers declare attributes such as the format and label, as described earlier in this chapter.

The SAS data types are character and numeric. SQL syntax also provides several other data types, but all the data types are actually stored in SAS files as the numeric and character data types. Some data types use arguments to set the width of the value. The following table lists the data types that are available in SAS SQL.

SAS SQL Data Types

SQL Data Type	Aliases	Arguments[1]	SAS Data Type
REAL	DOUBLE PRECISION		Numeric
DECIMAL	NUM NUMERIC DEC FLOAT	(*width*, *decimal*)	Numeric
INTEGER	INT SMALLINT		Numeric
DATE			Numeric
CHARACTER	CHAR VARCHAR	(*width*)	Character

[1] The arguments are optional. The default width of a character column is 8.

To create a new table that has the same columns as an existing table, name the existing table in a LIKE clause. For example, this statement creates the table MAIN.DEST that has the same columns as the table MAIN.ORIGIN:

```
CREATE TABLE MAIN.DEST LIKE MAIN.ORIGIN;
```

To create a table that contains rows when you create it, use the AS clause of the `CREATE TABLE` statement, as described earlier in this chapter. The `CREATE TABLE` statement with an AS clause creates a new table with the results of a query expression.

Modifying and Updating a Table

You can change the data and structure of an existing table by adding, modifying, and deleting rows and columns.

Use the `ALTER TABLE` statement for actions on columns. The `ALTER TABLE` statement indicates the table name followed by the details of an action on the table. An ADD clause contains column definitions to add columns to the table. For example, this statement adds the numeric columns FORECAST and ERROR to the table CORP.REVENUE:

```
ALTER TABLE CORP.REVENUE
  ADD
  FORECAST NUMERIC FORMAT=COMMA14.,
  ERROR NUMERIC FORMAT=COMMA14.2;
```

```
NOTE: Table CORP.REVENUE has been modified, with 11 columns.
```

Similarly, a MODIFY clause contains column definitions with column modifiers to apply new attributes to existing columns.

A DROP clause contains a list of columns to remove from the table. This `ALTER TABLE` statement deletes the columns CENTER and REGION from the table CORP.REVENUE:

```
ALTER TABLE CORP.REVENUE
  DROP CENTER, REGION;
```

```
NOTE: Table CORP.REVENUE has been modified, with 9 columns.
```

The INSERT statement adds rows to a table. It can also be used to add rows to some kinds of SQL views. To add rows with specific values, use the VALUES clause in the INSERT statement with a list of values in parentheses. The example below adds a row to the table MAIN.STOCK, which was defined in an earlier example with the columns SYMBOL, DATE, and CLOSE. In the new row, SYMBOL has a value of `'CPB'`, DATE has a value of `'30DEC1994'D`, and CLOSE has a value of 21.07.

```
INSERT INTO MAIN.STOCK VALUES ('CPB', '30DEC1994'D, 21.07);
```

```
NOTE: 1 row was inserted into MAIN.STOCK.
```

You can list selected columns of the table after the table name in the INSERT statement. List the values in the same order in the VALUES clause. This example adds another row to MAIN.STOCK:

```
INSERT INTO MAIN.STOCK (SYMBOL, DATE, CLOSE)
   VALUES ('CPB', '31DEC1999'D, 38.69);
```

```
NOTE: 1 row was inserted into MAIN.STOCK.
```

Any columns that are not listed get null values in the new rows of the table.

Use multiple VALUES clauses to add multiple rows. For example:

```
INSERT INTO MAIN.STOCK (SYMBOL, DATE, CLOSE)
   VALUES ('AOL', '30DEC1994'D, 0.88)
   VALUES ('AOL', '31DEC1999'D, 75.88)
   VALUES ('TWX', '30DEC1994'D, 17.56)
   VALUES ('TWX', '31DEC1999'D, 72.31)
   ;
```

```
NOTE: 4 rows were inserted into MAIN.STOCK.
```

To add existing data to a table, use a query expression in the INSERT statement. Write the query expression so that its columns are in the same order as the columns of the table or the columns listed in the INSERT statement. This is an example:

```
INSERT INTO MAIN.STOCK SELECT SYMBOL, DATE(), CLOSE FROM MAIN.DAILY;
```

Another way to add rows to a table is with the SET clause. In a SET clause, each column is listed with an equals sign and a value for the column, much like an assignment statement. Any columns that are not listed get null values in the new row. Use multiple SET clauses to add multiple rows. This example adds four rows to a table:

```
INSERT INTO MAIN.STOCK
   SET SYMBOL='HGSI', DATE='30DEC1994'D, CLOSE=7.38
   SET SYMBOL='HGSI', DATE='31DEC1999'D, CLOSE=76.31
   SET SYMBOL='CRA', DATE='28APR1999'D, CLOSE=12.50
   SET SYMBOL='CRA', DATE='31DEC1999'D, CLOSE=74.50
   ;
```

The UPDATE statement modifies existing values in a table. It uses a SET clause written the same way as the SET clause of the INSERT statement. Usually, the UPDATE statement includes a WHERE clause so that changes are made in one specific row or a selected set of rows. The new values of the SET clause are applied to all rows that meet the WHERE condition. Columns that are not listed in the SET clause are not changed. If there is no WHERE clause, the new values are applied to every row in the table.

This example changes one specific value to another in one column of a table:

```
UPDATE CORP.SOURCE SET VENDOR='Time Warner Inc.'
   WHERE VENDOR = 'Warner Communications Corp.';
```

```
NOTE: 4 rows were updated in CORP.SOURCE.
```

The DELETE statement removes rows from a table. A WHERE clause identifies the rows to delete. Without a WHERE clause, the DELETE statement removes all rows from a table. This statement removes from the table MAIN.ACTIVE any rows for which the value of EXPIR is earlier than the current date returned by the DATE function:

```
DELETE FROM MAIN.ACTIVE WHERE EXPIR < DATE();
```

```
NOTE: 5 rows were deleted from MAIN.ACTIVE.
```

Deleting

Use the DROP statement to delete a table, view, or index. To delete tables, list them in the `DROP TABLE` statement, for example:

```
DROP TABLE WORK.TEMP1, WORK.TEMP2, WORK.TEMP3;
```

```
NOTE: Table WORK.TEMP1 has been dropped.
NOTE: Table WORK.TEMP2 has been dropped.
NOTE: Table WORK.TEMP3 has been dropped.
```

To delete views, list them in the `DROP VIEW` statement. To delete indexes from a table, list the indexes in the `DROP INDEX` statement followed by a FROM clause to identify the table. For example, this statement removes three indexes from the table CORP.CENTURY:

```
DROP INDEX PRIORITY, CONT, START FROM CORP.CENTURY;
```

```
NOTE: Index PRIORITY has been dropped.
NOTE: Index CONT has been dropped.
NOTE: Index START has been dropped.
```

Database Connections

The statements described here act on SAS files. It is also possible to take actions on an external database in the `PROC SQL` step. This is part of the SQL pass-through feature of SAS/ACCESS. The CONNECT statement establishes a connection to a specific database; the EXECUTE statement passes an SQL statement to the database for execution; the `CONNECTION TO` clause in a query passes a query to the database; and the DISCONNECT statement ends the connection to the database.

SQL Execution

SQL performance cannot be easily predicted from the appearance of the SQL statements. You get a more accurate idea by considering the processing that a statement requires. Options in the SQL proc control details of its actions and can help you deal with the performance issues of SQL.

Engine Requirements and Performance Issues

When SAS data files are used as SQL tables, they are still subject to the usual limits and considerations that apply to SAS data files. The file's storage device and its library engine must support the specific kind of action that is requested in an SQL statement. For example, you must have write access to a library to execute the `CREATE TABLE` and `DROP TABLE` statements; the engine must support update access to execute the UPDATE and DELETE statements. Some view engines support update access; statements such as UPDATE and DELETE can be executed for those views, but not for other views, such as data step views.

The speed of SQL actions tends to be similar to that of other proc steps and data steps doing similar things. For example, when you sort a table with an `ORDER BY` clause, that uses approximately the same computer resources as sorting with the SORT proc; forming groups with a `GROUP BY` clause is comparable to forming them with a CLASS statement in another proc step. Adding and dropping columns in a large table can be a substantial task, because the SAS supervisor has to completely rewrite the table, the same way it would if you used a data step to add or remove a variable.

Table joins are a special area of concern in SQL programming, whether in the SAS environment or elsewhere. An SQL table join produces, at least in theory, every combination of rows of the tables (an effect often described as a *Cartesian product*). It then reduces the number of resulting rows by applying the relevant conditions from the WHERE and HAVING clauses. The number of rows produced in a join is the product of the number of rows of each of the tables. For two large tables or for four or more small tables, this can be a very large number of rows. For example, if two tables each have one million rows, joining the two tables generates one trillion rows. The same is true when you join four tables that each have 1,000 rows, or six tables that each have 100 rows. Table joins on this kind of scale can take a very long time to execute, or they might execute quickly, depending on other details of the processing that should be carefully considered. Similar issues may arise when a query contains subqueries or views.

The processing time for a query is not necessarily proportional to the number of rows read or generated, because the SAS supervisor optimizes queries to eliminate some unnecessary work. Joins can sometimes execute much faster when the WHERE conditions are written in a certain way and the appropriate indexes exist for the tables. Investigate these issues if a query takes too long to run. Several performance options of the SQL proc can reduce the risk that a poorly designed query may accidentally run for a very long time.

Options

Options for the `PROC SQL` step can be initialized in the `PROC SQL` statement. They can be changed in the RESET statement, which can appear between any two SQL statements in the step. The following table describes the options.

Option	Description
EXEC	Executes SQL statements.
NOEXEC	Checks the syntax of SQL statements, but does not execute them.
PRINT	Prints the results of SELECT statements.
NOPRINT	Executes SELECT statements, but does not print the results. This can be useful when you use a SELECT statement to assign the results of a query to macro variables.
NUMBER	Prints a column called Row that contains row numbers.
NONUMBER	Removes the Row column.
DOUBLE	Writes blank lines between rows in the print output.
NODOUBLE	Does not write blank lines between rows.
FLOW=*width*	Sets the width of character columns and flows longer character values on multiple lines.
FLOW=*min max*	Sets the minimum and maximum width of character columns. The SAS supervisor adjusts the column widths to make effective use of the width of the page.
FLOW	Equivalent to FLOW=12 200.
NOFLOW	Writes long character values consecutively, without flowing them.
FEEDBACK	Shows log messages with the query code that results when the SQL interpreter expands view references and wild-card references in a query.
NOFEEDBACK	Does not show expanded query code.
SORTMSG	Generates log messages about sort operations.
NOSORTMSG	Does not show messages about sort operations.
SORTSEQ=*collating sequence*	The collating sequence for sorting.
DQUOTE=ANSI	Treats text in double quotes as names, following the ANSI standards for SQL syntax.
DQUOTE=SAS	Treats text in double quotes as character values, following the rules of SAS syntax.
ERRORSTOP	Stops executing SQL statements after an error occurs.
NOERRORSTOP	Continues to execute SQL statements after an error occurs.
LOOPS=*n*	Limits the number of loop iterations in the execution of a query. Use this option especially for untested query expressions to limit the computer time and resources that an improperly constructed query might use.
INOBS=*n*	Limits the number of input rows from a table. This option is especially useful for debugging and testing queries.
OUTOBS=*n*	Limits the number of output rows from a query.

continued

Option	Description
PROMPT	Prompts the interactive user with the option to continue or stop when a query reaches the limit of the LOOPS=, INOBS=, or OUTOBS= option.
NOPROMPT	Stops executing when a query reaches the limit of the LOOPS=, INOBS=, or OUTOBS= option.
STIMER	Writes performance statistics in the log for each SQL statement. This requires the STIMER system option.
NOSTIMER	Writes performance statistics in the log for the SQL step as a whole. This requires the STIMER system option.

Checking for Errors

The VALIDATE statement lets you check the syntax of a query expression without executing it:

```
VALIDATE query expression;
```

SQL statements also generate several automatic macro variables that help you keep track of SQL performance and errors. The macro variable SQLOBS indicates the number of rows generated by a query or otherwise processed by an SQL statement. The macro variable SQLOOPS counts the number of row iterations required to execute a query.

The macro variable SQLRC is an error code. A value of 0 indicates that the SQL statement completed successfully. A positive value indicates a problem. The following table describes the various possible values of SQLRC.

Value	Meaning
0	The statement completed successfully.
4	There was something questionable about the statement. A warning message was issued.
8	Execution stopped because the statement contained an error.
12	There was a bug in the SQL interpreter.
16	The statement used data objects incorrectly.
24	Execution stopped because of an operating system failure.
28	There was a bug in SQL execution.

This example shows the value of SQLRC after an incorrect coded query:

```
SELECT COUNT(*);
%PUT &SQLRC;
```

```
8
```

Two more automatic macro variables, SQLXRC and SQLXMSG, contain error codes and messages from the SQL pass-through facility.