## Contents

*About This Book* ................................................................. vii

*What's New in the SAS 9.4 SQL Procedure* ........................ xiii

### PART 1 Using the SQL Procedure 1

**Chapter 1 • Introduction to the SQL Procedure** .......................... 3
- What Is SQL? .......................................................... 3
- What Is the SQL Procedure? ........................................... 3
- Terminology .............................................................. 4
- Comparing PROC SQL with the SAS DATA Step ....................... 5
- Notes about the Example Tables ........................................ 7

**Chapter 2 • Retrieving Data from a Single Table** .................... 19
- Overview of the SELECT Statement .................................. 20
- Selecting Columns in a Table ........................................ 22
- Creating New Columns ................................................ 27
- Sorting Data ............................................................ 37
- Retrieving Rows That Satisfy a Condition ............................ 44
- Summarizing Data ....................................................... 56
- Grouping Data ........................................................ 64
- Filtering Grouped Data ................................................ 69
- Validating a Query ...................................................... 71

**Chapter 3 • Retrieving Data from Multiple Tables** .................. 73
- Introduction ............................................................... 73
- Selecting Data from More Than One Table By Using Joins .......... 74
- Using Subqueries to Select Data ........................................ 96
- When to Use Joins and Subqueries .................................... 102
- Combining Queries with Set Operators ................................ 103

**Chapter 4 • Creating and Updating Tables and Views** ............... 111
- Introduction ............................................................... 112
- Creating Tables ......................................................... 112
- Inserting Rows into Tables ............................................. 116
- Updating Data Values in a Table ...................................... 120
- Deleting Rows ............................................................ 123
- Altering Columns ........................................................ 123
- Creating an Index ........................................................ 127
- Deleting a Table .......................................................... 128
- Using SQL Procedure Tables in SAS Software ......................... 128
- Creating and Using Integrity Constraints in a Table .................. 129
- Creating and Using PROC SQL Views ................................ 131

**Chapter 5 • Programming with the SQL Procedure** .................. 137
- Introduction ............................................................... 138
- Using PROC SQL Options to Create and Debug Queries .......... 138
- Improving Query Performance ......................................... 142
Chapter 6 • Practical Problem-Solving with PROC SQL

Overview .................................................. 178
Computing a Weighted Average ...................... 178
Comparing Tables ........................................... 180
Overlaying Missing Data Values ...................... 182
Computing Percentages within Subtotals .......... 185
Counting Duplicate Rows in a Table ............... 187
Expanding Hierarchical Data in a Table .......... 189
Summarizing Data in Multiple Columns .......... 192
Creating a Summary Report .......................... 194
Creating a Customized Sort Order .................. 197
Conditionally Updating a Table ...................... 200
Updating a Table with Values from Another Table 203
Creating and Using Macro Variables ............... 205
Using PROC SQL Tables in Other SAS Procedures 208

PART 2  SQL Procedure Reference 213

Chapter 7 • SQL Procedure ............................. 215
Overview ................................................. 216
Syntax: SQL Procedure ................................ 218
Examples: SQL Procedure ............................. 269

Chapter 8 • SQL Procedure Components .......... 315
Overview ................................................. 315
Dictionary ................................................ 316

PART 3  Appendixes 371

Appendix 1 • SQL Macro Variables and System Options ...
Dictionary ................................................ 373

Appendix 2 • PROC SQL and the ANSI Standard .... 387

Appendix 3 • Example Code Shown in Using the SQL Procedure .... 393

Appendix 4 • Data Sets for Examples in SQL Procedure Reference ....
Overview .................................................. 439
Employees ............................................... 440
Houses .................................................... 440
Match_11 .................................................. 440
Proclib.Delay ............................................ 442
Proclib.Houses .......................................... 443
Contents

Proclib.March ......................................................... 443
Proclib.Paylist2 ...................................................... 444
Proclib.Payroll ........................................................ 445
Proclib.Payroll2 ....................................................... 448
Proclib.Schedule2 .................................................... 448
Proclib.Staff ............................................................ 448
Proclib.Staff2 .......................................................... 451
Proclib.Superv2 ....................................................... 452
Stores ................................................................. 452
Survey ................................................................. 452

Recommended Reading ........................................... 453
Glossary ............................................................... 455
Index ................................................................. 459
Contents
About This Book

Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

SAS uses standard conventions in the documentation of syntax for SAS language elements. These conventions enable you to easily identify the components of SAS syntax. The conventions can be divided into these parts:

• syntax components
• style conventions
• special characters
• references to SAS libraries and external files

Syntax Components

The components of the syntax for most language elements include a keyword and arguments. For some language elements, only a keyword is necessary. For other language elements, the keyword is followed by an equal sign (=). The syntax for arguments has multiple forms in order to demonstrate the syntax of multiple arguments, with and without punctuation.

keyword

specifies the name of the SAS language element that you use when you write your program. Keyword is a literal that is usually the first word in the syntax. In a CALL routine, the first two words are keywords.

In these examples of SAS syntax, the keywords are bold:

CHAR (string, position)
CALL RANBIN (seed, n, p, x);
ALTER (alter-password)
BEST w.
REMOVE <data-set-name>

In this example, the first two words of the CALL routine are the keywords:

CALL RANBIN(seed, n, p, x)

The syntax of some SAS statements consists of a single keyword without arguments:

DO;
SAS code

END;

Some system options require that one of two keyword values be specified:

DUPLIC | NODUPLIC

Some procedure statements have multiple keywords throughout the statement syntax:

CREATE <UNIQUE> INDEX index-name ON table-name (column-1 <, column-2, …>)

argument
specifies a numeric or character constant, variable, or expression. Arguments follow the keyword or an equal sign after the keyword. The arguments are used by SAS to process the language element. Arguments can be required or optional. In the syntax, optional arguments are enclosed in angle brackets ( < > ).

In this example, string and position follow the keyword CHAR. These arguments are required arguments for the CHAR function:

CHAR (string, position)

Each argument has a value. In this example of SAS code, the argument string has a value of ‘summer’, and the argument position has a value of 4:

x=char('summer', 4);

In this example, string and substring are required arguments, whereas modifiers and startpos are optional.

FIND(string, substring <, modifiers> <, startpos>

argument(s)
specifies that one argument is required and that multiple arguments are allowed. Separate arguments with a space. Punctuation, such as a comma ( , ) is not required between arguments.

The MISSING statement is an example of this form of multiple arguments:

MISSING character(s);

<LITERAL_ARGUMENT> argument-1 <<LITERAL_ARGUMENT> argument-2 … >
specifies that one argument is required and that a literal argument can be associated with the argument. You can specify multiple literals and argument pairs. No punctuation is required between the literal and argument pairs. The ellipsis (...) indicates that additional literals and arguments are allowed.

The BY statement is an example of this argument:

BY <DESCENDING> variable-1 <DESCENDING> variable-2 …;

argument-1 <option(s)> <argument-2 <option(s)> …>
specifies that one argument is required and that one or more options can be associated with the argument. You can specify multiple arguments and associated options. No punctuation is required between the argument and the option. The ellipsis (...) indicates that additional arguments with an associated option are allowed.

The FORMAT procedure PICTURE statement is an example of this form of multiple arguments:

PICTURE name <(format-option(s))>
<value-range-set-1 <(picture-1-option(s))>
<value-range-set-2 <(picture-2-option(s))> …>>;
argument-1=value-1 <argument-2=value-2 ...>
specifies that the argument must be assigned a value and that you can specify multiple arguments. The ellipsis (...) indicates that additional arguments are allowed. No punctuation is required between arguments.

The LABEL statement is an example of this form of multiple arguments:

\[\text{LABEL variable-1=label-1 <variable-2=label-2 ...>;}\]

argument-1 <, argument-2, ...>
specifies that one argument is required and that you can specify multiple arguments that are separated by a comma or other punctuation. The ellipsis (...) indicates a continuation of the arguments, separated by a comma. Both forms are used in the SAS documentation.

Here are examples of this form of multiple arguments:

\[\text{AUTHPROVIDERDOMAIN (provider-1:domain-1 <, provider-2:domain-2, ...> INTO :macro-variable-specification-1 <, :macro-variable-specification-2, ...>);}\]

Note: In most cases, example code in SAS documentation is written in lowercase with a monospace font. You can use uppercase, lowercase, or mixed case in the code that you write.

**Style Conventions**

The style conventions that are used in documenting SAS syntax include uppercase bold, uppercase, and italic:

**UPPERCASE BOLD** identifies SAS keywords such as the names of functions or statements. In this example, the keyword ERROR is written in uppercase bold:

\[\text{ERROR <message>;}\]

**UPPERCASE** identifies arguments that are literals.

In this example of the CMPMODEL= system option, the literals include BOTH, CATALOG, and XML:

\[\text{CMPMODEL=BO\textsc{th} | CATALOG | XML |}\]

**italic** identifies arguments or values that you supply. Items in italic represent user-supplied values that are either one of the following:

- nonliteral arguments. In this example of the LINK statement, the argument label is a user-supplied value and therefore appears in italic:

\[\text{LINK label;}\]

- nonliteral values that are assigned to an argument.

In this example of the FORMAT statement, the argument DEFAULT is assigned the variable default-format:

\[\text{FORMAT variable(s) <format > <DEFAULT = default-format>;}\]

**Special Characters**

The syntax of SAS language elements can contain the following special characters:
an equal sign identifies a value for a literal in some language elements such as system options.

In this example of the MAPS system option, the equal sign sets the value of MAPS:

\[ \text{MAPS} = \text{location-of-maps} \]

angle brackets identify optional arguments. A required argument is not enclosed in angle brackets.

In this example of the CAT function, at least one item is required:

\[ \text{CAT} (\text{item-1}, \text{item-2}, \ldots) \]

a vertical bar indicates that you can choose one value from a group of values. Values that are separated by the vertical bar are mutually exclusive.

In this example of the CMPMODEL= system option, you can choose only one of the arguments:

\[ \text{CMPMODEL} = \text{BOTH} | \text{CATALOG} | \text{XML} \]

an ellipsis indicates that the argument can be repeated. If an argument and the ellipsis are enclosed in angle brackets, then the argument is optional. The repeated argument must contain punctuation if it appears before or after the argument.

In this example of the CAT function, multiple \textit{item} arguments are allowed, and they must be separated by a comma:

\[ \text{CAT} (\text{item-1}, \text{item-2}, \ldots) \]

'value' or "value"

indicates that an argument that is enclosed in single or double quotation marks must have a value that is also enclosed in single or double quotation marks.

In this example of the FOOTNOTE statement, the argument \textit{text} is enclosed in quotation marks:

\[ \text{FOOTNOTE} <n> <\text{ods-format-options} \text{ 'text'} | \text{"text"}>; \]

a semicolon indicates the end of a statement or CALL routine.

In this example, each statement ends with a semicolon:

```
data namegame;
  length color name $8;
  color = 'black';
  name = 'jack';
  game = trim(color) || name;
run;
```

**References to SAS Libraries and External Files**

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks. If you use a logical name, you typically have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the reference.
Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

In the examples that use external files, SAS documentation uses the italicized phrase *file-specification*. In the examples that use SAS libraries, SAS documentation uses the italicized phrase *SAS-library* enclosed in quotation marks:

```
infile file-specification obs = 100;
libname libref 'SAS-library';
```
About This Book
What's New in the SAS 9.4 SQL Procedure

Overview

Here are the new features and enhancements:

• additional SAS SQL system option and PROC SQL statement options
• support added for linguistic collation
• support added for extended attributes in DICTIONARY tables
• replaced BUFFERSIZE PROC SQL statement option
• support added for the SQLXOPENERRS macro variable

Additional SAS SQL System Option and PROC SQL Statement Options

The following SQL options have been added to enable PROC SQL to terminate an SQL query if an implicit pass-through request fails:

• SQLIPONEATTEMPT system option
• In SAS 9.4M3, DICTDIAG | NODICTDIAG PROC SQL statement options

For more information, see “SQLIPONEATTEMPT System Option” on page 377 and “PROC SQL Statement” on page 221.

Support Added for Linguistic Collation

Support has been added for linguistic collation with the SORTSEQ statement option. For more information, see “SORTSEQ=sort-table | LINGUISTIC” on page 229.
Support Added for Extended Attributes in DICTIONARY Tables

Support has been added to copy extended attribute information in DICTIONARY tables. The XATTRS DICTIONARY table and Vxattr Sashelp view have been added. For more information, see “Accessing SAS System Information By Using DICTIONARY Tables” on page 149.

Replaced BUFFERSIZE PROC SQL Statement Option

The BUFFERSIZE option for the PROC SQL statement has been replaced with the UBUFSIZE option. The BUFFERSIZE option that was used prior to SAS 9.4 works the same as the UBUFSIZE option and is still supported. As of SAS 9.4, UBUFSIZE is the preferred option. For more information, see “PROC SQL Statement” on page 221.

Support Added for the SQLXOPENERRS Macro Variable

SAS 9.4M3 adds the SQLXOPENERRS macro variable, which contains the number of DICTIONARY tables that failed to open. For more information, see “SQLXOPENERRS” on page 163.
Part 1

Using the SQL Procedure

Chapter 1
Introduction to the SQL Procedure ........................................ 3

Chapter 2
Retrieving Data from a Single Table ........................................ 19

Chapter 3
Retrieving Data from Multiple Tables ................................. 73

Chapter 4
Creating and Updating Tables and Views ............................. 111

Chapter 5
Programming with the SQL Procedure ................................. 137

Chapter 6
Practical Problem-Solving with PROC SQL ............................. 177
Chapter 1
Introduction to the SQL Procedure

What Is SQL?

Structured Query Language (SQL) is a standardized, widely used language that retrieves and updates data in relational tables and databases.

A relation is a mathematical concept that is similar to the mathematical concept of a set. Relations are represented physically as two-dimensional tables that are arranged in rows and columns. Relational theory was developed by E. F. Codd, an IBM researcher, and first implemented at IBM in a prototype called System R. This prototype evolved into commercial IBM products based on SQL. The Structured Query Language is now in the public domain and is part of many vendors' products.

What Is the SQL Procedure?

The SQL procedure is the Base SAS implementation of Structured Query Language. PROC SQL is part of Base SAS software, and you can use it with any SAS data set (table). Often, PROC SQL can be an alternative to other SAS procedures or the DATA step. You can use SAS language elements such as global statements, data set options, functions, informats, and formats with PROC SQL just as you can with other SAS procedures. PROC SQL enables you to perform the following tasks:

- generate reports
- generate summary statistics
• retrieve data from tables or views
• combine data from tables or views
• create tables, views, and indexes
• update the data values in PROC SQL tables
• update and retrieve data from database management system (DBMS) tables
• modify a PROC SQL table by adding, modifying, or dropping columns

PROC SQL can be used in an interactive SAS session or within batch programs, and it can include global statements, such as TITLE and OPTIONS.

---

**Terminology**

**Tables**

A PROC SQL table is the same as a SAS data file. It is a SAS file of type DATA. PROC SQL tables consist of rows and columns. The rows correspond to observations in SAS data files, and the columns correspond to variables. The following table lists equivalent terms that are used in SQL, SAS, and traditional data processing.

**Table 1.1 Comparing Equivalent Terms**

<table>
<thead>
<tr>
<th>SQL Term</th>
<th>SAS Term</th>
<th>Data Processing Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>SAS data file</td>
<td>file</td>
</tr>
<tr>
<td>row</td>
<td>observation</td>
<td>record</td>
</tr>
<tr>
<td>column</td>
<td>variable</td>
<td>field</td>
</tr>
</tbody>
</table>

You can create and modify tables by using the SAS DATA step, or by using the PROC SQL statements that are described in Chapter 4, “Creating and Updating Tables and Views,” on page 111. Other SAS procedures and the DATA step can read and update tables that are created with PROC SQL.

SAS data files can have a one-level name or a two-level name. Typically, the names of temporary SAS data files have only one level, and the data files are stored in the Work library. PROC SQL assumes that SAS data files that are specified with a one-level name are to be read from or written to the Work library, unless you specify a User library. You can assign a User library with a LIBNAME statement or with the SAS system option USER=. For more information about how to work with SAS data files and libraries, see “Temporary and Permanent SAS Data Sets” in Base SAS Procedures Guide.

DBMS tables are tables that were created with other software vendors' database management systems. PROC SQL can connect to, update, and modify DBMS tables, with some restrictions. For more information, see “Accessing a DBMS with SAS/ACCESS Software” on page 167.
Queries

Queries retrieve data from a table, view, or DBMS. A query returns a query result, which consists of rows and columns from a table. With PROC SQL, you use a SELECT statement and its subordinate clauses to form a query. Chapter 2, “Retrieving Data from a Single Table,” on page 19 describes how to build a query.

Views

PROC SQL views do not actually contain data as tables do. Rather, a PROC SQL view contains a stored SELECT statement or query. The query executes when you use the view in a SAS procedure or DATA step. When a view executes, it displays data that is derived from existing tables, from other views, or from SAS/ACCESS views. Other SAS procedures and the DATA step can use a PROC SQL view as they would any SAS data file. For more information about views, see Chapter 4, “Creating and Updating Tables and Views,” on page 111.

Note: When you process PROC SQL views between a client and a server, getting the correct results depends on the compatibility between the client and server architecture. For more information, see “Access a SAS View” in SAS/CONNECT User’s Guide.

Null Values

According to the ANSI standard for SQL, a missing value is called a null value. It is not the same as a blank or zero value. However, to be compatible with the rest of SAS, PROC SQL treats missing values the same as blanks or zero values and considers all three to be null values. This important concept comes up in several places in this document.

Comparing PROC SQL with the SAS DATA Step

PROC SQL can perform some of the operations that are provided by the DATA step and the PRINT, SORT, and SUMMARY procedures. The following query displays the total population of all the large countries (countries with population greater than 1 million) on each continent.

```sql
PROC SQL;
  title 'Population of Large Countries Grouped by Continent';
  select Continent, sum(Population) as TotPop format=comma15.
    from sql.countries
    where Population gt 1000000
    group by Continent
    order by TotPop;
QUIT;
```
Output 1.1 Sample SQL Output

Here is a SAS program that produces the same result.

```sas
title 'Large Countries Grouped by Continent';
proc summary data=sql.countries;
   where Population > 1000000;
   class Continent;
   var Population;
   output out=sumPop sum=TotPop;
run;

proc sort data=SumPop;
   by totPop;
run;

proc print data=SumPop noobs;
   var Continent TotPop;
   format TotPop comma15.;
   where _type_=_1;
run;
```
Output 1.2  Sample DATA Step Output

Large Countries Grouped by Continent

<table>
<thead>
<tr>
<th>Continent</th>
<th>TotPop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oceania</td>
<td>3,422,548</td>
</tr>
<tr>
<td>Australia</td>
<td>18,256,944</td>
</tr>
<tr>
<td>Central America and Caribbean</td>
<td>65,283,910</td>
</tr>
<tr>
<td>South America</td>
<td>316,303,397</td>
</tr>
<tr>
<td>North America</td>
<td>384,801,818</td>
</tr>
<tr>
<td>Africa</td>
<td>706,511,183</td>
</tr>
<tr>
<td>Europe</td>
<td>811,680,062</td>
</tr>
<tr>
<td>Asia</td>
<td>3,379,469,468</td>
</tr>
</tbody>
</table>

This example shows that PROC SQL can achieve the same results as Base SAS software but often with fewer and shorter statements. The SELECT statement that is shown in this example performs summation, grouping, sorting, and row selection. It also displays the query's results without the PRINT procedure.

PROC SQL executes without using the RUN statement. After you invoke PROC SQL, you can submit additional SQL procedure statements without submitting the PROC statement again. Use the QUIT statement to terminate the procedure.

Notes about the Example Tables

For all examples, the following global statement is in effect:

```plaintext
libname sql 'SAS-library';
```

The tables that are used in the whole section of “Using the SQL Procedure” contain geographic and demographic data. The data is intended to be used for the PROC SQL code examples only. The data is not necessarily up-to-date or accurate.

These tables are available in a ZIP file at http://support.sas.com/documentation/onlinedoc/base/index.html. Look for SAS SQL Procedure User’s Guide on this page. Download and extract the ZIP file to a location that is accessible by SAS. The extracted ZIP file is a transport file that contains the tables.

Start SAS. Submit this code to import the tables:

```plaintext
/* Substitute the pathname of your Sasuser directory for 'your-Sasuser-directory-path' and the appropriate pathname and filename for 'your-downloaded-file-location' */

libname new 'your-Sasuser-directory-path';
filename trans 'your-downloaded-file-location';
```
proc cimport library=new infile=trans;
run;

/* Assign a libref named SQL to provide access to some
of the sample data sets. */
libname sql 'your-Sasuser-directory-path';

The tables that follow show partial output for these data sets.

You can find all of the example SQL programs that are shown in the “Using the SQL
Procedure” section in Appendix 3, “Example Code Shown in Using the SQL
Procedure,” on page 393. If you copy the code to a SAS editor, you can preserve the
code spacing by copying the HTML version of the code.

The Countries table contains data that pertains to countries. The Area column contains a
country's area in square miles. The UNDate column contains the year in which a country
entered the United Nations, if applicable.

Output 1.3  Countries (Partial Output)

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
<th>Area</th>
<th>Continent</th>
<th>UNDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>Kabul</td>
<td>17070323</td>
<td>251825</td>
<td>Asia</td>
<td>1946</td>
</tr>
<tr>
<td>Albania</td>
<td>Tirane</td>
<td>3407400</td>
<td>11100</td>
<td>Europe</td>
<td>1955</td>
</tr>
<tr>
<td>Algeria</td>
<td>Algiers</td>
<td>28171132</td>
<td>919595</td>
<td>Africa</td>
<td>1962</td>
</tr>
<tr>
<td>Andorra</td>
<td>Andorra la Vell</td>
<td>64634</td>
<td>200</td>
<td>Europe</td>
<td>1993</td>
</tr>
<tr>
<td>Angola</td>
<td>Luanda</td>
<td>9901050</td>
<td>481300</td>
<td>Africa</td>
<td>1976</td>
</tr>
<tr>
<td>Antigua and Barbuda</td>
<td>St. John's</td>
<td>65644</td>
<td>171</td>
<td>Central America</td>
<td>1981</td>
</tr>
<tr>
<td>Argentina</td>
<td>Buenos Aires</td>
<td>34248705</td>
<td>1073518</td>
<td>South America</td>
<td>1945</td>
</tr>
<tr>
<td>Armenia</td>
<td>Yerevan</td>
<td>3565864</td>
<td>11500</td>
<td>Asia</td>
<td>1992</td>
</tr>
<tr>
<td>Australia</td>
<td>Canberra</td>
<td>18255944</td>
<td>2966200</td>
<td>Australia</td>
<td>1945</td>
</tr>
<tr>
<td>Austria</td>
<td>Vienna</td>
<td>8033746</td>
<td>32400</td>
<td>Europe</td>
<td>1955</td>
</tr>
<tr>
<td>Azerbaijan</td>
<td>Baku</td>
<td>7760064</td>
<td>33400</td>
<td>Asia</td>
<td>1992</td>
</tr>
<tr>
<td>Bahamas</td>
<td>Nassau</td>
<td>275703</td>
<td>5400</td>
<td>Central America</td>
<td>1973</td>
</tr>
<tr>
<td>Bahrain</td>
<td>Manama</td>
<td>591800</td>
<td>300</td>
<td>Asia</td>
<td>1971</td>
</tr>
<tr>
<td>Bangladesh</td>
<td>Dhaka</td>
<td>1.2639E8</td>
<td>57300</td>
<td>Asia</td>
<td>1974</td>
</tr>
<tr>
<td>Barbados</td>
<td>Bridgetown</td>
<td>258534</td>
<td>200</td>
<td>Central America</td>
<td>1966</td>
</tr>
</tbody>
</table>

The WorldCityCoords table contains latitude and longitude data for world cities. Cities
in the Western hemisphere have negative longitude coordinates. Cities in the Southern
hemisphere have negative latitude coordinates. Coordinates are rounded to the nearest degree.

**Output 1.4  WorldCityCoords (Partial Output)**

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kabul</td>
<td>Afghanistan</td>
<td>35</td>
<td>69</td>
</tr>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>-34</td>
<td>-59</td>
</tr>
<tr>
<td>Cordoba</td>
<td>Argentina</td>
<td>-31</td>
<td>-64</td>
</tr>
<tr>
<td>Tucuman</td>
<td>Argentina</td>
<td>-27</td>
<td>-65</td>
</tr>
<tr>
<td>Adelaide</td>
<td>Australia</td>
<td>-35</td>
<td>138</td>
</tr>
<tr>
<td>Alice Springs</td>
<td>Australia</td>
<td>-24</td>
<td>134</td>
</tr>
<tr>
<td>Brisbane</td>
<td>Australia</td>
<td>-27</td>
<td>153</td>
</tr>
<tr>
<td>Darwin</td>
<td>Australia</td>
<td>-12</td>
<td>131</td>
</tr>
<tr>
<td>Melbourne</td>
<td>Australia</td>
<td>-38</td>
<td>145</td>
</tr>
<tr>
<td>Perth</td>
<td>Australia</td>
<td>-32</td>
<td>116</td>
</tr>
<tr>
<td>Sydney</td>
<td>Australia</td>
<td>-34</td>
<td>151</td>
</tr>
<tr>
<td>Vienna</td>
<td>Austria</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>Nassau</td>
<td>Bahamas</td>
<td>26</td>
<td>-77</td>
</tr>
<tr>
<td>Chittagong</td>
<td>Bangladesh</td>
<td>22</td>
<td>92</td>
</tr>
</tbody>
</table>
The USCityCoords table contains the coordinates for cities in the United States. Because all cities in this table are in the Western hemisphere, all of the longitude coordinates are negative. Coordinates are rounded to the nearest degree.

Output 1.5  USCityCoords (Partial Output)

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td>NY</td>
<td>43</td>
<td>-74</td>
</tr>
<tr>
<td>Albuquerque</td>
<td>NM</td>
<td>36</td>
<td>-106</td>
</tr>
<tr>
<td>Amarillo</td>
<td>TX</td>
<td>35</td>
<td>-102</td>
</tr>
<tr>
<td>Anchorage</td>
<td>AK</td>
<td>61</td>
<td>-150</td>
</tr>
<tr>
<td>Annapolis</td>
<td>MD</td>
<td>39</td>
<td>-77</td>
</tr>
<tr>
<td>Atlanta</td>
<td>GA</td>
<td>34</td>
<td>-84</td>
</tr>
<tr>
<td>Augusta</td>
<td>ME</td>
<td>44</td>
<td>-70</td>
</tr>
<tr>
<td>Austin</td>
<td>TX</td>
<td>30</td>
<td>-98</td>
</tr>
<tr>
<td>Baker</td>
<td>OR</td>
<td>45</td>
<td>-118</td>
</tr>
<tr>
<td>Baltimore</td>
<td>MD</td>
<td>39</td>
<td>-76</td>
</tr>
<tr>
<td>Bangor</td>
<td>ME</td>
<td>45</td>
<td>-69</td>
</tr>
<tr>
<td>Baton Rouge</td>
<td>LA</td>
<td>31</td>
<td>-91</td>
</tr>
<tr>
<td>Birmingham</td>
<td>AL</td>
<td>33</td>
<td>-87</td>
</tr>
<tr>
<td>Bismarck</td>
<td>ND</td>
<td>47</td>
<td>-101</td>
</tr>
<tr>
<td>Boise</td>
<td>ID</td>
<td>43</td>
<td>-116</td>
</tr>
</tbody>
</table>
The United States table contains data that is associated with the states. The Statehood column contains the date on which the state was admitted into the Union.

**Output 1.6  United States (Partial Output)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
<th>Area</th>
<th>Continent</th>
<th>Statehood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>Montgomery</td>
<td>4227437</td>
<td>52423</td>
<td>North America</td>
<td>14DEC1819</td>
</tr>
<tr>
<td>Alaska</td>
<td>Juneau</td>
<td>604929</td>
<td>656400</td>
<td>North America</td>
<td>03JAN1959</td>
</tr>
<tr>
<td>Arizona</td>
<td>Phoenix</td>
<td>3974962</td>
<td>114000</td>
<td>North America</td>
<td>14FEB1912</td>
</tr>
<tr>
<td>Arkansas</td>
<td>Little Rock</td>
<td>2447996</td>
<td>53200</td>
<td>North America</td>
<td>15JUN1836</td>
</tr>
<tr>
<td>California</td>
<td>Sacramento</td>
<td>31518948</td>
<td>163700</td>
<td>North America</td>
<td>09SEP1850</td>
</tr>
<tr>
<td>Colorado</td>
<td>Denver</td>
<td>3601298</td>
<td>104100</td>
<td>North America</td>
<td>01AUG1876</td>
</tr>
<tr>
<td>Connecticut</td>
<td>Hartford</td>
<td>3309742</td>
<td>5500</td>
<td>North America</td>
<td>09JAN1788</td>
</tr>
<tr>
<td>Delaware</td>
<td>Dover</td>
<td>707232</td>
<td>2500</td>
<td>North America</td>
<td>07DEC1787</td>
</tr>
<tr>
<td>District of Columbia</td>
<td>Washington</td>
<td>612907</td>
<td>100</td>
<td>North America</td>
<td>21FEB1871</td>
</tr>
<tr>
<td>Florida</td>
<td>Tallahassee</td>
<td>13814408</td>
<td>65800</td>
<td>North America</td>
<td>03MAR1845</td>
</tr>
<tr>
<td>Georgia</td>
<td>Atlanta</td>
<td>6985572</td>
<td>59400</td>
<td>North America</td>
<td>02JAN1788</td>
</tr>
<tr>
<td>Hawaii</td>
<td>Honolulu</td>
<td>1183198</td>
<td>10900</td>
<td>Oceania</td>
<td>21AUG1959</td>
</tr>
<tr>
<td>Idaho</td>
<td>Boise</td>
<td>1109980</td>
<td>83600</td>
<td>North America</td>
<td>03JUL1890</td>
</tr>
<tr>
<td>Illinois</td>
<td>Springfield</td>
<td>11813091</td>
<td>57900</td>
<td>North America</td>
<td>03DEC1818</td>
</tr>
<tr>
<td>Indiana</td>
<td>Indianapolis</td>
<td>5769553</td>
<td>36400</td>
<td>North America</td>
<td>11DEC1816</td>
</tr>
</tbody>
</table>
The PostalCodes table contains postal code abbreviations.

**Output 1.7  PostalCodes (Partial Output)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>AL</td>
</tr>
<tr>
<td>Alaska</td>
<td>AK</td>
</tr>
<tr>
<td>American Samoa</td>
<td>AS</td>
</tr>
<tr>
<td>Arizona</td>
<td>AZ</td>
</tr>
<tr>
<td>Arkansas</td>
<td>AR</td>
</tr>
<tr>
<td>California</td>
<td>CA</td>
</tr>
<tr>
<td>Colorado</td>
<td>CO</td>
</tr>
<tr>
<td>Connecticut</td>
<td>CT</td>
</tr>
<tr>
<td>Delaware</td>
<td>DE</td>
</tr>
<tr>
<td>District Of Columbia</td>
<td>DC</td>
</tr>
<tr>
<td>Florida</td>
<td>FL</td>
</tr>
<tr>
<td>Georgia</td>
<td>GA</td>
</tr>
<tr>
<td>Guam</td>
<td>GU</td>
</tr>
<tr>
<td>Hawaii</td>
<td>HI</td>
</tr>
<tr>
<td>Idaho</td>
<td>ID</td>
</tr>
</tbody>
</table>
The WorldTemps table contains average high and low temperatures from various international cities.

**Output 1.8  WorldTemps (Partial Output)**

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>AvgHigh</th>
<th>AvgLow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>90</td>
<td>45</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>Netherlands</td>
<td>70</td>
<td>33</td>
</tr>
<tr>
<td>Athens</td>
<td>Greece</td>
<td>89</td>
<td>41</td>
</tr>
<tr>
<td>Auckland</td>
<td>New Zealand</td>
<td>75</td>
<td>44</td>
</tr>
<tr>
<td>Bangkok</td>
<td>Thailand</td>
<td>95</td>
<td>69</td>
</tr>
<tr>
<td>Beijing</td>
<td>China</td>
<td>86</td>
<td>17</td>
</tr>
<tr>
<td>Belgrade</td>
<td>Yugoslavia</td>
<td>80</td>
<td>29</td>
</tr>
<tr>
<td>Berlin</td>
<td>Germany</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>Bogota</td>
<td>Colombia</td>
<td>69</td>
<td>43</td>
</tr>
<tr>
<td>Bombay</td>
<td>India</td>
<td>90</td>
<td>68</td>
</tr>
<tr>
<td>Bucharest</td>
<td>Romania</td>
<td>83</td>
<td>24</td>
</tr>
<tr>
<td>Budapest</td>
<td>Hungary</td>
<td>80</td>
<td>25</td>
</tr>
<tr>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>87</td>
<td>48</td>
</tr>
<tr>
<td>Cairo</td>
<td>Egypt</td>
<td>95</td>
<td>48</td>
</tr>
<tr>
<td>Calcutta</td>
<td>India</td>
<td>97</td>
<td>56</td>
</tr>
</tbody>
</table>
The OilProd table contains oil production statistics from oil-producing countries.

**Output 1.9 OilProd (Partial Output)**

<table>
<thead>
<tr>
<th>Country</th>
<th>BarrelsPerDay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algeria</td>
<td>1,400,000</td>
</tr>
<tr>
<td>Canada</td>
<td>2,500,000</td>
</tr>
<tr>
<td>China</td>
<td>3,000,000</td>
</tr>
<tr>
<td>Egypt</td>
<td>900,000</td>
</tr>
<tr>
<td>Indonesia</td>
<td>1,500,000</td>
</tr>
<tr>
<td>Iran</td>
<td>4,000,000</td>
</tr>
<tr>
<td>Iraq</td>
<td>600,000</td>
</tr>
<tr>
<td>Kuwait</td>
<td>2,500,000</td>
</tr>
<tr>
<td>Libya</td>
<td>1,500,000</td>
</tr>
<tr>
<td>Mexico</td>
<td>3,400,000</td>
</tr>
<tr>
<td>Nigeria</td>
<td>2,000,000</td>
</tr>
<tr>
<td>Norway</td>
<td>3,500,000</td>
</tr>
<tr>
<td>Oman</td>
<td>900,000</td>
</tr>
<tr>
<td>Saudi Arabia</td>
<td>9,000,000</td>
</tr>
<tr>
<td>United States of America</td>
<td>8,000,000</td>
</tr>
</tbody>
</table>
The OilRsrvs table lists approximate oil reserves of oil-producing countries.

**Output 1.10 OilRsrvs (Partial Output)**

<table>
<thead>
<tr>
<th>Country</th>
<th>Barrels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algeria</td>
<td>9,200,000,000</td>
</tr>
<tr>
<td>Canada</td>
<td>7,000,000,000</td>
</tr>
<tr>
<td>China</td>
<td>25,000,000,000</td>
</tr>
<tr>
<td>Egypt</td>
<td>4,000,000,000</td>
</tr>
<tr>
<td>Gabon</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>Indonesia</td>
<td>5,000,000,000</td>
</tr>
<tr>
<td>Iran</td>
<td>90,000,000,000</td>
</tr>
<tr>
<td>Iraq</td>
<td>110,000,000,000</td>
</tr>
<tr>
<td>Kuwait</td>
<td>95,000,000,000</td>
</tr>
<tr>
<td>Libya</td>
<td>30,000,000,000</td>
</tr>
<tr>
<td>Mexico</td>
<td>50,000,000,000</td>
</tr>
<tr>
<td>Nigeria</td>
<td>16,000,000,000</td>
</tr>
<tr>
<td>Norway</td>
<td>11,000,000,000</td>
</tr>
<tr>
<td>Saudi Arabia</td>
<td>260,000,000,000</td>
</tr>
<tr>
<td>United Arab Emirates</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>
The Continents table contains geographic data that relates to world continents.

Output 1.11  Continents

<table>
<thead>
<tr>
<th>Name</th>
<th>Area</th>
<th>HighPoint</th>
<th>Height</th>
<th>LowPoint</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>11506000</td>
<td>Kilimanjaro</td>
<td>19340</td>
<td>Lake Assal</td>
<td>-512</td>
</tr>
<tr>
<td>Antarctica</td>
<td>5500000</td>
<td>Vinson Massif</td>
<td>16860</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Asia</td>
<td>16988000</td>
<td>Everest</td>
<td>29028</td>
<td>Dead Sea</td>
<td>-1302</td>
</tr>
<tr>
<td>Australia</td>
<td>2968000</td>
<td>Kosciusko</td>
<td>7310</td>
<td>Lake Eyre</td>
<td>-52</td>
</tr>
<tr>
<td>Central America</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Europe</td>
<td>3745000</td>
<td>El'brus</td>
<td>18510</td>
<td>Caspian Sea</td>
<td>-92</td>
</tr>
<tr>
<td>North America</td>
<td>9390000</td>
<td>McKinley</td>
<td>20320</td>
<td>Death Valley</td>
<td>-282</td>
</tr>
<tr>
<td>Oceania</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>South America</td>
<td>6795000</td>
<td>Aconcagua</td>
<td>22834</td>
<td>Valdes Peninsul</td>
<td>-131</td>
</tr>
</tbody>
</table>
The Features table contains statistics that describe various types of geographical features, such as oceans, lakes, and mountains.

**Output 1.12  Features (Partial Output)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Location</th>
<th>Area</th>
<th>Height</th>
<th>Depth</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aconcagua</td>
<td>Mountain</td>
<td>Argentina</td>
<td>22834</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon</td>
<td>River</td>
<td>South America</td>
<td>4000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amur</td>
<td>River</td>
<td>Asia</td>
<td>2700</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andaman</td>
<td>Sea</td>
<td></td>
<td>218100</td>
<td>3667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Angel Falls</td>
<td>Waterfall</td>
<td>Venezuela</td>
<td>3212</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Annapurna</td>
<td>Mountain</td>
<td>Nepal</td>
<td>26504</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aral Sea</td>
<td>Lake</td>
<td>Asia</td>
<td>25300</td>
<td>222</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ararat</td>
<td>Mountain</td>
<td>Turkey</td>
<td>16804</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arctic</td>
<td>Ocean</td>
<td></td>
<td>5105700</td>
<td>17880</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atlantic</td>
<td>Ocean</td>
<td></td>
<td>33420000</td>
<td>28374</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baflin</td>
<td>Island</td>
<td>Arctic</td>
<td>183810</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baltic</td>
<td>Sea</td>
<td></td>
<td>146500</td>
<td>180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baykal</td>
<td>Lake</td>
<td>Russia</td>
<td>11780</td>
<td>5315</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bering</td>
<td>Sea</td>
<td></td>
<td>873000</td>
<td>4893</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>Sea</td>
<td></td>
<td>196100</td>
<td>3906</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2
Retrieving Data from a Single Table

Overview of the SELECT Statement ........................................ 20
  How to Use the SELECT Statement ...................................... 20
  SELECT and FROM Clauses .................................................. 20
  WHERE Clause .............................................................. 21
  ORDER BY Clause .......................................................... 21
  GROUP BY Clause ........................................................... 21
  HAVING Clause ............................................................. 21
  Ordering the SELECT Statement ......................................... 22

Selecting Columns in a Table .............................................. 22
  Selecting All Columns in a Table ....................................... 22
  Selecting Specific Columns in a Table ................................ 23
  Eliminating Duplicate Rows from the Query Results .............. 25
  Determining the Structure of a Table .................................. 27

Creating New Columns ...................................................... 27
  Adding Text to Output ................................................... 27
  Calculating Values ....................................................... 29
  Assigning a Column Alias ............................................... 30
  Referring to a Calculated Column by Alias ......................... 31
  Assigning Values Conditionally ....................................... 32
  Replacing Missing Values ............................................... 35
  Specifying Column Attributes ........................................ 36

Sorting Data ..................................................................... 37
  Overview of Sorting Data ................................................ 37
  Sorting by Column .......................................................... 38
  Sorting by Multiple Columns .......................................... 39
  Specifying a Sort Order .................................................. 39
  Sorting by Calculated Column ......................................... 40
  Sorting by Column Position ............................................. 41
  Sorting by Columns That Are Not Selected ......................... 42
  Specifying a Different Sorting Sequence .............................. 43
  Sorting Columns That Contain Missing Values ..................... 43

Retrieving Rows That Satisfy a Condition ............................ 44
  Using a Simple WHERE Clause ......................................... 44
  Retrieving Rows Based on a Comparison ............................... 45
  Retrieving Rows That Satisfy Multiple Conditions ............... 47
  Using Other Conditional Operators .................................. 49
  Using Truncated String Comparison Operators .................... 53
  Using a WHERE Clause with Missing Values ....................... 54
Overview of the SELECT Statement

How to Use the SELECT Statement

This chapter shows you how to perform the following tasks:

• retrieve data from a single table by using the SELECT statement
• validate the correctness of a SELECT statement by using the VALIDATE statement

With the SELECT statement, you can retrieve data from tables or data that is described by SAS data views.

Note: The examples in this chapter retrieve data from tables that are SAS data sets. However, you can use all of the operations that are described here with SAS data views.

The SELECT statement is the primary tool of PROC SQL. You use it to identify, retrieve, and manipulate columns of data from a table. You can also use several optional clauses within the SELECT statement to place restrictions on a query.

SELECT and FROM Clauses

The following simple SELECT statement is sufficient to produce a useful result:

```sql
select Name
from sql.countries;
```

The SELECT statement must contain a SELECT clause and a FROM clause, both of which are required in a PROC SQL query. This SELECT statement contains the following:

• a SELECT clause that lists the Name column
• a FROM clause that lists the table in which the Name column resides

WHERE Clause

The WHERE clause enables you to restrict the data that you retrieve by specifying a condition that each row of the table must satisfy. PROC SQL output includes only those rows that satisfy the condition. The following SELECT statement contains a WHERE clause that restricts the query output to only those countries that have a population that is greater than 5,000,000 people:

```sql
select Name
  from sql.countries
  where Population gt 5000000;
```

ORDER BY Clause

The ORDER BY clause enables you to sort the output from a table by one or more columns. That is, you can put character values in either ascending or descending alphabetical order, and you can put numerical values in either ascending or descending numerical order. The default order is ascending. For example, you can modify the previous example to list the data by descending population:

```sql
select Name
  from sql.countries
  where Population gt 5000000
  order by Population desc;
```

GROUP BY Clause

The GROUP BY clause enables you to break query results into subsets of rows. When you use the GROUP BY clause, you use an aggregate function in the SELECT clause or a HAVING clause to instruct PROC SQL how to group the data. For details about aggregate functions, see “Summarizing Data” on page 56. PROC SQL calculates the aggregate function separately for each group. When you do not use an aggregate function, PROC SQL treats the GROUP BY clause as if it were an ORDER BY clause, and any aggregate functions are applied to the entire table.

The following query uses the SUM function to list the total population of each continent. The GROUP BY clause groups the countries by continent, and the ORDER BY clause puts the continents in alphabetical order:

```sql
select Continent, sum(Population)
  from sql.countries
  group by Continent
  order by Continent;
```

HAVING Clause

The HAVING clause works with the GROUP BY clause to restrict the groups in a query's results based on a given condition. PROC SQL applies the HAVING condition after grouping the data and applying aggregate functions. For example, the following query restricts the groups to include only the continents of Asia and Europe:

```sql
select Continent, sum(Population)
  from sql.countries
  group by Continent
  having Continent in ("Asia", "Europe");
```
ordering the select statement

when you construct a select statement, you must specify the clauses in the following order:

1. select
2. from
3. where
4. group by
5. having
6. order by

note: only the select and from clauses are required.

the proc sql select statement and its clauses are discussed in further detail in the following sections.

selecting columns in a table

when you retrieve data from a table, you can select one or more columns by using variations of the basic select statement.

selecting all columns in a table

use an asterisk in the select clause to select all columns in a table. the following example selects all columns in the sql.uscitycoords table, which contains latitude and longitude values for u.s. cities:

```
libname sql 'SAS-library';

proc sql outobs=12;
  title 'U.S. Cities with Their States and Coordinates';
  select * from sql.uscitycoords;
```

note: the outobs= option limits the number of rows (observations) in the output.
outobs= is similar to the obs= data set option. outobs= is used throughout this document to limit the number of rows that are displayed in examples.

note: in the tables used in these examples, latitude values that are south of the equator are negative. longitude values that are west of the prime meridian are also negative.
Select All Columns in a Table

When you select all columns, PROC SQL displays the columns in the order in which they are stored in the table.

Select Specific Columns in a Table

To select a specific column in a table, list the name of the column in the SELECT clause. The following example selects only the City column in the Sql.USCityCoords table:

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Names of U.S. Cities';
  select City
    from sql.uscitycoords;
```

U.S. Cities with Their States and Coordinates

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td>NY</td>
<td>43</td>
<td>-74</td>
</tr>
<tr>
<td>Albuquerque</td>
<td>NM</td>
<td>36</td>
<td>-106</td>
</tr>
<tr>
<td>Amarillo</td>
<td>TX</td>
<td>35</td>
<td>-102</td>
</tr>
<tr>
<td>Anchorage</td>
<td>AK</td>
<td>61</td>
<td>-150</td>
</tr>
<tr>
<td>Annapolis</td>
<td>MD</td>
<td>39</td>
<td>-77</td>
</tr>
<tr>
<td>Atlanta</td>
<td>GA</td>
<td>34</td>
<td>-84</td>
</tr>
<tr>
<td>Augusta</td>
<td>ME</td>
<td>44</td>
<td>-70</td>
</tr>
<tr>
<td>Austin</td>
<td>TX</td>
<td>30</td>
<td>-98</td>
</tr>
<tr>
<td>Baker</td>
<td>OR</td>
<td>45</td>
<td>-118</td>
</tr>
<tr>
<td>Baltimore</td>
<td>MD</td>
<td>39</td>
<td>-76</td>
</tr>
<tr>
<td>Bangor</td>
<td>ME</td>
<td>45</td>
<td>-69</td>
</tr>
<tr>
<td>Baton Rouge</td>
<td>LA</td>
<td>31</td>
<td>-91</td>
</tr>
</tbody>
</table>

Note: When you select all columns, PROC SQL displays the columns in the order in which they are stored in the table.
If you want to select more than one column, then you must separate the names of the columns with commas, as in this example, which selects the City and State columns in the Sql.USCityCoords table:

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'U.S. Cities and Their States';
  select City, State
  from sql.uscitycoords;
```
Output 2.3  Selecting Multiple Columns

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td>NY</td>
</tr>
<tr>
<td>Albuquerque</td>
<td>NM</td>
</tr>
<tr>
<td>Amarillo</td>
<td>TX</td>
</tr>
<tr>
<td>Anchorage</td>
<td>AK</td>
</tr>
<tr>
<td>Annapolis</td>
<td>MD</td>
</tr>
<tr>
<td>Atlanta</td>
<td>GA</td>
</tr>
<tr>
<td>Augusta</td>
<td>ME</td>
</tr>
<tr>
<td>Austin</td>
<td>TX</td>
</tr>
<tr>
<td>Baker</td>
<td>OR</td>
</tr>
<tr>
<td>Baltimore</td>
<td>MD</td>
</tr>
<tr>
<td>Bangor</td>
<td>ME</td>
</tr>
<tr>
<td>Baton Rouge</td>
<td>LA</td>
</tr>
</tbody>
</table>

Note: When you select specific columns, PROC SQL displays the columns in the order in which you specify them in the SELECT clause.

Eliminating Duplicate Rows from the Query Results

In some cases, you might want to find only the unique values in a column. For example, if you want to find the unique continents in which U.S. states are located, then you might begin by constructing the following query:

```sql
libname sql 'SAS-library';

proc sql outobs=12;
   title 'Continents of the United States';
   select Continent
      from sql.unitedstates;
```

Output 2.4  Selecting a Column with Duplicate Values

You can eliminate the duplicate rows from the results by using the DISTINCT keyword in the SELECT clause. Compare the previous example with the following query, which uses the DISTINCT keyword to produce a single row of output for each continent that is in the Sql.UnitedStates table:

```sql
libname sql 'SAS-library';

proc sql;
  title 'Continents of the United States';
  select distinct Continent
  from sql.unitedstates;
```

Output 2.5  Eliminating Duplicate Values

Note: When you specify all of a table's columns in a SELECT clause with the DISTINCT keyword, PROC SQL eliminates duplicate rows, or rows in which the values in all of the columns match, from the results.
**Determining the Structure of a Table**

To obtain a list of all of the columns in a table and their attributes, you can use the DESCRIBE TABLE statement. The following example generates a description of the Sql.UnitedStates table. PROC SQL writes the description to the log.

```sql
libname sql 'SAS-library';

proc sql;
  describe table sql.unitedstates;
```

**Log 2.1  Portion of Log to Determine the Structure of a Table**

```sql
NOTE: SQL table SQL.UNITEDSTATES was created like:

create table SQL.UNITEDSTATES( bufsize=12288 )
{
  Name  char(35) format=$35. informat=$35. label='Name',
  Capital  char(35) format=$35. informat=$35. label='Capital',
  Population  num format=BEST8. informat=BEST8. label='Population',
  Area  num format=BEST8. informat=BEST8.,
  Continent  char(35) format=$35. informat=$35. label='Continent',
  Statehood  num
};
```

---

**Creating New Columns**

In addition to selecting columns that are stored in a table, you can create new columns that exist for the duration of the query. These columns can contain text or calculations. PROC SQL writes the columns that you create as if they were columns from the table.

**Adding Text to Output**

You can add text to the output by including a string expression, or literal expression, in a query. The following query includes two strings as additional columns in the output:

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'U.S. Postal Codes';
  select 'Postal code for', Name, 'is', Code
  from sql.postalcodes;
```
To prevent the column headings Name and Code from printing, you can assign a label that starts with a special character to each of the columns. PROC SQL does not write the column name when a label is assigned, and it does not write labels that begin with special characters. For example, you could use the following query to suppress the column headings that PROC SQL displayed in the previous example:

```sas
libname sql 'SAS-library';

proc sql outobs=12;
  title 'U.S. Postal Codes';
  select 'Postal code for', Name label='#', 'is', Code label='#'
    from sql.postalcodes;
```

Output 2.6  Adding Text to Output


### Calculating Values

You can perform calculations with values that you retrieve from numeric columns. The following example converts temperatures in the Sql.WorldTemps table from Fahrenheit to Celsius:

```sas
libname sql 'SAS-library';

proc sql outobs=12;
title 'Low Temperatures in Celsius';
select City, (AvgLow - 32) * 5/9 format=4.1
from sql.worldtemps;
```

*Note:* This example uses the FORMAT attribute to modify the format of the calculated output. For more information, see “Specifying Column Attributes” on page 36.
Assigning a Column Alias

By specifying a column alias, you can assign a new name to any column within a PROC SQL query. The new name must follow the rules for SAS names. The name persists only for that query.

When you use an alias to name a column, you can use the alias to reference the column later in the query. PROC SQL uses the alias as the column heading in output. The following example assigns an alias of LowCelsius to the calculated column from the previous example:

```
libname sql 'SAS-library';

proc sql outobs=12;
   title 'Low Temperatures in Celsius';
   select City, (AvgLow - 32) * 5/9 as LowCelsius format=4.1
   from sql.worldtemps;
```
Assigning a Column Alias to a Calculated Column

When you use a column alias to refer to a calculated value, you must use the CALCULATED keyword with the alias to inform PROC SQL that the value is calculated within the query. The following example uses two calculated values, LowC and HighC, to calculate a third value, Range:

```
libname sql 'SAS-library';

proc sql outobs=12;
   title 'Range of High and Low Temperatures in Celsius';
   select City, (AvgHigh - 32) * 5/9 as HighC format=5.1,
       (AvgLow - 32) * 5/9 as LowC format=5.1,
       (calculated HighC - calculated LowC) as Range format=4.1
   from sql.worldtemps;
```

Note: You can use an alias to refer to a calculated column in a SELECT clause, a WHERE clause, or ORDER BY clause.
Output 2.10  Referring to a Calculated Column by Alias

Note: Because this query sets a numeric format of 4.1 on the HighC, LowC, and Range columns, the values in those columns are rounded to the nearest tenth. As a result of the rounding, some of the values in the HighC and LowC columns do not reflect the range value output for the Range column. When you round numeric data values, this type of error sometimes occurs. If you want to avoid this problem, then you can specify additional decimal places in the format.

For more information, see “Using Column Aliases” on page 146.

Assigning Values Conditionally

Using a Simple CASE Expression

CASE expressions enable you to interpret and change some or all of the data values in a column to make the data more useful or meaningful.

You can use conditional logic within a query by using a CASE expression to conditionally assign a value. You can use a CASE expression anywhere that you can use a column name.

The following table, which is used in the next example, describes the world climate zones (rounded to the nearest degree) that exist between Location 1 and Location 2:
### Table 2.1  World Climate Zones

<table>
<thead>
<tr>
<th>Climate zone</th>
<th>Location 1</th>
<th>Latitude at Location 1</th>
<th>Location 2</th>
<th>Latitude at Location 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>North Frigid</td>
<td>North Pole</td>
<td>90</td>
<td>Arctic Circle</td>
<td>67</td>
</tr>
<tr>
<td>North Temperate</td>
<td>Arctic Circle</td>
<td>67</td>
<td>Tropic of Cancer</td>
<td>23</td>
</tr>
<tr>
<td>Torrid</td>
<td>Tropic of Cancer</td>
<td>23</td>
<td>Tropic of Capricorn</td>
<td>-23</td>
</tr>
<tr>
<td>South Temperate</td>
<td>Tropic of Capricorn</td>
<td>-23</td>
<td>Antarctic Circle</td>
<td>-67</td>
</tr>
<tr>
<td>South Frigid</td>
<td>Antarctic Circle</td>
<td>-67</td>
<td>South Pole</td>
<td>-90</td>
</tr>
</tbody>
</table>

In this example, a CASE expression determines the climate zone for each city based on the value in the Latitude column in the Sql.WorldCityCoords table. The query also assigns an alias of ClimateZone to the value. You must close the CASE logic with the END keyword.

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Climate Zones of World Cities';
  select City, Country, Latitude,
    case
      when Latitude gt 67 then 'North Frigid'
      when 67 ge Latitude ge 23 then 'North Temperate'
      when 23 gt Latitude gt -23 then 'Torrid'
      when -23 ge Latitude ge -67 then 'South Temperate'
      else 'South Frigid'
    end as ClimateZone
  from sql.worldcitycoords
  order by City;
```
Using a Simple CASE Expression

You can also construct a CASE expression by using the CASE-OPERAND form, as in the following example. This example selects states and assigns them to a region based on the value of the Continent column:

```sas
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Assigning Regions to Continents';
  select Name, Continent,
      case Continent
        when 'North America' then 'Continental U.S.'
        when 'Oceania' then 'Pacific Islands'
        else 'None'
      end as Region
  from sql.unitedstates;
```

**Note:** When you use the CASE-OPERAND form of the CASE expression, the conditions must all be equality tests. That is, they cannot use comparison operators or other types of operators, as are used in “Using a Simple CASE Expression” on page 32.
Using a CASE Expression in the CASE-OPERAND Form

Replacing Missing Values

The COALESCE function enables you to replace missing values in a column with a new value that you specify. For every row that the query processes, the COALESCE function checks each of its arguments until it finds a nonmissing value, and then returns that value. If all of the arguments are missing values, then the COALESCE function returns a missing value. For example, the following query replaces missing values in the LowPoint column in the Sql.Continents table with the words Not Available:

```sas
libname sql 'SAS-library';
proc sql;
   title 'Continental Low Points';
   select Name, coalesce(LowPoint, 'Not Available') as LowPoint
   from sql.continents;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Continent</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Alaska</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Arizona</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Arkansas</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>California</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Colorado</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Connecticut</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Delaware</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>District of Columbia</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Florida</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Georgia</td>
<td>North America</td>
<td>Continental U.S.</td>
</tr>
<tr>
<td>Hawaii</td>
<td>Oceania</td>
<td>Pacific Islands</td>
</tr>
</tbody>
</table>
Using the COALESCE Function to Replace Missing Values

Output 2.13 Using the COALESCE Function to Replace Missing Values

<table>
<thead>
<tr>
<th>Name</th>
<th>LowPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>Lake Assal</td>
</tr>
<tr>
<td>Antarctica</td>
<td>Not Available</td>
</tr>
<tr>
<td>Asia</td>
<td>Dead Sea</td>
</tr>
<tr>
<td>Australia</td>
<td>Lake Eyre</td>
</tr>
<tr>
<td>Central America and Caribbean</td>
<td>Not Available</td>
</tr>
<tr>
<td>Europe</td>
<td>Caspian Sea</td>
</tr>
<tr>
<td>North America</td>
<td>Death Valley</td>
</tr>
<tr>
<td>Oceania</td>
<td>Not Available</td>
</tr>
<tr>
<td>South America</td>
<td>Valdes Peninsula</td>
</tr>
</tbody>
</table>

The following CASE expression shows another way to perform the same replacement of missing values. However, the COALESCE function requires fewer lines of code to obtain the same results:

```
libname sql 'SAS-library';

proc sql;
  title 'Continental Low Points';
  select Name, case
    when LowPoint is missing then 'Not Available'
    else LowPoint
  end as LowPoint
  from sql.continents;
```

Specifying Column Attributes

You can specify the following column attributes, which determine how SAS data is displayed:

- FORMAT=
- INFORMAT=
- LABEL=
- LENGTH=

If you do not specify these attributes, then PROC SQL uses attributes that are already saved in the table or, if no attributes are saved, then it uses the default attributes.

The following example assigns a label of State to the Name column and a format of COMMA10. to the Area column:

```
libname sql 'SAS-library';

proc sql outobs=12;
```
title 'Areas of U.S. States in Square Miles';
select Name label='State', Area format=comma10.
    from sql.unitedstates;

Note: Using the LABEL= keyword is optional. For example, the following two select
clauses are the same:

select Name label='State', Area format=comma10.

select Name 'State', Area format=comma10.

Output 2.14  Specifying Column Attributes

Areas of U.S. States in Square Miles

<table>
<thead>
<tr>
<th>State</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>52,423</td>
</tr>
<tr>
<td>Alaska</td>
<td>656,400</td>
</tr>
<tr>
<td>Arizona</td>
<td>114,000</td>
</tr>
<tr>
<td>Arkansas</td>
<td>53,200</td>
</tr>
<tr>
<td>California</td>
<td>163,700</td>
</tr>
<tr>
<td>Colorado</td>
<td>104,100</td>
</tr>
<tr>
<td>Connecticut</td>
<td>5,500</td>
</tr>
<tr>
<td>Delaware</td>
<td>2,500</td>
</tr>
<tr>
<td>District of Columbia</td>
<td>100</td>
</tr>
<tr>
<td>Florida</td>
<td>65,800</td>
</tr>
<tr>
<td>Georgia</td>
<td>59,400</td>
</tr>
<tr>
<td>Hawaii</td>
<td>10,900</td>
</tr>
</tbody>
</table>

Sorting Data

Overview of Sorting Data

You can sort query results with an ORDER BY clause by specifying any of the columns in the table, including columns that are not selected or columns that are calculated.

Unless an ORDER BY clause is included in the SELECT statement, then a particular order to the output rows, such as the order in which the rows are encountered in the queried table, cannot be guaranteed, even if an index is present. Without an ORDER BY clause, the order of the output rows is determined by the internal processing of PROC SQL, the default collating sequence of SAS, and your operating environment. Therefore, if you want your result table to appear in a particular order, then use the ORDER BY clause.
For more information and examples, see the “ORDER BY Clause” on page 265.

Calculated statistics can vary slightly, depending on the order in which observations are processed. Such variations are due to numerical error introduced by floating point arithmetic, the results of which should be considered approximate and not exact. Order of observation processing can be affected by non-deterministic effects of multi-threaded or parallel processing. Order of processing can also be affected by inconsistent or non-deterministic ordering of observations produced by a data source, such as a DBMS delivering query results through an ACCESS engine. For more information, see “Numerical Accuracy in SAS Software” in SAS Language Reference: Concepts and “Threading in Base SAS” in SAS Language Reference: Concepts.

### Sorting by Column

The following example selects countries and their populations from the Sql.Countries table and orders the results by population:

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Country Populations';
  select Name, Population format=comma10.
    from sql.countries
    order by Population;
```

*Note:* When you use an ORDER BY clause, you change the order of the output but not the order of the rows that are stored in the table.

*Note:* The PROC SQL default sort order is ascending.

**Output 2.15  Sorting by Column**

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vatican City</td>
<td>1,010</td>
</tr>
<tr>
<td>Nauru</td>
<td>10,099</td>
</tr>
<tr>
<td>Tuvalu</td>
<td>10,099</td>
</tr>
<tr>
<td>Leeward Islands</td>
<td>12,119</td>
</tr>
<tr>
<td>Turks and Caicos Islands</td>
<td>12,119</td>
</tr>
<tr>
<td>Cayman Islands</td>
<td>23,228</td>
</tr>
<tr>
<td>San Marino</td>
<td>24,238</td>
</tr>
<tr>
<td>Liechtenstein</td>
<td>30,297</td>
</tr>
<tr>
<td>Gibraltar</td>
<td>30,297</td>
</tr>
<tr>
<td>Monaco</td>
<td>31,307</td>
</tr>
<tr>
<td>Saint Kitts and Nevis</td>
<td>41,406</td>
</tr>
<tr>
<td>Marshall Islands</td>
<td>54,535</td>
</tr>
</tbody>
</table>
Sorting by Multiple Columns

You can sort by more than one column by specifying the column names, separated by commas, in the ORDER BY clause. The following example sorts the Sql.Countries table by two columns, Continent and Name:

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Countries, Sorted by Continent and Name';
  select Name, Continent
  from sql.countries
  order by Continent, Name;
```

Output 2.16  Sorting by Multiple Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Continent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bermuda</td>
<td></td>
</tr>
<tr>
<td>Iceland</td>
<td></td>
</tr>
<tr>
<td>Kalaallit Nunaat</td>
<td></td>
</tr>
<tr>
<td>Algeria</td>
<td>Africa</td>
</tr>
<tr>
<td>Angola</td>
<td>Africa</td>
</tr>
<tr>
<td>Benin</td>
<td>Africa</td>
</tr>
<tr>
<td>Botswana</td>
<td>Africa</td>
</tr>
<tr>
<td>Burkina Faso</td>
<td>Africa</td>
</tr>
<tr>
<td>Burundi</td>
<td>Africa</td>
</tr>
<tr>
<td>Cameroon</td>
<td>Africa</td>
</tr>
<tr>
<td>Cape Verde</td>
<td>Africa</td>
</tr>
<tr>
<td>Central African Republic</td>
<td>Africa</td>
</tr>
</tbody>
</table>

*Note:* The results list countries without continents first because PROC SQL sorts missing values first in an ascending sort.

Specifying a Sort Order

To order the results, specify ASC for ascending or DESC for descending. You can specify a sort order for each column in the ORDER BY clause.

When you specify multiple columns in the ORDER BY clause, the first column determines the primary row order of the results. Subsequent columns determine the order of rows that have the same value for the primary sort. The following example sorts the Sql.Features table by feature type and name:
libname sql 'SAS-library';

proc sql outobs=12;
    title 'World Topographical Features';
    select Name, Type
    from sql.features
    order by Type desc, Name;

*Note:* The ASC keyword is optional because the PROC SQL default sort order is ascending.

**Output 2.17  Specifying a Sort Order**

**World Topographical Features**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angel Falls</td>
<td>Waterfall</td>
</tr>
<tr>
<td>Niagra Falls</td>
<td>Waterfall</td>
</tr>
<tr>
<td>Tugela Falls</td>
<td>Waterfall</td>
</tr>
<tr>
<td>Yosemite</td>
<td>Waterfall</td>
</tr>
<tr>
<td>Andaman</td>
<td>Sea</td>
</tr>
<tr>
<td>Baltic</td>
<td>Sea</td>
</tr>
<tr>
<td>Bering</td>
<td>Sea</td>
</tr>
<tr>
<td>Black</td>
<td>Sea</td>
</tr>
<tr>
<td>Caribbean</td>
<td>Sea</td>
</tr>
<tr>
<td>Gulf of Mexico</td>
<td>Sea</td>
</tr>
<tr>
<td>Hudson Bay</td>
<td>Sea</td>
</tr>
<tr>
<td>Mediterranean</td>
<td>Sea</td>
</tr>
</tbody>
</table>

**Sorting by Calculated Column**

You can sort by a calculated column by specifying its alias in the ORDER BY clause. The following example calculates population densities and then performs a sort on the calculated Density column:

libname sql 'SAS-library';

proc sql outobs=12;
    title 'World Population Densities per Square Mile';
    select Name, Population format=comma12., Area format=comma8.,
         Population/Area as Density format=comma10.
    from sql.countries
    order by Density desc;
Output 2.18  Sorting by Calculated Column

**World Population Densities per Square Mile**

<table>
<thead>
<tr>
<th>Name</th>
<th>Population</th>
<th>Area</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hong Kong</td>
<td>5,857,414</td>
<td>400</td>
<td>14,644</td>
</tr>
<tr>
<td>Singapore</td>
<td>2,887,301</td>
<td>200</td>
<td>14,437</td>
</tr>
<tr>
<td>Luxembourg</td>
<td>405,980</td>
<td>100</td>
<td>4,060</td>
</tr>
<tr>
<td>Malta</td>
<td>370,633</td>
<td>100</td>
<td>3,706</td>
</tr>
<tr>
<td>Maldives</td>
<td>254,495</td>
<td>100</td>
<td>2,545</td>
</tr>
<tr>
<td>Bangladesh</td>
<td>126,387,860</td>
<td>67,300</td>
<td>2,206</td>
</tr>
<tr>
<td>Bahrain</td>
<td>591,800</td>
<td>300</td>
<td>1,973</td>
</tr>
<tr>
<td>Taiwan</td>
<td>21,509,839</td>
<td>14,000</td>
<td>1,536</td>
</tr>
<tr>
<td>Channel Islands</td>
<td>146,436</td>
<td>100</td>
<td>1,464</td>
</tr>
<tr>
<td>Barbados</td>
<td>258,534</td>
<td>200</td>
<td>1,293</td>
</tr>
<tr>
<td>Korea, South</td>
<td>45,529,277</td>
<td>38,300</td>
<td>1,189</td>
</tr>
<tr>
<td>Mauritius</td>
<td>1,128,057</td>
<td>1,000</td>
<td>1,128</td>
</tr>
</tbody>
</table>

**Sorting by Column Position**

You can sort by any column within the SELECT clause by specifying its numerical position. By specifying a position instead of a name, you can sort by a calculated column that has no alias. The following example does not assign an alias to the calculated Density column. Instead, the column position of 4 in the ORDER BY clause refers to the position of the calculated column in the SELECT clause:

```plaintext
libname sql 'SAS-library';

proc sql outobs=12;
  title 'World Population Densities per Square Mile';
  select Name, Population format=comma12., Area format=comma8.,
      Population/Area format=comma10. label='Density'
  from sql.countries
  order by 4 desc;
```

**Note:** PROC SQL uses a label, if one has been assigned, as a heading for a column that does not have an alias.
You can sort query results by columns that are not included in the query. For example, the following query returns all the rows in the Sql.Countries table and sorts them by population, even though the Population column is not included in the query:

```sql
libname sql 'SAS-library';
proc sql outobs=12;
  title 'Countries, Sorted by Population';
  select Name, Continent
  from sql.countries
  order by Population;
```

### Sorting by Columns That Are Not Selected

You can sort query results by columns that are not included in the query. For example, the following query returns all the rows in the Sql.Countries table and sorts them by population, even though the Population column is not included in the query:

```sql
libname sql 'SAS-library';
proc sql outobs=12;
  title 'Countries, Sorted by Population';
  select Name, Continent
  from sql.countries
  order by Population;
```
Specifying a Different Sorting Sequence

SORTSEQ= is a PROC SQL statement option that specifies the sorting sequence for PROC SQL to use when a query contains an ORDER BY clause. Use this option only if you want to use a sorting sequence other than your operating environment's default sorting sequence. Possible values include ASCII, EBCDIC, and some languages other than English. For example, in an operating environment that supports the EBCDIC sorting sequence, you could use the following option in the PROC SQL statement to set the sorting sequence to EBCDIC:

```sql
proc sql sortseq=ebcdic;
```

Beginning with SAS 9.4M3, linguistic collation is supported with the SORTSEQ statement option. For more information, see “SORTSEQ=sort-table | LINGUISTIC” on page 229.

Note: SORTSEQ= affects only the ORDER BY clause. It does not override your operating environment's default comparison operations for the WHERE clause.

Operating Environment Information

For more information about the default and other sorting sequences for your operating environment, see the SAS documentation for your operating environment.

Sorting Columns That Contain Missing Values

PROC SQL sorts nulls, or missing values, before character or numeric data. Therefore, when you specify ascending order, missing values appear first in the query results.
The following example sorts the rows in the Continents table by the LowPoint column:

```
libname sql 'SAS-library';

proc sql;
  title 'Continents, Sorted by Low Point';
  select Name, LowPoint
  from sql.continents
  order by LowPoint;
```

Because three continents have a missing value in the LowPoint column, those continents appear first in the output. Note that because the query does not specify a secondary sort, rows that have the same value in the LowPoint column, such as the first three rows of output, are not displayed in any particular order. In general, if you do not explicitly specify a sort order, then PROC SQL output is not guaranteed to be in any particular order.

**Output 2.21  Sorting Columns That Contain Missing Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>LowPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central America and Caribbean</td>
<td></td>
</tr>
<tr>
<td>Antarctica</td>
<td></td>
</tr>
<tr>
<td>Oceania</td>
<td></td>
</tr>
<tr>
<td>Europe</td>
<td>Caspian Sea</td>
</tr>
<tr>
<td>Asia</td>
<td>Dead Sea</td>
</tr>
<tr>
<td>North America</td>
<td>Death Valley</td>
</tr>
<tr>
<td>Africa</td>
<td>Lake Assal</td>
</tr>
<tr>
<td>Australia</td>
<td>Lake Eyre</td>
</tr>
<tr>
<td>South America</td>
<td>Valdes Peninsula</td>
</tr>
</tbody>
</table>

---

**Retrieving Rows That Satisfy a Condition**

The WHERE clause enables you to retrieve only rows from a table that satisfy a condition. WHERE clauses can contain any of the columns in a table, including columns that are not selected.

**Using a Simple WHERE Clause**

The following example uses a WHERE clause to find all countries that are in the continent of Europe and their populations:

```
libname sql 'SAS-library';

proc sql outobs=12;
```
title 'Countries in Europe';
select Name, Population format=comma10.
from sql.countries
where Continent = 'Europe';

Output 2.22  Using a Simple WHERE Clause

<table>
<thead>
<tr>
<th>Countries in Europe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Albania</td>
</tr>
<tr>
<td>Andorra</td>
</tr>
<tr>
<td>Austria</td>
</tr>
<tr>
<td>Belarus</td>
</tr>
<tr>
<td>Belgium</td>
</tr>
<tr>
<td>Bosnia and Herzegovina</td>
</tr>
<tr>
<td>Bulgaria</td>
</tr>
<tr>
<td>Channel Islands</td>
</tr>
<tr>
<td>Croatia</td>
</tr>
<tr>
<td>Czech Republic</td>
</tr>
<tr>
<td>Denmark</td>
</tr>
<tr>
<td>England</td>
</tr>
</tbody>
</table>

Retrieving Rows Based on a Comparison

You can use comparison operators in a WHERE clause to select different subsets of data. The following table lists the comparison operators that you can use:

<table>
<thead>
<tr>
<th>Table 2.2  Comparison Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>=</td>
</tr>
<tr>
<td>~= or ^= or &lt;&gt;</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>Symbol</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>&gt;=</td>
</tr>
<tr>
<td>&lt;=</td>
</tr>
</tbody>
</table>

The following example subsets the Sql.UnitedStates table by including only states with populations greater than 5,000,000 people:

```sas
libname sql 'SAS-library';

proc sql;
  title 'States with Populations over 5,000,000';
  select Name, Population format=comma10.
    from sql.unitedstates
  where Population gt 5000000
  order by Population desc;
```
Retrieving Rows That Satisfy a Condition

**Output 2.23** Retrieving Rows Based on a Comparison

![Table 2.3](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>31,513,948</td>
</tr>
<tr>
<td>New York</td>
<td>18,377,334</td>
</tr>
<tr>
<td>Texas</td>
<td>18,209,994</td>
</tr>
<tr>
<td>Florida</td>
<td>13,814,408</td>
</tr>
<tr>
<td>Pennsylvania</td>
<td>12,167,566</td>
</tr>
<tr>
<td>Illinois</td>
<td>11,813,091</td>
</tr>
<tr>
<td>Ohio</td>
<td>11,200,790</td>
</tr>
<tr>
<td>Michigan</td>
<td>9,571,318</td>
</tr>
<tr>
<td>New Jersey</td>
<td>7,957,196</td>
</tr>
<tr>
<td>North Carolina</td>
<td>7,013,950</td>
</tr>
<tr>
<td>Georgia</td>
<td>6,965,572</td>
</tr>
<tr>
<td>Virginia</td>
<td>6,554,851</td>
</tr>
<tr>
<td>Massachusetts</td>
<td>6,071,816</td>
</tr>
<tr>
<td>Indiana</td>
<td>5,769,553</td>
</tr>
<tr>
<td>Washington</td>
<td>5,307,322</td>
</tr>
<tr>
<td>Missouri</td>
<td>5,285,610</td>
</tr>
<tr>
<td>Tennessee</td>
<td>5,149,273</td>
</tr>
<tr>
<td>Wisconsin</td>
<td>5,087,770</td>
</tr>
<tr>
<td>Maryland</td>
<td>5,014,048</td>
</tr>
</tbody>
</table>

**Retrieving Rows That Satisfy Multiple Conditions**

You can use logical, or Boolean, operators to construct a WHERE clause that contains two or more expressions. The following table lists the logical operators that you can use:

**Table 2.3 Logical (Boolean) Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Mnemonic Equivalent</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>specifies that both the previous and following conditions must be true</td>
<td>Continent = 'Asia' and Population &gt; 5000000</td>
</tr>
</tbody>
</table>
The following example uses two expressions to include only countries that are in Africa and that have a population greater than 20,000,000 people:

```sas
libname sql 'SAS-library';
proc sql;
title 'Countries in Africa with Populations over 20,000,000';
select Name, Population format=comma10.
from sql.countries
where Continent = 'Africa' and Population gt 20000000
order by Population desc;
```

**Output 2.24 Retrieving Rows That Satisfy Multiple Conditions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nigeria</td>
<td>99,062,003</td>
</tr>
<tr>
<td>Egypt</td>
<td>59,912,259</td>
</tr>
<tr>
<td>Ethiopia</td>
<td>59,291,170</td>
</tr>
<tr>
<td>South Africa</td>
<td>44,365,873</td>
</tr>
<tr>
<td>Congo, Democratic Republic of</td>
<td>43,106,529</td>
</tr>
<tr>
<td>Sudan</td>
<td>29,711,229</td>
</tr>
<tr>
<td>Morocco</td>
<td>28,841,705</td>
</tr>
<tr>
<td>Kenya</td>
<td>28,520,558</td>
</tr>
<tr>
<td>Tanzania</td>
<td>28,263,033</td>
</tr>
<tr>
<td>Algeria</td>
<td>28,171,132</td>
</tr>
<tr>
<td>Uganda</td>
<td>20,055,584</td>
</tr>
</tbody>
</table>

**Note:** You can use parentheses to improve the readability of WHERE clauses that contain multiple, or compound, expressions, such as the following:

```sas
where (Continent = 'Africa' and Population gt 20000000) or
(Continent = 'Asia' and Population gt 1000000)
```
Using Other Conditional Operators

Overview of Using Other Conditional Operators
You can use many different conditional operators in a WHERE clause. The following table lists other operators that you can use:

Table 2.4  Conditional Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>specifies that at least one of a set of values obtained from a subquery must</td>
<td>where Population &gt; any (select Population from sql.countries)</td>
</tr>
<tr>
<td></td>
<td>satisfy a given condition</td>
<td></td>
</tr>
<tr>
<td>ALL</td>
<td>specifies that all of the values obtained from a subquery must satisfy</td>
<td>where Population &gt; all (select Population from sql.countries)</td>
</tr>
<tr>
<td></td>
<td>satisfy a given condition</td>
<td></td>
</tr>
<tr>
<td>BETWEEN-AND</td>
<td>tests for values within an inclusive range</td>
<td>where Population between 1000000 and 5000000</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>tests for values that contain a specified string</td>
<td>where Continent contains 'America';</td>
</tr>
<tr>
<td>EXISTS</td>
<td>tests for the existence of a set of values obtained from a subquery</td>
<td>where exists (select * from sql.oilprod);</td>
</tr>
<tr>
<td>IN</td>
<td>tests for values that match one of a list of values</td>
<td>where Name in ('Africa', 'Asia');</td>
</tr>
<tr>
<td>IS NULL or IS</td>
<td>tests for missing values</td>
<td>where Population is missing;</td>
</tr>
<tr>
<td>MISSING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIKE</td>
<td>tests for values that match a specified pattern¹</td>
<td>where Continent like 'A %';</td>
</tr>
<tr>
<td>=*</td>
<td>tests for values that sound like a specified value</td>
<td>where Name =* 'Tiland';</td>
</tr>
</tbody>
</table>

Note: All of these operators can be prefixed with the NOT operator to form a negative condition.

¹ You can use a percent sign (%) to match any number of characters. You can use an underscore (_) to match one arbitrary character.
Using the IN Operator

The IN operator enables you to include values within a list that you supply. The following example uses the IN operator to include only the mountains and waterfalls in the Sql.Features table:

```sas
libname sql 'SAS-library';

proc sql outobs=12;
title 'World Mountains and Waterfalls';
select Name, Type, Height format=comma10.
from sql.features
where Type in ('Mountain', 'Waterfall')
order by Height;
```

Output 2.25  Using the IN Operator

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara Falls</td>
<td>Waterfall</td>
<td>193</td>
</tr>
<tr>
<td>Yosemite</td>
<td>Waterfall</td>
<td>2,425</td>
</tr>
<tr>
<td>Tugela Falls</td>
<td>Waterfall</td>
<td>3,110</td>
</tr>
<tr>
<td>Angel Falls</td>
<td>Waterfall</td>
<td>3,212</td>
</tr>
<tr>
<td>Kosciusko</td>
<td>Mountain</td>
<td>7,310</td>
</tr>
<tr>
<td>Pico Duarte</td>
<td>Mountain</td>
<td>10,417</td>
</tr>
<tr>
<td>Cook</td>
<td>Mountain</td>
<td>12,349</td>
</tr>
<tr>
<td>Matterhorn</td>
<td>Mountain</td>
<td>14,690</td>
</tr>
<tr>
<td>Wilhelm</td>
<td>Mountain</td>
<td>14,793</td>
</tr>
<tr>
<td>Mont Blanc</td>
<td>Mountain</td>
<td>15,771</td>
</tr>
<tr>
<td>Ararat</td>
<td>Mountain</td>
<td>16,804</td>
</tr>
<tr>
<td>Vinson Massif</td>
<td>Mountain</td>
<td>16,864</td>
</tr>
</tbody>
</table>

Using the IS MISSING Operator

The IS MISSING operator enables you to identify rows that contain columns with missing values. The following example selects countries that are not located on a continent. That is, these countries have a missing value in the Continent column:

```sas
proc sql;
title 'Countries with Missing Continents';
select Name, Continent
from sql.countries
where Continent is missing;
```

Note: The IS NULL operator is the same as, and interchangeable with, the IS MISSING operator.
Using the BETWEEN-AND Operators

To select rows based on a range of values, you can use the BETWEEN-AND operators. This example selects countries that have latitudes within five degrees of the Equator:

```sql
proc sql outobs=12;
  title 'Equatorial Cities of the World';
  select City, Country, Latitude
  from sql.worldcitycoords
  where Latitude between -5 and 5;
```

Note: In the tables used in these examples, latitude values that are south of the Equator are negative. Longitude values that are west of the Prime Meridian are also negative.

Note: Because the BETWEEN-AND operators are inclusive, the values that you specify in the BETWEEN-AND expression are included in the results.
Using the LIKE Operator

The LIKE operator enables you to select rows based on pattern matching. For example, the following query returns all countries in the Sql.Countries table that begin with the letter Z and are any number of characters long, or end with the letter a and are five characters long:

```
libname sql 'SAS-library';

proc sql;
  title1 'Country Names that Begin with the Letter "Z"';
  title2 'or Are 5 Characters Long and End with the Letter "a"';
  select Name
    from sql.countries
    where Name like 'Z%' or Name like '____a';
```

### Output 2.27 Using the BETWEEN-AND Operators

![Equatorial Cities of the World](image)

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belem</td>
<td>Brazil</td>
<td>-0.1</td>
</tr>
<tr>
<td>Fortaleza</td>
<td>Brazil</td>
<td>-0.4</td>
</tr>
<tr>
<td>Bogota</td>
<td>Colombia</td>
<td>0.4</td>
</tr>
<tr>
<td>Cali</td>
<td>Colombia</td>
<td>0.3</td>
</tr>
<tr>
<td>Brazzaville</td>
<td>Congo</td>
<td>-0.4</td>
</tr>
<tr>
<td>Quito</td>
<td>Ecuador</td>
<td>0</td>
</tr>
<tr>
<td>Cayenne</td>
<td>French Guiana</td>
<td>0.5</td>
</tr>
<tr>
<td>Accra</td>
<td>Ghana</td>
<td>0.5</td>
</tr>
<tr>
<td>Medan</td>
<td>Indonesia</td>
<td>0.3</td>
</tr>
<tr>
<td>Palembang</td>
<td>Indonesia</td>
<td>-0.3</td>
</tr>
<tr>
<td>Nairobi</td>
<td>Kenya</td>
<td>-0.1</td>
</tr>
<tr>
<td>Kuala Lumpur</td>
<td>Malaysia</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Using the LIKE Operator

The percent sign (%) and underscore (_) are wildcard characters. For more information about pattern matching with the LIKE comparison operator, see Chapter 7, “SQL Procedure,” on page 215.

Using Truncated String Comparison Operators

Truncated string comparison operators are used to compare two strings. They differ from conventional comparison operators in that, before executing the comparison, PROC SQL truncates the longer string to be the same length as the shorter string. The operand lengths are calculated after trailing blanks, if any, have been eliminated. The truncation is performed internally; neither operand is permanently changed. The following table lists the truncated comparison operators:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQT</td>
<td>equal to truncated strings</td>
<td><code>where Name eqt 'Aust';</code></td>
</tr>
<tr>
<td>GTT</td>
<td>greater than truncated strings</td>
<td><code>where Name gt 'Bah';</code></td>
</tr>
<tr>
<td>LTT</td>
<td>less than truncated strings</td>
<td><code>where Name lt 'An';</code></td>
</tr>
<tr>
<td>GET</td>
<td>greater than or equal to truncated strings</td>
<td><code>where Country get 'United A';</code></td>
</tr>
<tr>
<td>LET</td>
<td>less than or equal to truncated strings</td>
<td><code>where Lastname let 'Smith';</code></td>
</tr>
</tbody>
</table>
The following example returns a list of U.S. states that have 'New ' at the beginning of their names:

```sql
proc sql;
    title "New" U.S. States';
    select Name
        from sql.unitedstates
        where Name eqt 'New ';
```

**Output 2.29 Using a Truncated String Comparison Operator**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Hampshire</td>
</tr>
<tr>
<td>New Jersey</td>
</tr>
<tr>
<td>New Mexico</td>
</tr>
<tr>
<td>New York</td>
</tr>
</tbody>
</table>

**Using a WHERE Clause with Missing Values**

If a column that you specify in a WHERE clause contains missing values, then a query might provide unexpected results. For example, the following query returns all features from the Sql.Features table that have a depth of less than 500 feet:

```sql
libname sql 'SAS-library';
/* incorrect output */

proc sql outobs=12;
    title 'World Features with a Depth of Less than 500 Feet';
    select Name, Depth
        from sql.features
        where Depth lt 500
        order by Depth;
```
However, because PROC SQL treats missing values as smaller than nonmissing values, features for which depth is not listed are also included in the results. To avoid this problem, you could adjust the WHERE expression to check for missing values and exclude them from the query results, as follows:

```sql
libname sql 'SAS-library';

/* corrected output */

proc sql outobs=12;
  title 'World Features with a Depth of Less than 500 Feet';
  select Name, Depth
  from sql.features
  where Depth < 500 and Depth is not missing
  order by Depth;
```

### World Features with a Depth of Less than 500 Feet

<table>
<thead>
<tr>
<th>Name</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kalahari</td>
<td>.</td>
</tr>
<tr>
<td>Nile</td>
<td>.</td>
</tr>
<tr>
<td>Citlaltepec</td>
<td>.</td>
</tr>
<tr>
<td>Lena</td>
<td>.</td>
</tr>
<tr>
<td>Mont Blanc</td>
<td>.</td>
</tr>
<tr>
<td>Borneo</td>
<td>.</td>
</tr>
<tr>
<td>Rub al Khali</td>
<td>.</td>
</tr>
<tr>
<td>Amur</td>
<td>.</td>
</tr>
<tr>
<td>Yosemite</td>
<td>.</td>
</tr>
<tr>
<td>Cook</td>
<td>.</td>
</tr>
<tr>
<td>Mackenzie-Peace</td>
<td>.</td>
</tr>
<tr>
<td>Mekong</td>
<td>.</td>
</tr>
</tbody>
</table>
Summarizing Data

Overview of Summarizing Data

You can use an aggregate function (or summary function) to produce a statistical summary of data in a table. The aggregate function instructs PROC SQL in how to combine data in one or more columns. If you specify one column as the argument to an aggregate function, then the values in that column are calculated. If you specify multiple arguments, then the arguments or columns that are listed are calculated.

Note: When more than one argument is used within an SQL aggregate function, the function is no longer considered to be an SQL aggregate or summary function. If there is a like-named Base SAS function, then PROC SQL executes the Base SAS function and the results that are returned are based on the values for the current row. If no like-named Base SAS function exists, then an error will occur. For example, if you use multiple arguments for the AVG function, an error will occur because there is no AVG function for Base SAS.

When you use an aggregate function, PROC SQL applies the function to the entire table, unless you use a GROUP BY clause. You can use aggregate functions in the SELECT or HAVING clauses.

Note: See “Grouping Data” on page 64 for information about producing summaries of individual groups of data within a table.

Using Aggregate Functions

The following table lists the aggregate functions that you can use:

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG, MEAN</td>
<td>mean or average of values</td>
</tr>
<tr>
<td>COUNT, FREQ, N</td>
<td>number of nonmissing values</td>
</tr>
</tbody>
</table>

World Features with a Depth of Less than 500 Feet

<table>
<thead>
<tr>
<th>Name</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baltic</td>
<td>180</td>
</tr>
<tr>
<td>Aral Sea</td>
<td>222</td>
</tr>
<tr>
<td>Victoria</td>
<td>264</td>
</tr>
<tr>
<td>Hudson Bay</td>
<td>305</td>
</tr>
<tr>
<td>North</td>
<td>308</td>
</tr>
</tbody>
</table>
### Function Definitions

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS</td>
<td>corrected sum of squares</td>
</tr>
<tr>
<td>CV</td>
<td>coefficient of variation (percent)</td>
</tr>
<tr>
<td>MAX</td>
<td>largest value</td>
</tr>
<tr>
<td>MIN</td>
<td>smallest value</td>
</tr>
<tr>
<td>NMISS</td>
<td>number of missing values</td>
</tr>
<tr>
<td>PRT</td>
<td>probability of a greater absolute value of Student's ( t )</td>
</tr>
<tr>
<td>RANGE</td>
<td>range of values</td>
</tr>
<tr>
<td>STD</td>
<td>standard deviation</td>
</tr>
<tr>
<td>STDERR</td>
<td>standard error of the mean</td>
</tr>
<tr>
<td>SUM</td>
<td>sum of values</td>
</tr>
<tr>
<td>SUMWGT</td>
<td>sum of the WEIGHT variable values(^1)</td>
</tr>
<tr>
<td>T</td>
<td>Student's ( t ) value for testing the hypothesis that the population mean is zero</td>
</tr>
<tr>
<td>USS</td>
<td>uncorrected sum of squares</td>
</tr>
<tr>
<td>VAR</td>
<td>variance</td>
</tr>
</tbody>
</table>

**Note:** You can use most other SAS functions in PROC SQL, but they are not treated as aggregate functions.

---

**Summarizing Data with a WHERE Clause**

**Overview of Summarizing Data with a WHERE Clause**

You can use aggregate, or summary functions, by using a WHERE clause. For a complete list of the aggregate functions that you can use, see Table 2.6 on page 56.

**Using the MEAN Function with a WHERE Clause**

This example uses the MEAN function to find the annual mean temperature for each country in the Sql.WorldTemps table. The WHERE clause returns countries with a mean temperature that is greater than 75 degrees.

```sql
libname sql 'SAS-library';

proc sql outobs=12;
```

---

\(^1\) In the SQL procedure, each row has a weight of 1.
title 'Mean Temperatures for World Cities';
select City, Country, mean(AvgHigh, AvgLow)
as MeanTemp
from sql.worldtemps
where calculated MeanTemp gt 75
order by MeanTemp desc;

Note: You must use the CALCULATED keyword to reference the calculated column.

Output 2.32  Using the MEAN Function with a WHERE Clause

Mean Temperatures for World Cities

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>MeanTemp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lagos</td>
<td>Nigeria</td>
<td>82.5</td>
</tr>
<tr>
<td>Manila</td>
<td>Philippines</td>
<td>82</td>
</tr>
<tr>
<td>Bangkok</td>
<td>Thailand</td>
<td>82</td>
</tr>
<tr>
<td>Singapore</td>
<td>Singapore</td>
<td>81</td>
</tr>
<tr>
<td>Bombay</td>
<td>India</td>
<td>79</td>
</tr>
<tr>
<td>Kingston</td>
<td>Jamaica</td>
<td>78</td>
</tr>
<tr>
<td>San Juan</td>
<td>Puerto Rico</td>
<td>78</td>
</tr>
<tr>
<td>Calcutta</td>
<td>India</td>
<td>76.5</td>
</tr>
<tr>
<td>Havana</td>
<td>Cuba</td>
<td>76.5</td>
</tr>
<tr>
<td>Nassau</td>
<td>Bahamas</td>
<td>76.5</td>
</tr>
</tbody>
</table>

Displaying Sums

The following example uses the SUM function to return the total oil reserves for all countries in the Sql.OilRsrvs table:

libname sql 'SAS-library';
proc sql;
    title 'World Oil Reserves';
    select sum(Barrels) format=comma18. as TotalBarrels
    from sql.oilrsrvs;
    
Note: The SUM function produces a single row of output for the requested sum because no nonaggregate value appears in the SELECT clause.
Output 2.33 Displaying Sums

World Oil Reserves

<table>
<thead>
<tr>
<th>TotalBarrels</th>
</tr>
</thead>
<tbody>
<tr>
<td>878,300,000,000</td>
</tr>
</tbody>
</table>

Combining Data from Multiple Rows into a Single Row

In the previous example, PROC SQL combined information from multiple rows of data into a single row of output. Specifically, the world oil reserves for each country were combined to form a total for all countries. Combining, or rolling up, of rows occurs when the following conditions exist:

- The SELECT clause contains only columns that are specified within an aggregate function.
- The WHERE clause, if there is one, contains only columns that are specified in the SELECT clause.

Remerging Summary Statistics

The following example uses the MAX function to find the largest population in the Sql.Countries table and displays it in a column called MaxPopulation. Aggregate functions, such as the MAX function, can cause the same calculation to repeat for every row. This occurs whenever PROC SQL remerges data. Remerging occurs whenever any of the following conditions exist:

- The SELECT clause references a column that contains an aggregate function and other columns that are not listed in the GROUP BY clause.
- The ORDER BY clause references a column that is not referenced by the SELECT clause.

Note: When a query remerges data, PROC SQL displays a note in the log to indicate that data remerging has occurred.

In this example, PROC SQL writes the population of China, which is the largest population in the table:

```bash
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Largest Country Populations';
  select Name, Population format=comma20.,
        max(Population) as MaxPopulation format=comma20.
  from sql.countries
  order by Population desc;
```
In some cases, you might need to use an aggregate function so that you can use its results in another calculation. To do this, you need only to construct one query for PROC SQL to automatically perform both calculations. This type of operation also causes PROC SQL to remerge the data.

For example, if you want to find the percentage of the total world population that resides in each country, then you construct a single query that performs the following tasks:

- obtains the total world population by using the SUM function
- divides each country's population by the total world population

PROC SQL runs an internal query to find the sum and then runs another internal query to divide each country's population by the sum.

```sql
libname sql 'SAS-library';

proc sql outobs=12;
  title 'Percentage of World Population in Countries';
  select Name, Population format=comma14.,
    (Population / sum(Population) * 100) as Percentage format=comma8.2
  from sql.countries
  order by Percentage desc;
```

### Output 2.34 Remerging Summary Statistics

<table>
<thead>
<tr>
<th>Name</th>
<th>Population</th>
<th>MaxPopulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1,202,215,077</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>India</td>
<td>929,009,120</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>United States</td>
<td>263,294,808</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Indonesia</td>
<td>202,393,859</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Brazil</td>
<td>160,310,357</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Russia</td>
<td>151,089,979</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Bangladesh</td>
<td>126,387,850</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Japan</td>
<td>126,345,434</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Pakistan</td>
<td>123,062,252</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Nigeria</td>
<td>99,062,003</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Mexico</td>
<td>93,114,708</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Germany</td>
<td>81,890,690</td>
<td>1,202,215,077</td>
</tr>
</tbody>
</table>
Using Aggregate Functions with Unique Values

### Counting Unique Values
You can use DISTINCT with an aggregate function to cause the function to use only unique values from a column.

The following query returns the number of distinct, nonmissing continents in the Sql.Countries table:

```sql
libname sql 'SAS-library';

proc sql;
  title 'Number of Continents in the Countries Table';
  select count(distinct Continent) as Count
  from sql.countries;
```

**Output 2.36** Using DISTINCT with the COUNT Function

#### Number of Continents in the Countries Table

<table>
<thead>
<tr>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
Note: You cannot use `select count(distinct *)` to count distinct rows in a table. This code generates an error because PROC SQL does not know which duplicate column values to eliminate.

**Counting Nonmissing Values**

Compare the previous example with the following query, which does not use the DISTINCT keyword. This query counts every nonmissing occurrence of a continent in the Sql.Countries table, including duplicate values:

```sql
libname sql 'SAS-library';
proc sql;
  title 'Countries for Which a Continent is Listed';
  select count(Continent) as Count
  from sql.countries;
```

**Output 2.37  Effect of Not Using DISTINCT with the COUNT Function**

<table>
<thead>
<tr>
<th>Countries for Which a Continent is Listed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
</tr>
<tr>
<td>205</td>
</tr>
</tbody>
</table>

**Counting All Rows**

In the previous two examples, countries that have a missing value in the Continent column are ignored by the COUNT function. To obtain a count of all rows in the table, including countries that are not on a continent, you can use the following code in the SELECT clause:

```sql
proc sql;
  title 'Number of Countries in the Sql.Countries Table';
  select count(*) as Number
  from sql.countries;
```

**Output 2.38  Using the COUNT Function to Count All Rows in a Table**

<table>
<thead>
<tr>
<th>Number of Countries in the Sql.Countries Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
</tr>
<tr>
<td>208</td>
</tr>
</tbody>
</table>

**Summarizing Data with Missing Values**

**Overview of Summarizing Data with Missing Values**

When you use an aggregate function with data that contains missing values, the results might not provide the information that you expect because many aggregate functions ignore missing values.
Finding Errors Caused by Missing Values

The AVG function returns the average of only the nonmissing values. The following query calculates the average length of three features in the Sql.Features table: Angel Falls and the Amazon and Nile rivers:

```sql
libname sql 'SAS-library';

/* unexpected output */

proc sql;
  title 'Average Length of Angel Falls, Amazon and Nile Rivers';
  select Name, Length, avg(Length) as AvgLength
  from sql.features
  where Name in ('Angel Falls', 'Amazon', 'Nile');
```

**Output 2.39  Finding Errors Caused by Missing Values (Unexpected Output)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>AvgLength</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>4000</td>
<td>4072.5</td>
</tr>
<tr>
<td>Angel Falls</td>
<td>.</td>
<td>4072.5</td>
</tr>
<tr>
<td>Nile</td>
<td>4145</td>
<td>4072.5</td>
</tr>
</tbody>
</table>

Because no length is stored for Angel Falls, the average includes only the values for the Amazon and Nile rivers. Therefore, the average contains unexpected output results.

Compare the results from the previous example with the following query, which includes a COALESCE expression to handle missing values:

```sql
/* modified output */

proc sql;
  title 'Average Length of Angel Falls, Amazon and Nile Rivers';
  select Name, Length, coalesce(Length, 0) as NewLength,
        avg(calculated NewLength) as AvgLength
  from sql.features
  where Name in ('Angel Falls', 'Amazon', 'Nile');
```

**Output 2.40  Finding Errors Caused by Missing Values (Modified Output)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>NewLength</th>
<th>AvgLength</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>4000</td>
<td>4000</td>
<td>2715</td>
</tr>
<tr>
<td>Angel Falls</td>
<td>.</td>
<td>0</td>
<td>2715</td>
</tr>
<tr>
<td>Nile</td>
<td>4145</td>
<td>4145</td>
<td>2715</td>
</tr>
</tbody>
</table>
Grouping Data

The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, you also use an aggregate function in the SELECT clause or in a HAVING clause to instruct PROC SQL in how to summarize the data for each group. PROC SQL calculates the aggregate function separately for each group.

Grouping by One Column

The following example sums the populations of all countries to find the total population of each continent:

```sql
libname sql 'SAS-library';

proc sql;
  title 'Total Populations of World Continents';
  select Continent, sum(Population) format=comma14 as TotalPopulation
  from sql.countries
  where Continent is not missing
  group by Continent;
```

*Note:* Countries for which a continent is not listed are excluded by the WHERE clause.

Output 2.41  Grouping by One Column

<table>
<thead>
<tr>
<th>Continent</th>
<th>TotalPopulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>710,529,592</td>
</tr>
<tr>
<td>Asia</td>
<td>3,381,858,879</td>
</tr>
<tr>
<td>Australia</td>
<td>18,255,944</td>
</tr>
<tr>
<td>Central America and Caribbean</td>
<td>66,815,930</td>
</tr>
<tr>
<td>Europe</td>
<td>813,481,724</td>
</tr>
<tr>
<td>North America</td>
<td>384,801,818</td>
</tr>
<tr>
<td>Oceania</td>
<td>5,342,368</td>
</tr>
<tr>
<td>South America</td>
<td>317,568,801</td>
</tr>
</tbody>
</table>

Grouping without Summarizing

When you use a GROUP BY clause without an aggregate function, PROC SQL treats the GROUP BY clause as if it were an ORDER BY clause and displays a message in the log that informs you that this has happened. The following example attempts to group high and low temperature information for each city in the Sql.WorldTemps table by country:

```sql
libname sql 'SAS-library';
```
proc sql outobs=12;
  title 'High and Low Temperatures';
  select City, Country, AvgHigh, AvgLow
  from sql.worldtemps
  group by Country;

The output and log show that PROC SQL transforms the GROUP BY clause into an ORDER BY clause.

Output 2.42  Grouping without Aggregate Functions

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>AvgHigh</th>
<th>AvgLow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>90</td>
<td>45</td>
</tr>
<tr>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>87</td>
<td>48</td>
</tr>
<tr>
<td>Sydney</td>
<td>Australia</td>
<td>79</td>
<td>44</td>
</tr>
<tr>
<td>Vienna</td>
<td>Austria</td>
<td>76</td>
<td>28</td>
</tr>
<tr>
<td>Nassau</td>
<td>Bahamas</td>
<td>88</td>
<td>65</td>
</tr>
<tr>
<td>Hamilton</td>
<td>Bermuda</td>
<td>85</td>
<td>69</td>
</tr>
<tr>
<td>Sao Paulo</td>
<td>Brazil</td>
<td>81</td>
<td>53</td>
</tr>
<tr>
<td>Rio de Janeiro</td>
<td>Brazil</td>
<td>85</td>
<td>64</td>
</tr>
<tr>
<td>Quebec</td>
<td>Canada</td>
<td>76</td>
<td>5</td>
</tr>
<tr>
<td>Montreal</td>
<td>Canada</td>
<td>77</td>
<td>8</td>
</tr>
<tr>
<td>Toronto</td>
<td>Canada</td>
<td>80</td>
<td>17</td>
</tr>
<tr>
<td>Beijing</td>
<td>China</td>
<td>86</td>
<td>17</td>
</tr>
</tbody>
</table>

Log 2.2  Grouping without Aggregate Functions (Partial Log)

WARNING: A GROUP BY clause has been transformed into an ORDER BY clause because neither the SELECT clause nor the optional HAVING clause of the associated table-expression referenced a summary function.

Grouping by Multiple Columns

To group by multiple columns, separate the column names with commas within the GROUP BY clause. You can use aggregate functions with any of the columns that you select. The following example groups by both Location and Type, producing total square miles for the deserts and lakes in each location in the Sql.Features table:

libname sql 'SAS-library';

proc sql;
title 'Total Square Miles of Deserts and Lakes';
select Location, Type, sum(Area) as TotalArea format=comma16.
from sql.features
where type in ('Desert', 'Lake')
group by Location, Type;

Output 2.43  Grouping by Multiple Columns

<table>
<thead>
<tr>
<th>Location</th>
<th>Type</th>
<th>TotalArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>Desert</td>
<td>3,725,000</td>
</tr>
<tr>
<td>Africa</td>
<td>Lake</td>
<td>50,968</td>
</tr>
<tr>
<td>Asia</td>
<td>Lake</td>
<td>25,300</td>
</tr>
<tr>
<td>Australia</td>
<td>Desert</td>
<td>300,000</td>
</tr>
<tr>
<td>Canada</td>
<td>Lake</td>
<td>12,275</td>
</tr>
<tr>
<td>China</td>
<td>Desert</td>
<td>500,000</td>
</tr>
<tr>
<td>Europe - Asia</td>
<td>Lake</td>
<td>143,560</td>
</tr>
<tr>
<td>North America</td>
<td>Desert</td>
<td>140,000</td>
</tr>
<tr>
<td>North America</td>
<td>Lake</td>
<td>77,200</td>
</tr>
<tr>
<td>Russia</td>
<td>Lake</td>
<td>11,780</td>
</tr>
<tr>
<td>Saudi Arabia</td>
<td>Desert</td>
<td>250,000</td>
</tr>
</tbody>
</table>

Grouping and Sorting Data

You can order grouped results with an ORDER BY clause. The following example takes the previous example and adds an ORDER BY clause to change the order of the Location column from ascending order to descending order:

libname sql 'SAS-library';

proc sql;
    title 'Total Square Miles of Deserts and Lakes';
    select Location, Type, sum(Area) as TotalArea format=comma16.
    from sql.features
    where type in ('Desert', 'Lake')
    group by Location, Type
    order by Location desc;
Grouping with Missing Values

Finding Grouping Errors Caused by Missing Values

When a column contains missing values, PROC SQL treats the missing values as a single group. This can sometimes provide unexpected results.

In this example, because the Sql.Countries table contains some missing values in the Continent column, the missing values combine to form a single group that has the total area of the countries that have a missing value in the Continent column:

```plaintext
libname sql 'SAS-library';
/* unexpected output */
proc sql outobs=12;
    title 'Areas of World Continents';
    select Name format=$25.,
           Continent,
           sum(Area) format=comma12. as TotalArea
    from sql.countries
    group by Continent,
    order by Continent, Name;
```

The output is incorrect because Bermuda, Iceland, and Kalaallit Nunaat are not actually part of the same continent. However, PROC SQL treats them that way because they all have a missing character value in the Continent column.
Finding Grouping Errors Caused by Missing Values (Unexpected Output)

To correct the query from the previous example, you can write a WHERE clause to exclude the missing values from the results:

```sql
/* modified output */

proc sql outobs=12;
  title 'Areas of World Continents';
  select Name format=$25.,
          Continent,
          sum(Area) format=comma12. as TotalArea
  from sql.countries
  where Continent is not missing
  group by Continent
  order by Continent, Name;
```
Filtering Grouped Data

Overview of Filtering Grouped Data

You can use a HAVING clause with a GROUP BY clause to filter grouped data. The HAVING clause affects groups in a way that is similar to how a WHERE clause affects individual rows. When you use a HAVING clause, PROC SQL displays only the groups that satisfy the HAVING expression.

Using a Simple HAVING Clause

The following example groups the features in the Sql.Features table by type and then displays only the numbers of islands, oceans, and seas:

```
libname sql 'SAS-library';

proc sql;
    title 'Numbers of Islands, Oceans, and Seas';
    select Type, count(*) as Number
    from sql.features
    group by Type
```
having Type in ('Island', 'Ocean', 'Sea')
order by Type;

Output 2.47   Using a Simple HAVING Clause

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Island</td>
<td>6</td>
</tr>
<tr>
<td>Ocean</td>
<td>4</td>
</tr>
<tr>
<td>Sea</td>
<td>13</td>
</tr>
</tbody>
</table>

Choosing between HAVING and WHERE

The differences between the HAVING clause and the WHERE clause are shown in the following table. Because you use the HAVING clause when you work with groups of data, queries that contain a HAVING clause usually also contain the following:

- a GROUP BY clause
- an aggregate function

TIP A HAVING clause is like a WHERE clause for groups.

Note: If you use a HAVING clause without a GROUP BY clause and if the query references at least one aggregate function, PROC SQL treats the input data as if it all comes from a single group of data.

Table 2.7   Differences between the HAVING Clause and WHERE Clause

<table>
<thead>
<tr>
<th>HAVING clause attributes</th>
<th>WHERE clause attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>is typically used to specify conditions for including or excluding groups of rows from a table.</td>
<td>is used to specify conditions for including or excluding individual rows from a table.</td>
</tr>
<tr>
<td>must follow the GROUP BY clause in a query, if used with a GROUP BY clause.</td>
<td>must precede the GROUP BY clause in a query, if used with a GROUP BY clause.</td>
</tr>
<tr>
<td>is affected by a GROUP BY clause, when there is no GROUP BY clause, the HAVING clause is treated like a WHERE clause.</td>
<td>is not affected by a GROUP BY clause.</td>
</tr>
<tr>
<td>is processed after the GROUP BY clause and any aggregate functions.</td>
<td>is processed before a GROUP BY clause, if there is one, and before any aggregate functions.</td>
</tr>
</tbody>
</table>

Using HAVING with Aggregate Functions

The following query returns the populations of all continents that have more than 15 countries:
libname sql 'SAS-library';

proc sql;
  title 'Total Populations of Continents with More than 15 Countries';
  select Continent,
      sum(Population) as TotalPopulation format=comma16.,
      count(*) as Count
  from sql.countries
  group by Continent
  having count(*) gt 15
  order by Continent;

The HAVING expression contains the COUNT function, which counts the number of rows within each group.

**Output 2.48 Using HAVING with the COUNT Function**

<table>
<thead>
<tr>
<th>Continent</th>
<th>TotalPopulation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>710,529,592</td>
<td>53</td>
</tr>
<tr>
<td>Asia</td>
<td>3,381,858,879</td>
<td>48</td>
</tr>
<tr>
<td>Central America and Caribbean</td>
<td>66,815,930</td>
<td>25</td>
</tr>
<tr>
<td>Europe</td>
<td>813,481,724</td>
<td>51</td>
</tr>
</tbody>
</table>

**Validating a Query**

The VALIDATE statement enables you to check the syntax of a query for correctness without submitting it to PROC SQL. PROC SQL displays a message in the log to indicate whether the syntax is correct.

`libname sql 'SAS-library';`

`proc sql;`
`  validate`
`    select Name, Statehood`
`    from sql.unitedstates`
`    where Statehood lt '01Jan1800'd;`

**Log 2.3 Validating a Query (Partial Log)**

```
3  proc sql;
4    validate
5        select Name, Statehood
6           from sql.unitedstates
7           where Statehood lt '01Jan1800'd;
NOTE: PROC SQL statement has valid syntax.
```

The following example shows an invalid query and the corresponding log message:
libname sql 'SAS-library';

proc sql;
  validate
    select Name, Statehood
    from sql.unitedstates
    where lt '01Jan1800'd;

Log 2.4  Validating an Invalid Query (Partial Log)

3  proc sql;
4   validate
5     select Name, Statehood
6     from sql.unitedstates
7     where lt '01Jan1800'd;
            22
76
ERROR 22-322: Syntax error, expecting one of the following: !, !!, &,*,**,
+,-, /, <, <=, <>, =, >=, ?, AND, CONTAINS, EQ, GE,
GROUP,
   GT, HAVING, LE, LIKE, LT, NE, OR, ORDER, ^=, |, ||, ~=.
ERROR 76-322: Syntax error, statement will be ignored.

NOTE: The SAS System stopped processing this step because of errors.
Chapter 3
Retrieving Data from Multiple Tables

Introduction

This chapter shows you how to perform the following tasks:

• select data from more than one table by joining the tables together
• use subqueries to select data from one table based on data values from another table
• combine the results of more than one query by using set operators

Note: Unless otherwise noted, the PROC SQL operations that are shown in this chapter apply to views as well as tables. For more information about views, see Chapter 4, “Creating and Updating Tables and Views,” on page 111.
Overview of Selecting Data from More Than One Table By Using Joins

The data that you need for a report could be located in more than one table. In order to select the data from the tables, join the tables in a query. Joining tables enables you to select data from multiple tables as if the data were contained in one table. Joins do not alter the original tables.

The most basic type of join is simply two tables that are listed in the FROM clause of a SELECT statement. The following query joins the two tables that are shown in Output 3.1 on page 74 and creates Output 3.2 on page 75.

```sql
proc sql;
  title 'Table One and Table Two';
  select *
    from one, two;
proc sql;
  title 'Table One';
  select * from one;
  title 'Table Two';
  select * from two;
quit;
```

Output 3.1  Table One, Table Two

### Table One

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table Two

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>
Joining tables in this way returns the Cartesian product of the tables. Each row from the first table is combined with every row from the second table. When you run this query, the following message is written to the SAS log:

**Log 3.1  Cartesian Product Log Message**

```
NOTE: The execution of this query involves performing one or more Cartesian product joins that can not be optimized.
```

The Cartesian product of large tables can be huge. Typically, you want a subset of the Cartesian product. You specify the subset by declaring the join type.

There are two types of joins:

- **Inner Joins** return a result table for all the rows in a table that have one or more matching rows in the other table or tables that are listed in the FROM clause.
- **Outer Joins** are inner joins that are augmented with rows that did not match with any row from the other table in the join. There are three types of outer joins: left, right, and full.

### Inner Joins

**Overview of Inner Joins**

An inner join returns only the subset of rows from the first table that matches rows from the second table. You can specify the columns that you want to be compared for matching values in a WHERE clause.

The following code adds a WHERE clause to the previous query. The WHERE clause specifies that only rows whose values in column X of Table One match values in column
X of Table Two should appear in the output. Compare this query's output to Output 3.2 on page 75.

```
proc sql;
  title 'Table One and Table Two';
  select * from one, two
  where one.x=two.x;
```

**Output 3.3 Table One and Table Two Joined**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

The output contains only one row because only one value in column X matches from each table. In an inner join, only the matching rows are selected. Outer joins can return nonmatching rows; they are covered in “Outer Joins” on page 84.

Note that the column names in the WHERE clause are prefixed by their table names. This is known as qualifying the column names, and it is necessary when you specify columns that have the same name from more than one table. Qualifying the column name avoids creating an ambiguous column reference.

**Using Table Aliases**

A table alias is a temporary, alternate name for a table. You specify table aliases in the FROM clause. Table aliases are used in joins to qualify column names and can make a query easier to read by abbreviating table names.

The following example compares the oil production of countries to their oil reserves by joining the OilProd and OilRsrvs tables on their Country columns. Because the Country columns are common to both tables, they are qualified with their table aliases. You could also qualify the columns by prefixing the column names with the table names.

**Note:** The AS keyword is optional.

```
libname sql 'SAS-library';

proc sql outobs=6;
  title 'Oil Production/Reserves of Countries';
  select * from sql.oilprod as p, sql.oilrsrvs as r
  where p.country = r.country;
```
Note that each table's Country column is displayed. Typically, once you have determined that a join is functioning correctly, you include just one of the matching columns in the `SELECT` clause.

**Specifying the Order of Join Output**
You can order the output of joined tables by one or more columns from either table. The next example's output is ordered in descending order by the `BarrelsPerDay` column. It is not necessary to qualify `BarrelsPerDay`, because the column exists only in the `OilProd` table.

```
libname sql 'SAS-library';

proc sql outobs=6;
title 'Oil Production/Reserves of Countries';
select p.country, barrelsperday 'Production', barrels 'Reserves'
  from sql.oilprod p, sql.oilrsrvs r
  where p.country = r.country
  order by barrelsperday desc;
```

**Output 3.5  Ordering the Output of Joined Tables**

<table>
<thead>
<tr>
<th>Country</th>
<th>Production</th>
<th>Reserves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saudi Arabia</td>
<td>9,000,000</td>
<td>260,000,000,000</td>
</tr>
<tr>
<td>United States of America</td>
<td>8,000,000</td>
<td>30,000,000,000</td>
</tr>
<tr>
<td>Iran</td>
<td>4,000,000</td>
<td>90,000,000,000</td>
</tr>
<tr>
<td>Norway</td>
<td>3,500,000</td>
<td>11,000,000,000</td>
</tr>
<tr>
<td>Mexico</td>
<td>3,400,000</td>
<td>50,000,000,000</td>
</tr>
<tr>
<td>China</td>
<td>3,000,000</td>
<td>25,000,000,000</td>
</tr>
</tbody>
</table>
Creating Inner Joins Using INNER JOIN Keywords

The INNER JOIN keywords can be used to join tables. The ON clause replaces the WHERE clause for specifying columns to join. PROC SQL provides these keywords primarily for compatibility with the other joins (OUTER, RIGHT, and LEFT JOIN). Using INNER JOIN with an ON clause provides the same functionality as listing tables in the FROM clause and specifying join columns with a WHERE clause.

This code produces the same output as the previous code but uses the INNER JOIN construction.

```sql
proc sql;
select p.country, barrelsperday 'Production', barrels 'Reserves'
from sql.oilprod p inner join sql.oilrsrvs r
on p.country = r.country
order by barrelsperday desc;
```

Joining Tables Using Comparison Operators

Tables can be joined by using comparison operators other than the equal sign (=) in the WHERE clause. For more information about comparison operators, see “Retrieving Rows Based on a Comparison” on page 45. In this example, all US cities in the USCityCoords table that are south of Cairo, Egypt, are selected. The compound WHERE clause specifies the city of Cairo in the WorldCityCoords table and joins USCityCoords and WorldCityCoords on their Latitude columns, using a less-than (<) operator.

```sql
libname sql 'SAS-library';

proc sql;
   title 'US Cities South of Cairo, Egypt';
   select us.City, us.State, us.Latitude, world.city, world.latitude
   from sql.worldcitycoords world, sql.uscitycoords us
   where world.city = 'Cairo' and
   us.latitude < world.latitude;
```

**Output 3.6 Using Comparison Operators to Join Tables**

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Latitude</th>
<th>City</th>
<th>Latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honolulu</td>
<td>HI</td>
<td>21</td>
<td>Cairo</td>
<td>30</td>
</tr>
<tr>
<td>Key West</td>
<td>FL</td>
<td>24</td>
<td>Cairo</td>
<td>30</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>26</td>
<td>Cairo</td>
<td>30</td>
</tr>
<tr>
<td>San Antonio</td>
<td>TX</td>
<td>29</td>
<td>Cairo</td>
<td>30</td>
</tr>
<tr>
<td>Tampa</td>
<td>FL</td>
<td>28</td>
<td>Cairo</td>
<td>30</td>
</tr>
</tbody>
</table>

When you run this query, the following message is written to the SAS log:

**Log 3.2 Comparison Query Log Message**

*NOTE: The execution of this query involves performing one or more Cartesian product joins that can not be optimized.*
Recall that you see this message when you run a query that joins tables without specifying matching columns in a WHERE clause. PROC SQL also displays this message whenever tables are joined by using an inequality operator.

**The Effects of Null Values on Joins**
Most database products treat nulls as distinct entities and do not match them in joins. PROC SQL treats nulls as missing values and as matches for joins. Any null will match with any other null of the same type (character or numeric) in a join.

The following example joins Table One and Table Two on column B. There are null values in column B of both tables. Notice in the output that the null value in row c of Table One matches all the null values in Table Two. This is probably not the intended result for the join.

```sql
proc sql;
  title 'One and Two Joined';
  select one.a 'One', one.b, two.a 'Two', two.b
     from one, two
     where one.b=two.b;
```

**Output 3.7  Joining Tables That Contain Null Values**

**Table One**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table Two**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>
**One and Two Joined**

<table>
<thead>
<tr>
<th>One</th>
<th>b</th>
<th>Two</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

In order to specify only the nonmissing values for the join, use the IS NOT MISSING operator:

```sql
proc sql;
select one.a 'One', one.b, two.a 'Two', two.b
from one, two
where one.b=two.b and
    one.b is not missing;
```

**Output 3.8  Results of Adding IS NOT MISSING to Joining Tables That Contain Null Values**

**Creating Multicolumn Joins**

When a row is distinguished by a combination of values in more than one column, use all the necessary columns in the join. For example, a city name could exist in more than one country. To select the correct city, you must specify both the city and country columns in the joining query's WHERE clause.

This example displays the latitude and longitude of capital cities by joining the Countries table with the WorldCityCoords table. To minimize the number of rows in the example output, the first part of the WHERE expression selects capitals with names that begin with the letter `L` from the Countries table.

```sql
libname sql 'SAS-library';
proc sql;
    title 'Coordinates of Capital Cities';
    select Capital format=$12., Name format=$12.,
               City format=$12., Country format=$12.,
               Latitude, Longitude
```
from sql.countries, sql.worldcitycoords
where Capital like 'L%' and
   Capital = City;

London occurs once as a capital city in the Countries table. However, in
WorldCityCoords, London is found twice: as a city in England and again as a city in
Canada. Specifying only \texttt{Capital = City} in the WHERE expression yields the
following incorrect output:

\textbf{Output 3.9} \textit{Selecting Capital City Coordinates (incorrect output)}

<table>
<thead>
<tr>
<th>Capital</th>
<th>Name</th>
<th>City</th>
<th>Country</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>England</td>
<td>London</td>
<td>Canada</td>
<td>43</td>
<td>-81</td>
</tr>
<tr>
<td>Lima</td>
<td>Peru</td>
<td>Lima</td>
<td>Peru</td>
<td>-13</td>
<td>-77</td>
</tr>
<tr>
<td>Lisbon</td>
<td>Portugal</td>
<td>Lisbon</td>
<td>Portugal</td>
<td>39</td>
<td>-10</td>
</tr>
<tr>
<td>London</td>
<td>England</td>
<td>London</td>
<td>England</td>
<td>51</td>
<td>0</td>
</tr>
</tbody>
</table>

Notice in the output that the inner join incorrectly matches London, England, to both
London, Canada, and London, England. By also joining the country name columns
together (Countries.Name to WorldCityCoords.Country), the rows match correctly.

libname sql 'SAS-library';

proc sql;
   title 'Coordinates of Capital Cities';
   select Capital format=$12., Name format=$12.,
      City format=$12., Country format=$12.,
      latitude, longitude
   from sql.countries, sql.worldcitycoords
   where Capital like 'L%' and
      Capital = City and
      Name = Country;

\textbf{Output 3.10} \textit{Selecting Capital City Coordinates (correct output)}

<table>
<thead>
<tr>
<th>Capital</th>
<th>Name</th>
<th>City</th>
<th>Country</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Paz</td>
<td>Bolivia</td>
<td>La Paz</td>
<td>Bolivia</td>
<td>-16</td>
<td>-69</td>
</tr>
<tr>
<td>Lima</td>
<td>Peru</td>
<td>Lima</td>
<td>Peru</td>
<td>-13</td>
<td>-77</td>
</tr>
<tr>
<td>Lisbon</td>
<td>Portugal</td>
<td>Lisbon</td>
<td>Portugal</td>
<td>39</td>
<td>-10</td>
</tr>
<tr>
<td>London</td>
<td>England</td>
<td>London</td>
<td>England</td>
<td>51</td>
<td>0</td>
</tr>
</tbody>
</table>
Selecting Data from More Than Two Tables

The data that you need could be located in more than two tables. For example, if you want to show the coordinates of the capitals of the states in the United States, then you need to join the UnitedStates table, which contains the state capitals, with the USCityEngine table, which contains the coordinates of cities in the United States. Because cities must be joined along with their states for an accurate join (similarly to the previous example), you must join the tables on both the city and state columns of the tables.

Joining the cities, by joining the UnitedStates.Capital column to the USCityEngine.City column, is straightforward. However, in the UnitedStates table, the Name column contains the full state name. In USCityEngine, the states are specified by their postal code. It is therefore impossible to directly join the two tables on their state columns. To solve this problem, it is necessary to use the PostalCodes table, which contains both the state names and their postal codes, as an intermediate table to make the correct relationship between UnitedStates and USCityEngine. The correct solution joins the UnitedStates.Name column to the PostalCodes.Name column (matching the full state names), and the PostalCodes.Code column to the USCityEngine.State column (matching the state postal codes).

```
libname sql 'SAS-library';

title 'Coordinates of State Capitals';
proc sql outobs=10;
  select us.Capital format=$15., us.Name 'State' format=$15.,
       pc.Code, c.Latitude, c.Longitude
  from sql.unitedstates us, sql.postalcodes pc,
       sql.uscitycoords c
  where us.Capital = c.City and
       us.Name = pc.Name and
       pc.Code = c.State;
```

Output 3.11  Selecting Data from More Than Two Tables

Coordinates of State Capitals

<table>
<thead>
<tr>
<th>Capital</th>
<th>State</th>
<th>Code</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montgomery</td>
<td>Alabama</td>
<td>AL</td>
<td>32</td>
<td>-86</td>
</tr>
<tr>
<td>Juneau</td>
<td>Alaska</td>
<td>AK</td>
<td>58</td>
<td>-134</td>
</tr>
<tr>
<td>Phoenix</td>
<td>Arizona</td>
<td>AZ</td>
<td>33</td>
<td>-113</td>
</tr>
<tr>
<td>Little Rock</td>
<td>Arkansas</td>
<td>AR</td>
<td>35</td>
<td>-92</td>
</tr>
<tr>
<td>Sacramento</td>
<td>California</td>
<td>CA</td>
<td>38</td>
<td>-121</td>
</tr>
<tr>
<td>Denver</td>
<td>Colorado</td>
<td>CO</td>
<td>40</td>
<td>-105</td>
</tr>
<tr>
<td>Hartford</td>
<td>Connecticut</td>
<td>CT</td>
<td>42</td>
<td>-73</td>
</tr>
<tr>
<td>Dover</td>
<td>Delaware</td>
<td>DE</td>
<td>39</td>
<td>-76</td>
</tr>
<tr>
<td>Tallahassee</td>
<td>Florida</td>
<td>FL</td>
<td>31</td>
<td>-84</td>
</tr>
<tr>
<td>Atlanta</td>
<td>Georgia</td>
<td>GA</td>
<td>34</td>
<td>-84</td>
</tr>
</tbody>
</table>
Showing Relationships within a Single Table Using Self-Joins

When you need to show comparative relationships between values in a table, it is sometimes necessary to join columns within the same table. Joining a table to itself is called a self-join, or reflexive join. You can think of a self-join as PROC SQL making an internal copy of a table and joining the table to its copy.

For example, the following code uses a self-join to select cities that have average yearly high temperatures equal to the average yearly low temperatures of other cities.

```
libname sql 'SAS-library';

proc sql;
  title "Cities' High Temps = Cities' Low Temps";
  select High.City format $12., High.Country format $12.,
         High.AvgHigh, ' | ',
         Low.City format $12., Low.Country format $12.,
         Low.AvgLow
  from sql.worldtemps High, sql.worldtemps Low
  where High.AvgHigh = Low.AvgLow and
       High.city ne Low.city and
       High.country ne Low.country;
```

Notice that the WorldTemps table is assigned two aliases, `High` and `Low`. Conceptually, this makes a copy of the table so that a join can be made between the table and its copy. The WHERE clause selects those rows that have high temperature equal to low temperature.

The WHERE clause also prevents a city from being joined to itself (`City ne City` and `Country ne Country`), although, in this case, it is highly unlikely that the high temperature would be equal to the low temperature for the same city.
Overview of Outer Joins

Outer joins are inner joins that are augmented with rows from one table that do not match any row from the other table in the join. The resulting output includes rows that match and rows that do not match from the join's source tables. Nonmatching rows have null values in the columns from the unmatched table. Use the ON clause instead of the WHERE clause to specify the column or columns on which you are joining the tables. However, you can continue to use the WHERE clause to subset the query result.

Including Nonmatching Rows with the Left Outer Join

A left outer join lists matching rows and rows from the left-hand table (the first table listed in the FROM clause) that do not match any row in the right-hand table. A left join is specified with the keywords LEFT JOIN and ON.

For example, to list the coordinates of the capitals of international cities, join the Countries table, which contains capitals, with the WorldCityCoords table, which
contains cities' coordinates, by using a left join. The left join lists all capitals, regardless of whether the cities exist in WorldCityCoords. Using an inner join would list only capital cities for which there is a matching city in WorldCityCoords.

libname sql 'SAS-library';

proc sql outobs=10;
  title 'Coordinates of Capital Cities';
  select Capital format=$20., Name 'Country' format=$20.,
    Latitude, Longitude
  from sql.countries a left join sql.worldcitycoords b
    on a.Capital = b.City and
    a.Name = b.Country
  order by Capital;

Output 3.13  Left Join of Countries and WorldCityCoords

<table>
<thead>
<tr>
<th>Capital</th>
<th>Country</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Islands</td>
<td>.  .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abu Dhabi</td>
<td>United Arab Emirates</td>
<td>.  .</td>
<td></td>
</tr>
<tr>
<td>Abuja</td>
<td>Nigeria</td>
<td>.  .</td>
<td></td>
</tr>
<tr>
<td>Accra</td>
<td>Ghana</td>
<td>5  0</td>
<td></td>
</tr>
<tr>
<td>Addis Ababa</td>
<td>Ethiopia</td>
<td>9  39</td>
<td></td>
</tr>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>37  3</td>
<td></td>
</tr>
<tr>
<td>Almaty</td>
<td>Kazakhstan</td>
<td>.  .</td>
<td></td>
</tr>
<tr>
<td>Amman</td>
<td>Jordan</td>
<td>32  36</td>
<td></td>
</tr>
<tr>
<td>Amsterdam</td>
<td>Netherlands</td>
<td>52  5</td>
<td></td>
</tr>
<tr>
<td>Andorra la Vella</td>
<td>Andorra</td>
<td>.  .</td>
<td></td>
</tr>
</tbody>
</table>

Including Nonmatching Rows with the Right Outer Join

A right join, specified with the keywords RIGHT JOIN and ON, is the opposite of a left join: nonmatching rows from the right-hand table (the second table listed in the FROM clause) are included with all matching rows in the output. This example reverses the join of the last example; it uses a right join to select all the cities from the WorldCityCoords table and displays the population only if the city is the capital of a country (that is, if the city exists in the Countries table).

libname sql 'SAS-library';

proc sql outobs=10;
title 'Populations of Capitals Only';
select City format=$20., Country 'Country' format=$20.,
    Population
from sql.countries right join sql.worldcitycoords
    on Capital = City and
    Name = Country
order by City;

Output 3.14  Right Join of Countries and WorldCityCoords

### Populations of Capitals Only

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abadan</td>
<td>Iran</td>
<td></td>
</tr>
<tr>
<td>Acapulco</td>
<td>Mexico</td>
<td></td>
</tr>
<tr>
<td>Accra</td>
<td>Ghana</td>
<td>17395511</td>
</tr>
<tr>
<td>Adana</td>
<td>Turkey</td>
<td></td>
</tr>
<tr>
<td>Addis Ababa</td>
<td>Ethiopia</td>
<td>59291170</td>
</tr>
<tr>
<td>Adelaide</td>
<td>Australia</td>
<td></td>
</tr>
<tr>
<td>Aden</td>
<td>Yemen</td>
<td></td>
</tr>
<tr>
<td>Ahmenabad</td>
<td>India</td>
<td></td>
</tr>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>28171132</td>
</tr>
<tr>
<td>Alice Springs</td>
<td>Australia</td>
<td></td>
</tr>
</tbody>
</table>

Selecting All Rows with the Full Outer Join

A full outer join, specified with the keywords FULL JOIN and ON, selects all matching and nonmatching rows. This example displays the first ten matching and nonmatching rows from the City and Capital columns of WorldCityCoords and Countries. Note that the number sign (#) is used as a line split character in the labels.

libname sql 'SAS-library';

proc sql outobs=10;
    title 'Populations and/or Coordinates of World Cities';
    select City '#City#(WorldCityCoords)' format=$20.,
        Capital '#Capital#(Countries)' format=$20.,
        Population, Latitude, Longitude
    from sql.countries full join sql.worldcitycoords
        on Capital = City and
        Name = Country;
**Output 3.15**  Full Outer Join of Countries and WorldCityCoords

<table>
<thead>
<tr>
<th>City (WorldCityCoords)</th>
<th>Capital (Countries)</th>
<th>Population</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abadan</td>
<td>.</td>
<td>146436</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Abu Dhabi</td>
<td>2818628</td>
<td>30</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Abuja</td>
<td>99062003</td>
<td>30</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Acapulco</td>
<td>.</td>
<td>17</td>
<td>-100</td>
<td></td>
</tr>
<tr>
<td>Accra</td>
<td>Accra</td>
<td>17395511</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Adana</td>
<td>.</td>
<td>37</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Addis Ababa</td>
<td>Addis Ababa</td>
<td>59291170</td>
<td>9</td>
<td>39</td>
</tr>
<tr>
<td>Adelaide</td>
<td>.</td>
<td>-35</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>Aden</td>
<td>.</td>
<td>13</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

**Specialty Joins**

**Overview of Specialty Joins**

Three types of joins—cross joins, union joins, and natural joins—are special cases of the standard join types.

**Including All Combinations of Rows with the Cross Join**

A cross join is a Cartesian product; it returns the product of two tables. Like a Cartesian product, a cross join's output can be limited by a WHERE clause.

This example shows a cross join of the tables One and Two:

```sas
data one;
  input X Y $;
  datalines;
  1 2
  2 3
  ;

data two;
  input W Z $;
  datalines;
  2 5
  3 6
  4 9
  ;
run;
```
proc sql;
    title 'Table One';
    select * from one;

    title 'Table Two';
    select * from two;

    title;
    quit;

Output 3.16  Tables One and Two

Table One

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table Two

<table>
<thead>
<tr>
<th>W</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

proc sql;
    title 'Table One and Table Two';
    select *
        from one cross join two;
Like a conventional Cartesian product, a cross join causes a note regarding Cartesian products in the SAS log.

**Including All Rows with the Union Join**

A union join combines two tables without attempting to match rows. All columns and rows from both tables are included. Combining tables with a union join is similar to combining them with the OUTER UNION set operator. (See “Combining Queries with Set Operators” on page 103.) A union join’s output can be limited by a WHERE clause.

This example shows a union join of the same One and Two tables that were used earlier to demonstrate a cross join:

```sql
proc sql;
select *
from one union join two;
```

**Matching Rows with a Natural Join**

A natural join automatically selects columns from each table to use in determining matching rows. With a natural join, PROC SQL identifies columns in each table that have the same name and type; rows in which the values of these columns are equal are returned as matching rows. The ON clause is implied.
This example produces the same results as the example in “Specifying the Order of Join Output” on page 77:

```sql
libname sql 'SAS-library';
proc sql outobs=6;
title 'Oil Production/Reserves of Countries';
select country, barrelsperday 'Production', barrels 'Reserve'
from sql.oilprod natural join sql.oilrsrvs
order by barrelsperday desc;
```

Output 3.19  Natural Inner Join of OilProd and OilRsrvs

<table>
<thead>
<tr>
<th>Country</th>
<th>Production</th>
<th>Reserve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saudi Arabia</td>
<td>9,000,000</td>
<td>260,000,000,000</td>
</tr>
<tr>
<td>United States of America</td>
<td>8,000,000</td>
<td>30,000,000,000</td>
</tr>
<tr>
<td>Iran</td>
<td>4,000,000</td>
<td>90,000,000,000</td>
</tr>
<tr>
<td>Norway</td>
<td>3,500,000</td>
<td>11,000,000,000</td>
</tr>
<tr>
<td>Mexico</td>
<td>3,400,000</td>
<td>50,000,000,000</td>
</tr>
<tr>
<td>China</td>
<td>3,000,000</td>
<td>25,000,000,000</td>
</tr>
</tbody>
</table>

The advantage of using a natural join is that the coding is streamlined. The ON clause is implied, and you do not need to use table aliases to qualify column names that are common to both tables. These two queries return the same results:

```sql
proc sql;
select a.W, a.X, Y, Z
from table1 a left join table2 b
on a.W=b.W and a.X=b.X
order by a.W;
```

```sql
proc sql;
select W, X, Y, Z
from table1 natural left join table2
order by W;
```

If you specify a natural join on tables that do not have at least one column with a common name and type, then the result is a Cartesian product. You can use a WHERE clause to limit the output.

Because the natural join makes certain assumptions about what you want to accomplish, you should know your data thoroughly before using it. You could get unexpected or incorrect results. For example, if you are expecting two tables to have only one column in common when they actually have two. You can use the FEEDBACK option to see exactly how PROC SQL is implementing your query. For information about the FEEDBACK option, see “Using PROC SQL Options to Create and Debug Queries” on page 138.

A natural join assumes that you want to base the join on equal values of all pairs of common columns. To base the join on inequalities or other comparison operators, use standard inner or outer join syntax.
Using the Coalesce Function in Joins

As you can see from the previous examples, the nonmatching rows in outer joins contain missing values. By using the COALESCE function, you can overlay columns so that only the row from the table that contains data is listed. Recall that COALESCE takes a list of columns as its arguments and returns the first nonmissing value that it encounters.

This example adds the COALESCE function to the previous example to overlay the Countries.Capital, WorldCityCoords.City, and Countries.Name columns. Countries.Name is supplied as an argument to COALESCE because some islands do not have capitals.

```
libname sql 'SAS-library';

proc sql outobs=10;
   title 'Populations and/or Coordinates of World Cities';
   select coalesce(Capital, City, Name) format=$20. 'City',
          coalesce(Name, Country) format=$20. 'Country',
          Population, Latitude, Longitude
   from sql.countries full join sql.worldcitycoords
      on Capital = City and
      Name = Country;
```

Output 3.20 Using COALESCE in Full Outer Join of Countries and WorldCityCoords

### Populations and/or Coordinates of World Cities

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Population</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Islands</td>
<td>Channel Islands</td>
<td>146436</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Abadan</td>
<td>Iran</td>
<td>.</td>
<td>30</td>
<td>48</td>
</tr>
<tr>
<td>Abu Dhabi</td>
<td>United Arab Emirates</td>
<td>2818628</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Abuja</td>
<td>Nigeria</td>
<td>99062003</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Acapulco</td>
<td>Mexico</td>
<td>.</td>
<td>17</td>
<td>-100</td>
</tr>
<tr>
<td>Accra</td>
<td>Ghana</td>
<td>17395511</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Adana</td>
<td>Turkey</td>
<td>.</td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td>Addis Ababa</td>
<td>Ethiopia</td>
<td>69291170</td>
<td>9</td>
<td>39</td>
</tr>
<tr>
<td>Adelaide</td>
<td>Australia</td>
<td>.</td>
<td>-35</td>
<td>138</td>
</tr>
<tr>
<td>Aden</td>
<td>Yemen</td>
<td>.</td>
<td>13</td>
<td>45</td>
</tr>
</tbody>
</table>

COALESCE can be used in both inner and outer joins. For more information about COALESCE, see “Replacing Missing Values” on page 35.
Comparing DATA Step Match-Merges with PROC SQL Joins

Overview of Comparing DATA Step Match-Merges with PROC SQL Joins

Many SAS users are familiar with using a DATA step to merge data sets. This section compares merges to joins. DATA step match-merges and PROC SQL joins can produce the same results. However, a significant difference between a match-merge and a join is that you do not have to sort the tables before you join them.

When All of the Values Match

When all of the values match in the BY variable and there are no duplicate BY variables, you can use an inner join to produce the same result as a match-merge. To demonstrate this result, here are two tables that have the column Flight in common. The values of Flight are the same in both tables:

<table>
<thead>
<tr>
<th>FltSuper</th>
<th>FltDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight</td>
<td>Supervisor</td>
</tr>
<tr>
<td>145</td>
<td>Kang</td>
</tr>
<tr>
<td>150</td>
<td>Miller</td>
</tr>
<tr>
<td>155</td>
<td>Evanko</td>
</tr>
</tbody>
</table>

FltSuper and FltDest are already sorted by the matching column Flight. A DATA step merge produces Output 3.21 on page 93.
With PROC SQL, presorting the data is not necessary. The following PROC SQL join gives the same result as that shown in Output 3.21 on page 93.

```sql
proc sql;
  title 'Table MERGED';
  select s.flight, Supervisor, Destination
  from fltsuper s, fltdest d
  where s.Flight=d.Flight;
```

**When Only Some of the Values Match**

When only some of the values match in the BY variable, you can use an outer join to produce the same result as a match-merge. To demonstrate this result, here are two tables that have the column Flight in common. The values of Flight are not the same in both tables:

<table>
<thead>
<tr>
<th>FltSuper</th>
<th>FltDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight</td>
<td>Supervisor</td>
</tr>
<tr>
<td>145</td>
<td>Kang</td>
</tr>
<tr>
<td>150</td>
<td>Miller</td>
</tr>
<tr>
<td>155</td>
<td>Evanko</td>
</tr>
<tr>
<td>157</td>
<td>Lei</td>
</tr>
</tbody>
</table>

A DATA step merge produces Output 3.22 on page 94:

```plaintext
data merged;
  merge fltsuper fltdest;
  by flight;
run;
proc print data=merged noobs;
  title 'Table Merged';
run;
```
Output 3.22  Merged Tables When Some of the Values Match

<table>
<thead>
<tr>
<th>Flight</th>
<th>Supervisor</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>Kang</td>
<td>Brussels</td>
</tr>
<tr>
<td>150</td>
<td>Miller</td>
<td>Paris</td>
</tr>
<tr>
<td>155</td>
<td>Evanko</td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>Lei</td>
<td>Seattle</td>
</tr>
<tr>
<td>165</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To get the same result with PROC SQL, use an outer join so that the query result will contain the nonmatching rows from the two tables. In addition, use the COALESCE function to overlay the Flight columns from both tables. The following PROC SQL join gives the same result as that shown in Output 3.22 on page 94:

```sql
proc sql;
  select coalesce(s.Flight,d.Flight) as Flight, Supervisor, Destination
  from fltsuper s full join fltdest d
  on s.Flight=d.Flight;
run;
```

When the Position of the Values Is Important

When you want to merge two tables and the position of the values is important, you might need to use a DATA step merge. To demonstrate this idea, here are two tables to consider:

<table>
<thead>
<tr>
<th>FltSuper</th>
<th>FltDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight</td>
<td>Supervisor</td>
</tr>
<tr>
<td>145</td>
<td>Kang</td>
</tr>
<tr>
<td>145</td>
<td>Ramirez</td>
</tr>
<tr>
<td>150</td>
<td>Miller</td>
</tr>
<tr>
<td>150</td>
<td>Picard</td>
</tr>
<tr>
<td>155</td>
<td>Evanko</td>
</tr>
<tr>
<td>157</td>
<td>Lei</td>
</tr>
</tbody>
</table>

For Flight 145, Kang matches with Brussels and Ramirez matches with Edmonton. Because the DATA step merges data based on the position of values in BY groups, the values of Supervisor and Destination match appropriately. A DATA step merge produces Output 3.23 on page 95:

```sas
data merged;
  merge fltsuper fltdest;
  by flight;
run;
proc print data=merged noobs;
  title 'Table Merged';
run;
```
PROC SQL does not process joins according to the position of values in BY groups. Instead, PROC SQL processes data only according to the data values. Here is the result of an inner join for FltSuper and FltDest:

```sql
proc sql;
  title 'Table Joined';
  select *
    from fltsuper s, fltdest d
  where s.Flight=d.Flight;
```

PROC SQL builds the Cartesian product and then lists the rows that meet the WHERE clause condition. The WHERE clause returns two rows for each supervisor, one row for each destination. Because Flight has duplicate values and there is no other matching column, there is no way to associate Kang only with Brussels, Ramirez only with Edmonton, and so on.
Using Subqueries to Select Data

A table join combines multiple tables into a new table. A subquery (enclosed in parentheses) selects rows from one table based on values in another table. A subquery, or inner query, is a query expression that is nested as part of another query expression. Depending on the clause that contains it, a subquery can return a single value or multiple values. Subqueries are most often used in the WHERE and the HAVING expressions.

**Single-Value Subqueries**

A single-value subquery returns a single row and column. It can be used in a WHERE or HAVING clause with a comparison operator. The subquery must return only one value, or else the query fails and an error message is printed to the log.

This query uses a subquery in its WHERE clause to select US states that have a population greater than Belgium. The subquery is evaluated first, and then it returns the population of Belgium to the outer query.

```
libname sql 'SAS-library';

proc sql;
  title 'U.S. States with Population Greater than Belgium';
  select Name 'State', population format=comma10.
    from sql.unitedstates
    where population gt
      (select population from sql.countries
       where name = "Belgium");
```

Internally, this is what the query looks like after the subquery has executed:

```
proc sql;
  title 'U.S. States with Population Greater than Belgium';
  select Name 'State', population format=comma10.
    from sql.unitedstates
    where population gt 10162614;
```

The outer query lists the states whose populations are greater than the population of Belgium.
### Output 3.25  Single-Value Subquery

#### U.S. States with Population Greater than Belgium

<table>
<thead>
<tr>
<th>State</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>31,518,948</td>
</tr>
<tr>
<td>Florida</td>
<td>13,814,408</td>
</tr>
<tr>
<td>Illinois</td>
<td>11,813,091</td>
</tr>
<tr>
<td>New York</td>
<td>18,377,334</td>
</tr>
<tr>
<td>Ohio</td>
<td>11,200,790</td>
</tr>
<tr>
<td>Pennsylvania</td>
<td>12,167,566</td>
</tr>
<tr>
<td>Texas</td>
<td>18,209,994</td>
</tr>
</tbody>
</table>

### Multiple-Value Subqueries

A multiple-value subquery can return more than one value from one column. It is used in a WHERE or HAVING expression that contains IN or a comparison operator that is modified by ANY or ALL. This example displays the populations of oil-producing countries. The subquery first returns all countries that are found in the OilProd table. The outer query then matches countries in the Countries table to the results of the subquery.

```sql
libname sql 'SAS-library';

proc sql outobs=5;
title 'Populations of Major Oil Producing Countries';
select name 'Country', Population format=comma15.
from sql.countries
where Name in (select Country from sql.oilprod);
```

### Output 3.26  Multiple-Value Subquery Using IN

#### Populations of Major Oil Producing Countries

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algeria</td>
<td>28,171,132</td>
</tr>
<tr>
<td>Canada</td>
<td>28,392,302</td>
</tr>
<tr>
<td>China</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>Egypt</td>
<td>59,912,259</td>
</tr>
<tr>
<td>Indonesia</td>
<td>202,393,859</td>
</tr>
</tbody>
</table>

If you use the NOT IN operator in this query, then the query result will contain all the countries that are not contained in the OilProd table.
libname sql 'SAS-library';

proc sql outobs=5;
   title 'Populations of NonMajor Oil Producing Countries';
   select name 'Country', Population format=comma15.
   from sql.countries
   where Name not in
       (select Country from sql.oilprod);

Output 3.27  Multiple-Value Subquery Using NOT IN

### Populations of NonMajor Oil Producing Countries

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>17,070,323</td>
</tr>
<tr>
<td>Albania</td>
<td>3,407,400</td>
</tr>
<tr>
<td>Andorra</td>
<td>64,634</td>
</tr>
<tr>
<td>Angola</td>
<td>9,901,050</td>
</tr>
<tr>
<td>Antigua and Barbuda</td>
<td>65,644</td>
</tr>
</tbody>
</table>

**Correlated Subqueries**

The previous subqueries have been simple subqueries that are self-contained and that execute independently of the outer query. A correlated subquery requires a value or values to be passed to it by the outer query. After the subquery runs, it passes the results back to the outer query. Correlated subqueries can return single or multiple values.

This example selects all major oil reserves of countries on the continent of Africa.

libname sql 'SAS-library';

proc sql;
   title 'Oil Reserves of Countries in Africa';
   select * from sql.oilrsrvs o
   where 'Africa' =
       (select Continent from sql.countries c
        where c.Name = o.Country);

The outer query selects the first row from the OilRsrvs table and then passes the value of the Country column, Algeria, to the subquery. At this point, the subquery internally looks like this:

(select Continent from sql.countries c
 where c.Name = 'Algeria');

The subquery selects that country from the Countries table. The subquery then passes the country's continent back to the WHERE clause in the outer query. If the continent is Africa, then the country is selected and displayed. The outer query then selects each subsequent row from the OilRsrvs table and passes the individual values of Country to the subquery. The subquery returns the appropriate values of Continent to the outer query for comparison in its WHERE clause.
Note that the WHERE clause uses an = (equal) operator. You can use an = (equal) operator if the subquery returns only a single value. However, if the subquery returns multiple values, then you must use IN or a comparison operator with ANY or ALL. For detailed information about the operators that are available for use with subqueries, see Chapter 7, “SQL Procedure,” on page 215.

**Output 3.28  Correlated Subquery**

<table>
<thead>
<tr>
<th>Country</th>
<th>Barrels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algeria</td>
<td>9,200,000,000</td>
</tr>
<tr>
<td>Egypt</td>
<td>4,000,000,000</td>
</tr>
<tr>
<td>Gabon</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>Libya</td>
<td>30,000,000,000</td>
</tr>
<tr>
<td>Nigeria</td>
<td>16,000,000,000</td>
</tr>
</tbody>
</table>

**Testing for the Existence of a Group of Values**

The EXISTS condition tests for the existence of a set of values. An EXISTS condition is true if any rows are produced by the subquery, and it is false if no rows are produced. Conversely, the NOT EXISTS condition is true when a subquery produces an empty table.

This example produces the same result as **Output 3.28 on page 99**. EXISTS checks for the existence of countries that have oil reserves on the continent of Africa. Note that the WHERE clause in the subquery now contains the condition Continent = 'Africa' that was in the outer query in the previous example.

```sql
libname sql 'SAS-library';

proc sql;
  title 'Oil Reserves of Countries in Africa';
  select * from sql.oilrsrvs o
  where exists
    (select Continent from sql.countries c
     where o.Country = c.Name and
     Continent = 'Africa');
```
Multiple Levels of Subquery Nesting

Subqueries can be nested so that the innermost subquery returns a value or values to be used by the next outer query. Then, that subquery’s value or values are used by the next outer query, and so on. Evaluation always begins with the innermost subquery and works outward.

This example lists cities in Africa that are in countries with major oil reserves.

1. The innermost query is evaluated first. It returns countries that are located on the continent of Africa.

2. The outer subquery is evaluated. It returns a subset of African countries that have major oil reserves by comparing the list of countries that was returned by the inner subquery against the countries in OilRsrvs.

3. Finally, the WHERE clause in the outer query lists the coordinates of the cities that exist in the WorldCityCoords table whose countries match the results of the outer subquery.

```sql
libname sql 'SAS-library';

proc sql;
   title 'Coordinates of African Cities with Major Oil Reserves';
   select * from sql.worldcitycoords
      where country in
         (select Country from sql.oilrsrvs o
            where o.Country in
               (select Name from sql.countries c
                  where c.Continent='Africa'));
```

---

**Output 3.29  Testing for the Existence of a Group of Values**

<table>
<thead>
<tr>
<th>Country</th>
<th>Barrels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algeria</td>
<td>9,200,000,000</td>
</tr>
<tr>
<td>Egypt</td>
<td>4,000,000,000</td>
</tr>
<tr>
<td>Gabon</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>Libya</td>
<td>30,000,000,000</td>
</tr>
<tr>
<td>Nigeria</td>
<td>16,000,000,000</td>
</tr>
</tbody>
</table>
Combining a Join with a Subquery

You can combine joins and subqueries in a single query. Suppose that you want to find the city nearest to each city in the USCityCoords table. The query must first select a city A, compute the distance from a city A to every other city, and finally select the city with the minimum distance from city A. This can be done by joining the USCityCoords table to itself (self-join) and then determining the closest distance between cities by using another self-join in a subquery.

This is the formula to determine the distance between coordinates:

\[
\text{SQRT}((\text{Latitude2} - \text{Latitude1})^2 + (\text{Longitude2} - \text{Longitude1})^2)
\]

Although the results of this formula are not exactly accurate because of the distortions caused by the curvature of the earth, they are accurate enough for this example to determine whether one city is closer than another.

```sql
libname sql 'SAS-library';

proc sql outobs=10;
  title 'Neighboring Cities';
  select a.City format=$10., a.State,
     a.Latitude 'Lat', a.Longitude 'Long',
     b.City format=$10., b.State,
     b.Latitude 'Lat', b.Longitude 'Long',
     sqrt(((b.Latitude-a.Latitude)**2) +
     (b.Longitude-a.Longitude)**2)) as dist format=6.1
  from sql.uscitycoords a, sql.uscitycoords b
  where a.city ne b.city and
calculated dist =
  (select min(sqrt(((d.Latitude-c.Latitude)**2) +
  (d.Longitude-c.Longitude)**2)))
  from sql.uscitycoords c, sql.uscitycoords d
  where c.city = a.city and
c.state = a.state and
d.city ne c.city)
order by a.city;
```
Combining a Join with a Subquery

The outer query joins the table to itself and determines the distance between the first city A1 in table A and city B2 (the first city that is not equal to city A1) in Table B. PROC SQL then runs the subquery. The subquery does another self-join and calculates the minimum distance between city A1 and all other cities in the table other than city A1. The outer query tests to see whether the distance between cities A1 and B2 is equal to the minimum distance that was calculated by the subquery. If they are equal, then a row that contains cities A1 and B2 with their coordinates and distance is written.

When to Use Joins and Subqueries

Use a join or a subquery anytime that you reference information from multiple tables. Joins and subqueries are often used together in the same query. In many cases, you can solve a data retrieval problem by using a join, a subquery, or both. Here are some guidelines for using joins and queries.

- If your report needs data that is from more than one table, then you must perform a join. Whenever multiple tables (or views) are listed in the FROM clause, those tables become joined.
- If you need to combine related information from different rows within a table, then you can join the table with itself.
- Use subqueries when the result that you want requires more than one query and each subquery provides a subset of the table involved in the query.
- If a membership question is asked, then a subquery is usually used. If the query requires a NOT EXISTS condition, then you must use a subquery because NOT EXISTS operates only in a subquery; the same principle holds true for the EXISTS condition.

---

### Neighboring Cities

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Lat</th>
<th>Long</th>
<th>City</th>
<th>State</th>
<th>Lat</th>
<th>Long</th>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td>NY</td>
<td>43</td>
<td>-74</td>
<td>Hartford</td>
<td>CT</td>
<td>42</td>
<td>-73</td>
<td>1.4</td>
</tr>
<tr>
<td>Albuquerque</td>
<td>NM</td>
<td>36</td>
<td>-106</td>
<td>Santa Fe</td>
<td>NM</td>
<td>36</td>
<td>-106</td>
<td>0.0</td>
</tr>
<tr>
<td>Amarillo</td>
<td>TX</td>
<td>35</td>
<td>-102</td>
<td>Carlsbad</td>
<td>NM</td>
<td>32</td>
<td>-104</td>
<td>3.6</td>
</tr>
<tr>
<td>Anchorage</td>
<td>AK</td>
<td>61</td>
<td>-150</td>
<td>Nome</td>
<td>AK</td>
<td>64</td>
<td>-165</td>
<td>15.3</td>
</tr>
<tr>
<td>Annapolis</td>
<td>MD</td>
<td>39</td>
<td>-77</td>
<td>Washington</td>
<td>DC</td>
<td>39</td>
<td>-77</td>
<td>0.0</td>
</tr>
<tr>
<td>Atlanta</td>
<td>GA</td>
<td>34</td>
<td>-84</td>
<td>Knoxville</td>
<td>TN</td>
<td>36</td>
<td>-84</td>
<td>2.0</td>
</tr>
<tr>
<td>Augusta</td>
<td>ME</td>
<td>44</td>
<td>-70</td>
<td>Portland</td>
<td>ME</td>
<td>44</td>
<td>-70</td>
<td>0.0</td>
</tr>
<tr>
<td>Austin</td>
<td>TX</td>
<td>30</td>
<td>-98</td>
<td>San Antonio</td>
<td>TX</td>
<td>29</td>
<td>-98</td>
<td>1.0</td>
</tr>
<tr>
<td>Baker</td>
<td>OR</td>
<td>45</td>
<td>-118</td>
<td>Lewiston</td>
<td>ID</td>
<td>46</td>
<td>-117</td>
<td>1.4</td>
</tr>
<tr>
<td>Baltimore</td>
<td>MD</td>
<td>39</td>
<td>-76</td>
<td>Dover</td>
<td>DE</td>
<td>39</td>
<td>-76</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Many queries can be formulated as joins or subqueries. Although the PROC SQL query optimizer changes some subqueries to joins, a join is generally more efficient to process.

Combining Queries with Set Operators

Working with Two or More Query Results

PROC SQL can combine the results of two or more queries in various ways by using the following set operators:

UNION
produces all unique rows from both queries.

EXCEPT
produces rows that are part of the first query only.

INTERSECT
produces rows that are common to both query results.

OUTER UNION
concatenates the query results.

The operator is used between the two queries, for example:

```sql
select columns from table
set-operator
select columns from table;
```

Place a semicolon after the last SELECT statement only. Set operators combine columns from two queries based on their position in the referenced tables without regard to the individual column names. Columns in the same relative position in the two queries must have the same data types. The column names of the tables in the first query become the column names of the output table. For information about using set operators with more than two query results, see the Chapter 7, “SQL Procedure,” on page 215. The following optional keywords give you more control over set operations:

ALL
does not suppress duplicate rows. When the keyword ALL is specified, PROC SQL does not make a second pass through the data to eliminate duplicate rows. Thus, using ALL is more efficient than not using it. ALL is not allowed with the OUTER UNION operator.

CORRESPONDING (CORR)
overlays columns that have the same name in both tables. When used with EXCEPT, INTERSECT, and UNION, CORR suppresses columns that are not in both tables.

Each set operator is described and used in an example based on the following two tables.
Whereas join operations combine tables horizontally, set operations combine tables vertically. Therefore, the set diagrams that are included in each section are displayed vertically.

**Producing Unique Rows from Both Queries (UNION)**

The UNION operator combines two query results. It produces all the unique rows that result from both queries. That is, it returns a row if it occurs in the first table, the second, or both. UNION does not return duplicate rows. If a row occurs more than once, then only one occurrence is returned.

```sql
proc sql;
  title 'A UNION B';
  select * from sql.a
  union
  select * from sql.b;
```
You can use the ALL keyword to request that duplicate rows remain in the output.

```sql
proc sql;
  title 'A UNION ALL B';
  select * from sql.a
  union all
  select * from sql.b;
```

**Output 3.34  Producing Rows from Both Queries (UNION ALL)**

The EXCEPT operator returns rows that result from the first query but not from the second query. In this example, the row that contains the values 3 and **three** exists in the first query (table A) only and is returned by EXCEPT.
proc sql;
  title 'A EXCEPT B';
  select * from sql.a
  except
  select * from sql.b;

Output 3.35  Producing Rows That Are in Only the First Query Result (EXCEPT)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>three</td>
</tr>
</tbody>
</table>

Note that the duplicated row in Table A containing the values 2 and two does not appear in the output. EXCEPT does not return duplicate rows that are unmatched by rows in the second query. Adding ALL keeps any duplicate rows that do not occur in the second query.

proc sql;
  title 'A EXCEPT ALL B';
  select * from sql.a
  except all
  select * from sql.b;

Output 3.36  Producing Rows That Are in Only the First Query Result (EXCEPT ALL)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
</tr>
</tbody>
</table>

**Producing Rows That Belong to Both Query Results (INTERSECT)**

The INTERSECT operator returns rows from the first query that also occur in the second.

proc sql;
  title 'A INTERSECT B';
  select * from sql.a
  intersect
  select * from sql.b;
Combining Queries with Set Operators

**Output 3.37  Producing Rows That Belong to Both Query Results (INTERSECT)**

A INTERSECT B

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
</tr>
</tbody>
</table>

The output of an INTERSECT ALL operation contains the rows produced by the first query that are matched one-to-one with a row produced by the second query. In this example, the output of INTERSECT ALL is the same as INTERSECT.

**Concatenating Query Results (OUTER UNION)**

The OUTER UNION operator concatenates the results of the queries. This example concatenates tables A and B.

```sql
proc sql;
  title 'A OUTER UNION B';
  select * from sql.a
  outer union
  select * from sql.b;
```

**Output 3.38  Concatenating the Query Results (OUTER UNION)**

A OUTER UNION B

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>three</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>.</td>
<td></td>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>.</td>
<td></td>
<td>4</td>
<td>four</td>
</tr>
</tbody>
</table>
Notice that OUTER UNION does not overlay columns from the two tables. To overlay columns in the same position, use the CORRESPONDING keyword.

```sql
proc sql;
  title 'A OUTER UNION CORR B';
  select * from sql.a
  outer union corr
  select * from sql.b;
```

**Output 3.39 Concatenating the Query Results (OUTER UNION CORR)**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>three</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>one</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>two</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>four</td>
<td></td>
</tr>
</tbody>
</table>

**Producing Rows from the First Query or the Second Query**

There is no keyword in PROC SQL that returns unique rows from the first and second table, but not rows that occur in both. Here is one way that you can simulate this operation:

```sql
(query1 except query2)
union
(query2 except query1)
```

This example shows how to use this operation.

```sql
proc sql;
  title 'A EXCLUSIVE UNION B';
  (select * from sql.a
   except
   select * from sql.b)
  union
  (select * from sql.b
   except
```
The first EXCEPT returns one unique row from the first table (table A) only. The second EXCEPT returns one unique row from the second table (table B) only. The middle UNION combines the two results. Thus, this query returns the row from the first table that is not in the second table, as well as the row from the second table that is not in the first table.
Chapter 4
Creating and Updating Tables and Views

Introduction ................................................................. 112
Creating Tables ......................................................... 112
    Creating Tables from Column Definitions ....................... 112
    Creating Tables from a Query Result ................................. 113
    Creating Tables like an Existing Table .............................. 115
    Copying an Existing Table ........................................... 115
    Using Data Set Options ............................................. 115
Inserting Rows into Tables ........................................... 116
    Inserting Rows with the SET Clause ................................. 116
    Inserting Rows with the VALUES Clause ............................ 117
    Inserting Rows with a Query ....................................... 118
Updating Data Values in a Table ..................................... 120
    Updating All Rows in a Column with the Same Expression ........ 120
    Updating Rows in a Column with Different Expressions ............ 121
    Handling Update Errors ............................................ 122
Deleting Rows ............................................................ 123
Altering Columns ....................................................... 123
    Adding a Column .................................................. 123
    Modifying a Column ............................................... 125
    Deleting a Column ................................................ 126
Creating an Index ...................................................... 127
    Using PROC SQL to Create Indexes ................................. 127
    Tips for Creating Indexes ......................................... 127
    Deleting Indexes .................................................. 128
Deleting a Table .......................................................... 128
Using SQL Procedure Tables in SAS Software ....................... 128
Creating and Using Integrity Constraints in a Table ................. 129
Creating and Using PROC SQL Views .................................. 131
    Overview of Creating and Using PROC SQL Views .................. 131
    Creating Views .................................................... 131
    Describing a View ................................................ 132
    Updating a View .................................................. 132
    Embedding a LIBNAME in a View .................................... 133
    Deleting a View ................................................... 134
    Specifying In-Line Views ......................................... 134
Introduction

This chapter shows you how to perform the following tasks:

• create a table
• update tables
• alter existing tables
• delete a table
• create indexes
• use integrity constraints in table creation
• create views

Creating Tables

The CREATE TABLE statement enables you to create tables without rows from column definitions or to create tables from a query result. You can also use CREATE TABLE to copy an existing table.

Creating Tables from Column Definitions

You can create a new table without rows by using the CREATE TABLE statement to define the columns and their attributes. You can specify a column's name, type, length, informat, format, and label.

The following CREATE TABLE statement creates the NewStates table:

```sql
proc sql;
create table sql.newstates
(state char(2),            /* 2-character column for               */
/* state abbreviation                   */
date num                 /* column for date of entry into the US */
informat=date9.      /* with an informat                     */
format=date9.,      /* and format of DATE9.                 */
population num);         /* column for population                */
```

The table NewStates has three columns and 0 rows. The `char(2)` modifier is used to change the length for State.

Use the DESCRIBE TABLE statement to verify that the table exists and to see the column attributes. The following DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log:

```sql
proc sql;
describe table sql.newstates;
```
Log 4.1  Table Created from Column Definitions

```
1  proc sql;
2     describe table sql.newstates;
NOTE: SQL table SQL.NEWSTATES was created like:

create table SQL.NEWSTATES( bufsize=4096 )
(
    state char(2),
    date num format=DATE9. informat=DATE9.,
    population num
);
```

DESCRIBE TABLE writes a CREATE TABLE statement to the SAS log even if you did not create the table with the CREATE TABLE statement. You can also use the CONTENTS statement in the DATASETS procedure to get a description of NewStates.

Creating Tables from a Query Result

To create a PROC SQL table from a query result, use a CREATE TABLE statement with the AS keyword, and place it before the SELECT statement. When a table is created this way, its data is derived from the table or view that is referenced in the query’s FROM clause. The new table’s column names are as specified in the query’s SELECT clause list. The new table’s column attributes (the type, length, informat, format, and extended attributes) are the same as the selected source columns.

Note: Extended attributes are not copied to tables that are created using multi-table joins or outer joins. When UNION, INTERSECT, or minus operators are used, extended attributes are copied only if the table that is listed before the UNION, INTERSECT, or minus operator has extended attributes.

The following CREATE TABLE statement creates the Densities table from the Countries table. The newly created table is not displayed in SAS output unless you query the table. Note the use of the OUTOBS option, which limits the size of the Densities table to 10 rows.

```
libname sql 'SAS-library';

proc sql outobs=10;
    title 'Densities of Countries';
    create table sql.densities as
    select Name 'Country' format $15.,
            Population format=comma10.0,
            Area as SquareMiles,
            Population/Area format=6.2 as Density
    from sql.countries;

    select * from sql.densities;
```
The following DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log:

```sas
proc sql;
    describe table sql.densities;
```

**Log 4.2  SAS Log for DESCRIBE TABLE Statement for DENSITIES**

```
NOTE: SQL table SQL.DENSITIES was created like:
create table SQL.DENSITIES( bufsize=8192 )
  {
    Name char(35) format=$15. informat=$35. label='Country',
    Population num format=COMMA10. informat=BEST8. label='Population',
    SquareMiles num format=BEST8. informat=BEST8.,
    Density num format=6.2
  );
```

In this form of the CREATE TABLE statement, assigning an alias to a column renames the column, but assigning a label does not. In this example, the Area column has been renamed to SquareMiles, and the calculated column has been named Density. However, the Name column retains its name, and its display label is **Country**.
Creating Tables like an Existing Table

To create an empty table that has the same columns and attributes as an existing table or view, use the LIKE clause in the CREATE TABLE statement. In the following example, the CREATE TABLE statement creates the NewCountries table with six columns and 0 rows and with the same column attributes as those in Countries. The DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log:

```
proc sql;
   create table sql.newcountries
     like sql.countries;

describe table sql.newcountries;
```

**Log 4.3  SAS Log for DESCRIBE TABLE Statement for NewCountries**

```
NOTE: SQL table SQL.NEWCOUNTRIES was created like:
create table SQL.NEWCOUNTRIES( bufsize=65536 )
(
   Name char(35) format=$35. informat=$35.,
   Capital char(35) format=$35. informat=$35. label='Capital',
   Population num format=BEST8. informat=BEST8. label='Population',
   Area num format=BEST8. informat=BEST8.,
   Continent char(30) format=$30. informat=$30. label='Continent',
   UNDate num format=YEAR4.
);
```

Copying an Existing Table

A quick way to copy a table using PROC SQL is to use the CREATE TABLE statement with a query that returns an entire table. This example creates Countries1, which contains a copy of all the columns and rows that are in Countries:

```
create table countries1 as
   select * from sql.countries;
```

Using Data Set Options

You can use SAS data set options in the CREATE TABLE statement. The following CREATE TABLE statement creates Countries2 from Countries. The DROP= option deletes the UNDate column, and UNDate does not become part of Countries2:

```
create table countries2 as
   select * from sql.countries(drop=UNDate);
```
Inserting Rows into Tables

Use the INSERT statement to insert data values into tables. The INSERT statement first adds a new row to an existing table, and then inserts the values that you specify into the row. You specify values by using a SET clause or VALUES clause. You can also insert the rows resulting from a query. Under most conditions, you can insert data into tables through PROC SQL and SAS/ACCESS views. For more information, see “Updating a View” on page 132.

Inserting Rows with the SET Clause

With the SET clause, you assign values to columns by name. The columns can appear in any order in the SET clause. The following INSERT statement uses multiple SET clauses to add two rows to NewCountries:

```sql
libname sql 'SAS-library';
/* Create the newcountries table. */
proc sql;
  create table sql.newcountries
    like sql.countries;
/* Insert all of the rows from countries into newcountries based on a population of 130000000. */
proc sql;
  insert into sql.newcountries
    select * from sql.countries
    where population ge 130000000;
/* Insert 2 new rows in the newcountries table. */
/* Print the table. */
proc sql;
  insert into sql.newcountries
    set name='Bangladesh',
    capital='Dhaka',
    population=126391060
  set name='Japan',
    capital='Tokyo',
    population=126352003;
  title "World's Largest Countries";
  select name format=$20.,
    capital format=$15.,
    population format=comma15.0
  from sql.newcountries;
```
Note the following features of SET clauses:

- As with other SQL clauses, use commas to separate columns. In addition, you must use a semicolon after the last SET clause only.
- If you omit data for a column, then the value in that column is a missing value.
- To specify that a value is missing, use a blank in single quotation marks for character values and a period for numeric values.

### Inserting Rows with the VALUES Clause

With the VALUES clause, you assign values to a column by position. The following INSERT statement uses multiple VALUES clauses to add rows to NewCcountries. Recall that NewCountries has six columns, so it is necessary to specify a value or an appropriate missing value for all six columns. See the results of the DESCRIBE TABLE statement in “Creating Tables like an Existing Table” on page 115 for information about the columns of NewCountries.

```sql
libname sql 'SAS-library';
proc sql;
    insert into sql.newcountries
    values ('Pakistan', 'Islamabad', 123060000, ., ' ', .)
    values ('Nigeria', 'Lagos', 99062000, ., ' ', .);
    title "World's Largest Countries";
    select name format=$20.,
           capital format=$15.,
           population format=comma15.0
    from sql.newcountries;
```
Output 4.3  Rows Inserted with the VALUES Clause

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>Brasilia</td>
<td>160,310,357</td>
</tr>
<tr>
<td>China</td>
<td>Beijing</td>
<td>1,202,215,077</td>
</tr>
<tr>
<td>India</td>
<td>New Delhi</td>
<td>929,009,120</td>
</tr>
<tr>
<td>Indonesia</td>
<td>Jakarta</td>
<td>202,393,859</td>
</tr>
<tr>
<td>Russia</td>
<td>Moscow</td>
<td>151,089,979</td>
</tr>
<tr>
<td>United States</td>
<td>Washington</td>
<td>263,294,808</td>
</tr>
<tr>
<td>Bangladesh</td>
<td>Dhaka</td>
<td>126,391,060</td>
</tr>
<tr>
<td>Japan</td>
<td>Tokyo</td>
<td>126,352,003</td>
</tr>
<tr>
<td>Pakistan</td>
<td>Islamabad</td>
<td>123,060,000</td>
</tr>
<tr>
<td>Nigeria</td>
<td>Lagos</td>
<td>99,062,000</td>
</tr>
</tbody>
</table>

Note the following features of VALUES clauses:

- As with other SQL clauses, use commas to separate columns. In addition, you must use a semicolon after the last VALUES clause only.

- If you omit data for a column without indicating a missing value, then you receive an error message and the row is not inserted.

- To specify that a value is missing, use a space in single quotation marks for character values and a period for numeric values.

Inserting Rows with a Query

You can insert the rows from a query result into a table. The following query returns rows for large countries (more than 130 million in population) from the Countries table. The INSERT statement adds the data to the empty table NewCountries, which was created earlier in “Creating Tables like an Existing Table” on page 115:

```
libname sql 'SAS-library';

proc sql;
创建工作sql.newcountries
   like sql.countries;

proc sql;
   title "World's Largest Countries";
   insert into sql.newcountries
      select * from sql.countries
         where population ge 130000000;

   select name format=$20.,
      capital format=$15.,
      population format=comma15.0
```

Because the target table that is specified by the INSERT statement does not specify specific columns to insert in the Sql.Newcountries table, you must select all columns in the query. If the query does not select every column that exists in the target table, an error occurs and the row is not inserted. The UNDO_POLICY= option does not prevent the error. For more information about how PROC SQL handles errors during data insertions, see “Handling Update Errors” on page 122.

To insert rows by using a query for a subset of columns from the source table, specify all column names in a comma-separated list, enclosed in parentheses, in the INSERT statement. In the SELECT clause, specify column names that correspond to the columns of the INSERT statement. The order and number of columns must match in the INSERT statement and in the SELECT clause.

```
proc sql;
 create table sql.newcountries
   like sql.countries;

proc sql;
   title "World's Largest Countries";
   insert into sql.newcountries (Name,Population)
      select Name,Population from sql.countries
      where population ge 130000000;

   select name format=$20., population format=comma15.0
      from sql.newcountries;
```
An error occurs if the query selects more columns than what exists for column names in the table that is specified in the INSERT statement.

### Updating Data Values in a Table

You can use the UPDATE statement to modify data values in tables and in the tables that underlie PROC SQL and SAS/ACCESS views. For more information about updating views, see “Updating a View” on page 132. The UPDATE statement updates data in existing columns; it does not create new columns. To add new columns, see “Altering Columns” on page 123 and “Creating New Columns” on page 27. The examples in this section update the original NewCountries table.

#### Updating All Rows in a Column with the Same Expression

The following UPDATE statement increases all populations in the NewCountries table by 5%:

```sql
/* code for all examples in updating section */
libname sql 'SAS-library';

proc sql;
  create table sql.newcountries like sql.countries;
  insert into sql.newcountries
  select * from sql.countries
  where population ge 13000000;
  update sql.newcountries
  set population=population*1.05;
  title "Updated Population Values";
  select name format=$20.,
     capital format=$15.,
     population format=comma15.0
  from sql.newcountries;
```

Output 4.5  Smaller Number of Columns in Rows Inserted with a Query

<table>
<thead>
<tr>
<th>World's Largest Countries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>Brazil</td>
</tr>
<tr>
<td>China</td>
</tr>
<tr>
<td>India</td>
</tr>
<tr>
<td>Indonesia</td>
</tr>
<tr>
<td>Russia</td>
</tr>
<tr>
<td>United States</td>
</tr>
</tbody>
</table>

An error occurs if the query selects more columns than what exists for column names in the table that is specified in the INSERT statement.
Updating Rows in a Column with Different Expressions

To update some, but not all, of a column's values, use a WHERE expression in the UPDATE statement. You can use multiple UPDATE statements, each of which can contain a different WHERE expression. Each UPDATE statement can have only one WHERE expression. The following UPDATE statements result in different population increases for different countries in the NewCountries table.

```sas
libname sql 'SAS-library';

proc sql;
  create table sql.newcountries like sql.countries;
  insert into sql.newcountries
  select * from sql.countries
  where population ge 130000000;
proc sql;
  update sql.newcountries
  set population=population*1.05
  where name like 'B%';

update sql.newcountries
  set population=population*1.07
  where name in ('China', 'Russia');

title "Selectively Updated Population Values";
select name format=$20.,
capital format=$15.,
population format=comma15.0
from sql.newcountries;
```
Output 4.7 Selectively Updating a Column

```
update sql.newcountries
  set population = population * case when name like 'B%' then 1.05
                             when name in ('China', 'Russia') then 1.07
                             else 1
                         end;
```

If the WHEN clause is true, then the corresponding THEN clause returns a value that the SET clause then uses to complete its expression. In this example, when Name starts with the letter B, the SET expression becomes `population = population * 1.05`.

**CAUTION:**

Make sure that you specify the ELSE clause. If you omit the ELSE clause, then each row that is not described in one of the WHEN clauses receives a missing value for the column that you are updating. This happens because the CASE expression supplies a missing value to the SET clause, and the Population column is multiplied by a missing value, which produces a missing value.

### Handling Update Errors

While you are updating or inserting rows in a table, you might receive an error message that the update or insert cannot be performed. By using the `UNDO_POLICY=` option, you can control whether the changes that have already been made will be permanent.

The `UNDO_POLICY=` option in the PROC SQL and RESET statements determines how PROC SQL handles the rows that have been inserted or updated by the current INSERT or UPDATE statement up to the point of error.

- `UNDO_POLICY=REQUIRED` is the default. It undoes all updates or inserts up to the point of error.
- `UNDO_POLICY=NONE` does not undo any updates or inserts.
- `UNDO_POLICY=OPTIONAL` undoes any updates or inserts that it can undo reliably.

---

### Selectively Updated Population Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>Brasilia</td>
<td>168,325,875</td>
</tr>
<tr>
<td>China</td>
<td>Beijing</td>
<td>1,286,370,132</td>
</tr>
<tr>
<td>India</td>
<td>New Delhi</td>
<td>929,009,120</td>
</tr>
<tr>
<td>Indonesia</td>
<td>Jakarta</td>
<td>202,393,859</td>
</tr>
<tr>
<td>Russia</td>
<td>Moscow</td>
<td>161,666,278</td>
</tr>
<tr>
<td>United States</td>
<td>Washington</td>
<td>263,294,808</td>
</tr>
</tbody>
</table>
Deleting Rows

The DELETE statement deletes one or more rows in a table or in a table that underlies a PROC SQL or SAS/ACCESS view. For more information about deleting rows from views, see “Updating a View” on page 132. The following DELETE statement deletes the names of countries that begin with the letter R:

```sql
/* Create and populate Newcountries */
proc sql;
  create table sql.newcountries like sql.countries;
  insert into sql.newcountries
    select * from sql.countries
    where population ge 13000000;
proc sql;
  delete
    from sql.newcountries
    where name like 'R%';
```

A note in the SAS log tells you how many rows were deleted.

Log 4.4  SAS Log for the DELETE Statement

```
NOTE: 1 row was deleted from SQL.NEWCOUNTRIES.
```

Note: For PROC SQL tables, SAS deletes the data in the rows but retains the space in the table.

CAUTION:

If you omit a WHERE clause, then the DELETE statement deletes all the rows from the specified table or the table that is described by a view. The rows are not deleted from the table until it is re-created.

Altering Columns

The ALTER TABLE statement adds, modifies, and deletes columns in existing tables. You can use the ALTER TABLE statement with tables only; it does not work with views. A note appears in the SAS log that describes how you have modified the table.

Adding a Column

The ADD clause adds a new column to an existing table. You must specify the column name and data type. You can also specify a length (LENGTH=), format (FORMAT=), informat (INFORMAT=), and a label (LABEL=). The following ALTER TABLE statement adds the numeric data column Density to the NewCountries table:

```sql
proc sql;
  create table sql.newcountries like sql.countries;
  insert into sql.newcountries
```
```sql
select * from sql.countries
   where population ge 130000000;

proc sql;
   alter table sql.newcountries
     add density num label='Population Density' format=6.2;

title "Population Density Table";
select name format=$20.,
    capital format=$15.,
    population format=comma15.0,
    density
from sql.newcountries;
```

**Output 4.8  Adding a New Column**

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
<th>Population Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>Brasilia</td>
<td>160,310,357</td>
<td>.</td>
</tr>
<tr>
<td>China</td>
<td>Beijing</td>
<td>1,202,215,077</td>
<td>.</td>
</tr>
<tr>
<td>India</td>
<td>New Delhi</td>
<td>929,009,120</td>
<td>.</td>
</tr>
<tr>
<td>Indonesia</td>
<td>Jakarta</td>
<td>202,393,859</td>
<td>.</td>
</tr>
<tr>
<td>Russia</td>
<td>Moscow</td>
<td>151,089,979</td>
<td>.</td>
</tr>
<tr>
<td>United States</td>
<td>Washington</td>
<td>263,294,808</td>
<td>.</td>
</tr>
</tbody>
</table>

The new column is added to NewCountries, but it has no data values. The following UPDATE statement changes the missing values for Density from missing to the appropriate population densities for each country:

```sql
proc sql;
   update sql.newcountries
     set density=population/area;

title "Population Density Table";
select name format=$20.,
    capital format=$15.,
    population format=comma15.0,
    density
from sql.newcountries;
```
Output 4.9  Filling in the New Column's Values

### Population Density Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
<th>Population Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>Brasilia</td>
<td>160,310,357</td>
<td>48.78</td>
</tr>
<tr>
<td>China</td>
<td>Beijing</td>
<td>1,202,215,077</td>
<td>325.27</td>
</tr>
<tr>
<td>India</td>
<td>New Delhi</td>
<td>929,009,120</td>
<td>759.86</td>
</tr>
<tr>
<td>Indonesia</td>
<td>Jakarta</td>
<td>202,393,859</td>
<td>273.10</td>
</tr>
<tr>
<td>Russia</td>
<td>Moscow</td>
<td>151,089,979</td>
<td>22.92</td>
</tr>
<tr>
<td>United States</td>
<td>Washington</td>
<td>263,294,808</td>
<td>69.52</td>
</tr>
</tbody>
</table>

For more information about how to change data values, see “Updating Data Values in a Table” on page 120.

You can accomplish the same update by using an arithmetic expression to create the Population Density column as you re-create the table:

```sql
proc sql;
create table sql.newcountries as
   select *, population/area as density
   label='Population Density'
   format=6.2
   from sql.newcountries;
```

See “Calculating Values” on page 29 for another example of creating columns with arithmetic expressions.

### Modifying a Column

You can use the MODIFY clause to change the width, informat, format, and label of a column. To change a column's name, use the RENAME= data set option. You cannot change a column's data type by using the MODIFY clause.

The following MODIFY clause permanently changes the format for the Population column:

```sql
proc sql;
create table sql.newcountries like sqlcountries;
create table sql.newcountries as
   select * from sql.countries
   where population ge 13000000;
```

```sql
proc sql;
title "World's Largest Countries";
alter table sql.newcountries
   modify population format=comma15.;
select name, population from sql.newcountries;
```
You might have to change a column's width (and format) before you can update the column. For example, before you can prefix a long text string to Name, you must change the width and format of Name from 35 to 60. The following statements modify and update the Name column:

```sql
proc sql;
  title "World's Largest Countries";
  alter table sql.newcountries
    modify name char(60) format=$60.;
  update sql.newcountries
    set name='The United Nations member country is '||name;

  select name from sql.newcountries;
```

Output 4.11  Changing a Column's Width

Deleting a Column

The DROP clause deletes columns from tables. The following DROP clause deletes UNDate from NewCountries:

```sql
DROP column UNDate from NewCountries;
```
Creating an Index

An index is a file that is associated with a table. The index enables access to rows by index value. Indexes can provide quick access to small subsets of data, and they can enhance table joins. You can create indexes, but you cannot instruct PROC SQL to use an index. PROC SQL determines whether it is efficient to use the index. Some columns might not be appropriate for an index. In general, create indexes for columns that have many unique values or are columns that you use regularly in joins.

Using PROC SQL to Create Indexes

You can create a simple index, which applies to one column only. The name of a simple index must be the same as the name of the column that it indexes. Specify the column name in parentheses after the table name. The following CREATE INDEX statement creates an index for the Area column in NewCountries:

```sql
proc sql;
create index area
  on sql.newcountries(area);
```

You can also create a composite index, which applies to two or more columns. The following CREATE INDEX statement creates the index Places for the Name and Continent columns in NewCountries:

```sql
proc sql;
create index places
  on sql.newcountries(name, continent);
```

To ensure that each value of the indexed column (or each combination of values of the columns in a composite index) is unique, use the UNIQUE keyword:

```sql
proc sql;
create unique index places
  on sql.newcountries(name, continent);
```

Using the UNIQUE keyword causes SAS to reject any change to a table that would cause more than one row to have the same index value.

Tips for Creating Indexes

- The name of the composite index cannot be the same as the name of one of the columns in the table.
- If you use two columns to access data regularly, such as a first name column and a last name column from an employee database, then you should create a composite index for the columns.
- Keep the number of indexes to a minimum to reduce disk space and update costs.
- Use indexes for queries that retrieve a relatively small number of rows (less than 15%).
- In general, indexing a small table does not result in a performance gain.
In general, indexing on a column with a small number (less than 6 or 7) of distinct values does not result in a performance gain.

You can use the same column in a simple index and in a composite index. However, for tables that have a primary key integrity constraint, do not create more than one index that is based on the same column as the primary key.

**Deleting Indexes**

To delete an index from a table, use the DROP INDEX statement. The following DROP INDEX statement deletes the index Places from NewCountries:

```sql
proc sql;
    drop index places from sql.newcountries;
```

**Deleting a Table**

To delete a PROC SQL table, use the DROP TABLE statement:

```sql
proc sql;
    drop table sql.newcountries;
```

**Using SQL Procedure Tables in SAS Software**

Because PROC SQL tables are SAS data files, you can use them as input to a DATA step or to other SAS procedures. For example, the following PROC MEANS step calculates the mean for Area for all countries that are in the Countries table:

```sql
proc means data=sql.countries mean maxdec=2;
    title "Mean Area for All Countries";
    var area;
run;
```

**Output 4.12 Using a PROC SQL Table in PROC MEANS**

Mean Area for All Countries

The MEANS Procedure

<table>
<thead>
<tr>
<th>Analysis Variable : Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
</tr>
<tr>
<td>250998.76</td>
</tr>
</tbody>
</table>
Creating and Using Integrity Constraints in a Table

Integrity constraints are rules that you specify to guarantee the accuracy, completeness, or consistency of data in tables. All integrity constraints are enforced when you insert, delete, or alter data values in the columns of a table for which integrity constraints have been defined. Before a constraint is added to a table that contains existing data, all the data is checked to determine that it satisfies the constraints.

You can use general integrity constraints to verify that data in a column is one of the following:

- nonmissing
- unique
- both nonmissing and unique
- within a specified set or range of values

You can also apply referential integrity constraints to link the values in a specified column (called a primary key) of one table to values of a specified column in another table. When linked to a primary key, a column in the second table is called a foreign key.

When you define referential constraints, you can also choose what action occurs when a value in the primary key is updated or deleted.

- You can prevent the primary key value from being updated or deleted when matching values exist in the foreign key. This is the default.
- You can allow updates and deletions to the primary key values. By default, any affected foreign key values are changed to missing values. However, you can specify the CASCADE option to update foreign key values instead. Currently, the CASCADE option does not apply to deletions.

You can choose separate actions for updates and for deletions.

Note: Integrity constraints cannot be defined for views.

The following example creates integrity constraints for a table, MyStates, and another table, USPostal. The constraints are as follows:

- state name must be unique and nonmissing in both tables
- population must be greater than 0
- continent must be either North America or Oceania

```sql
proc sql;
  create table sql.mystates
  (state      char(15),
   population num,
   continent  char(15),
   /* contraint specifications */
   constraint prim_key    primary key(state),
   constraint population  check(population gt 0),
   constraint continent   check(continent in ('North America', 'Oceania')));
```
create table sql.uspostal
(name      char(15),
code      char(2) not null,          /* constraint specified as    */          /* a column attribute */
constraint for_key foreign key(name) /* links NAME to the */
references sql.mystates /* primary key in MYSTATES */
on delete restrict /* forbids deletions to STATE */
/* unless there is no */
/* matching NAME value */
on update set null); /* allows updates to STATE, */
/* changes matching NAME */
/* values to missing */

The DESCRIBE TABLE CONSTRAINTS statement shows only the table constraint specifications in the Results window.

```
proc sql;
describe table sql.mystates;
describe table constraints sql.uspostal;
```

**Output 4.13**  **PROC SQL DESCRIBE TABLE CONSTRAINTS Results Window Shows Integrity Constraints**

<table>
<thead>
<tr>
<th>#</th>
<th>Integrity Constraint</th>
<th>Type</th>
<th>Variables</th>
<th>Where Clause</th>
<th>Reference</th>
<th>On Delete</th>
<th>On Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>continent</td>
<td>Check</td>
<td>continent</td>
<td>continent in ('North America', 'Oceania')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>population</td>
<td>Check</td>
<td>population</td>
<td>population&gt;0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>prim_key</td>
<td>Primary Key</td>
<td>state</td>
<td></td>
<td>SQL.USPOSTAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>for_key</td>
<td>Referential</td>
<td></td>
<td></td>
<td></td>
<td>Restrict</td>
<td>Set Null</td>
</tr>
</tbody>
</table>

Integrity constraints cannot be used in views. For more information about integrity constraints, see *SAS Language Reference: Concepts*. 
Overview of Creating and Using PROC SQL Views

A PROC SQL view contains a stored query that is executed when you use the view in a SAS procedure or DATA step. Views are useful for the following reasons:

- often save space, because a view is frequently quite small compared with the data that it accesses
- prevent users from continually submitting queries to omit unwanted columns or row
- shield sensitive or confidential columns from users while enabling the same users to view other columns in the same table
- ensure that input data sets are always current, because data is derived from tables at execution time
- hide complex joins or queries from users

Note: You can create a view that has two columns with the same variable name. However, if duplicate variable names exist in a view, PROC COMPARE cannot determine which column in the base data set should be compared to the column in the compare data set. As a result, PROC COMPARE issues an error if it finds duplicate variable names.

Creating Views

To create a PROC SQL view, use the CREATE VIEW statement, as shown in the following example:

```
libname sql 'SAS-library';

proc sql;
    title 'Current Population Information for Continents';
    create view sql.newcontinents as
        select continent,
            sum(population) as totpop  format=comma15. label='Total Population',
            sum(area) as totarea format=comma15. label='Total Area'
        from sql.countries
        group by continent;

    select * from sql.newcontinents;
```
Output 4.14  An SQL Procedure View

**Current Population Information for Continents**

<table>
<thead>
<tr>
<th>Continent</th>
<th>Total Population</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>384,772</td>
<td>876,800</td>
</tr>
<tr>
<td>Africa</td>
<td>710,529,592</td>
<td>11,299,595</td>
</tr>
<tr>
<td>Asia</td>
<td>3,381,858,879</td>
<td>12,198,325</td>
</tr>
<tr>
<td>Australia</td>
<td>18,255,944</td>
<td>2,966,200</td>
</tr>
<tr>
<td>Central America and Caribbean</td>
<td>66,815,930</td>
<td>291,463</td>
</tr>
<tr>
<td>Europe</td>
<td>813,481,724</td>
<td>9,167,250</td>
</tr>
<tr>
<td>North America</td>
<td>384,801,818</td>
<td>8,393,092</td>
</tr>
<tr>
<td>Oceania</td>
<td>5,342,368</td>
<td>129,600</td>
</tr>
<tr>
<td>South America</td>
<td>317,568,801</td>
<td>6,885,418</td>
</tr>
</tbody>
</table>

**Note:** In this example, each column has a name. If you are planning to use a view in a procedure that requires variable names, then you must supply column aliases that you can reference as variable names in other procedures. For more information, see “Using SQL Procedure Views in SAS Software” on page 136.

### Describing a View

The DESCRIBE VIEW statement writes a description of the PROC SQL view to the SAS log. The following SAS log describes the view NewContinents, which is created in “Creating Views” on page 131:

```sas
proc sql;
    describe view sql.newcontinents;
```

**Log 4.5  SAS Log from DESCRIBE VIEW Statement**

```
NOTE: SQL view SQL.NEWCONTINENTS is defined as:

    select continent, SUM(population) as totpop label='Total Population'
    format=COMMA15.0, SUM(area) as toatarea label='Total Area' format=COMMA15.0
    from SQL.COUNTRIES
    group by continent;
```

To define a password-protected SAS view, you must specify a password. If the SAS view was created with more than one password, you must specify its most restrictive password if you want to access a definition of the view. For more information, see “DESCRIBE Statement” on page 247.

### Updating a View

You can update data through a PROC SQL and SAS/ACCESS view with the INSERT, DELETE, and UPDATE statements, under the following conditions.
• You can update only a single table through a view. The underlying table cannot be
joined to another table or linked to another table with a set operator. The view cannot
contain a subquery.

• If the view accesses a DBMS table, then you must have been granted the appropriate
authorization by the external database management system (for example, ORACLE).
You must have installed the SAS/ACCESS software for your DBMS. For more
information about SAS/ACCESS views, see the SAS/ACCESS documentation for
your DBMS.

• You can update a column in a view by using the column's alias, but you cannot
update a derived column, that is, a column that is produced by an expression. In the
following example, you can update SquareMiles, but not Density:

```
proc sql;
create view mycountries as
  select Name,
       area as SquareMiles,
       population/area as Density
  from sql.countries;
```

• You can update a view that contains a WHERE clause. The WHERE clause can be in
the UPDATE clause or in the view. You cannot update a view that contains any other
clause, such as ORDER BY, HAVING, and so on.

**Embedding a LIBNAME in a View**

You can embed a SAS LIBNAME statement or a SAS/ACCESS LIBNAME statement in
a view by using the USING LIBNAME clause. When PROC SQL executes the view, the
stored query assigns the libref. For SAS/ACCESS librefs, PROC SQL establishes a
connection to a DBMS. The scope of the libref is local to the view and does not conflict
with any identically named librefs in the SAS session. When the query finishes, the
libref is disassociated. The connection to the DBMS is terminated and all data in the
library becomes unavailable.

The advantage of embedded librefs is that you can store engine-host options and DBMS
connection information, such as passwords, in the view. That, in turn, means that you do
not have to remember and reenter that information when you want to use the libref.

**Note:** The USING LIBNAME clause must be the last clause in the SELECT statement.
Multiple clauses can be specified, separated by commas.

In the following example, the libref OilInfo is assigned and a connection is made to an
ORACLE database:

```
proc sql;
create view sql.view1 as
  select *
  from oilinfo.reserves as newreserves
  using libname oilinfo oracle
     user=\username
     pass=\password
     path='\dbms-path';
```

For more information about the SAS/ACCESS LIBNAME statement, see the
SAS/ACCESS documentation for your DBMS.

The following example embeds a SAS LIBNAME statement in a view:
proc sql;
create view sql.view2 as
  select *
  from oil.reserves
  using libname oil 'SAS-library';

Deleting a View

To delete a view, use the DROP VIEW statement:

proc sql;
drop view sql.newcontinents;

Specifying In-Line Views

In some cases, you might want to use a query in a FROM clause instead of a table or view. You could create a view and refer to it in your FROM clause, but that process involves two steps. To save the extra step, specify the view in-line, enclosed in parentheses, in the FROM clause.

An in-line view is a query that appears in the FROM clause. An in-line view produces a table internally that the outer query uses to select data. Unlike views that are created with the CREATE VIEW statement, in-line views are not assigned names and cannot be referenced in other queries or SAS procedures as if they were tables. An in-line view can be referenced only in the query in which it is defined.

In the following query, the populations of all Caribbean and Central American countries are summed in an in-line query. The WHERE clause compares the sum with the populations of individual countries. Only countries that have a population greater than the sum of Caribbean and Central American populations are displayed.

libname sql 'SAS-library';

proc sql;
  title 'Countries With Population GT Caribbean Countries';
  select w.Name, w.Population format=comma15., c.TotCarib
  from (select sum(population) as TotCarib format=comma15.
           from sql.countries
           where continent = 'Central America and Caribbean') as c,
           sql.countries as w
  where w.population gt c.TotCarib;
Using an In-Line View

**Output 4.15  Using an In-Line View**

<table>
<thead>
<tr>
<th>Countries With Population GT Caribbean Countries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>Bangladesh</td>
</tr>
<tr>
<td>Brazil</td>
</tr>
<tr>
<td>China</td>
</tr>
<tr>
<td>Germany</td>
</tr>
<tr>
<td>India</td>
</tr>
<tr>
<td>Indonesia</td>
</tr>
<tr>
<td>Japan</td>
</tr>
<tr>
<td>Mexico</td>
</tr>
<tr>
<td>Nigeria</td>
</tr>
<tr>
<td>Pakistan</td>
</tr>
<tr>
<td>Philippines</td>
</tr>
<tr>
<td>Russia</td>
</tr>
<tr>
<td>United States</td>
</tr>
<tr>
<td>Vietnam</td>
</tr>
</tbody>
</table>

**Tips for Using SQL Procedure Views**

- Avoid using an ORDER BY clause in a view. If you specify an ORDER BY clause, then the data must be sorted each time that the view is referenced.

- If data is used many times in one program or in multiple programs, then it is more efficient to create a table rather than a view. If a view is referenced often in one program, then the data must be accessed at each reference.

- If the view resides in the same SAS library as the contributing table or tables, then specify a one-level name in the FROM clause. The default for the libref for the FROM clause's table or tables is the libref of the library that contains the view. This prevents you from having to change the view if you assign a different libref to the SAS library that contains the view and its contributing table or tables. This tip is used in the view that is described in “Creating Views” on page 131.

- Avoid creating views that are based on tables whose structures might change. A view is no longer valid when it references a nonexistent column.

- To limit the number of observations in a view, use the FIRSTOBS= or OBS= data set option, not the FIRSTOBS= or OBS= system option. When you use a system option, the observation limit is applied first to the underlying table, and then next to the view, effectively reducing the number of observations twice. When you use a data set option, the observation limit is applied to the view only.

- When you process PROC SQL views between a client and a server, getting the correct results depends on the compatibility between the client and server.
architecture. For more information, see “Access a SAS View” in SAS/CONNECT User’s Guide.

**Using SQL Procedure Views in SAS Software**

You can use PROC SQL views as input to a DATA step or to other SAS procedures. The syntax for using a PROC SQL view in SAS is the same as that for a PROC SQL table. For an example, see “Using SQL Procedure Tables in SAS Software” on page 128.
Chapter 5
Programming with the SQL Procedure

Introduction .................................................. 138
Using PROC SQL Options to Create and Debug Queries .................. 138
  Overview of Using PROC SQL Options to Create and Debug Queries .... 138
  Restricting Row Processing with the INOBS= and OUTOBS= Options .... 139
  Limiting Iterations with the LOOP= Option ................................. 139
  Checking Syntax with the NOEXEC Option and the VALIDATE Statement . 139
  Expanding SELECT * with the FEEDBACK Option ......................... 140
  Timing PROC SQL with the STIMER Option .................................. 140
  Resetting PROC SQL Options with the RESET Statement .................. 141

Improving Query Performance .................................. 142
  Overview of Improving Query Performance ................................. 142
  Using Indexes to Improve Performance ..................................... 142
  Using the Keyword ALL in Set Operations .................................. 143
  Omitting the ORDER BY Clause When Creating Tables and Views ........ 143
  Using In-Line Views versus Temporary Tables .............................. 143
  Comparing Subqueries with Joins .......................................... 143
  Using WHERE Expressions with Joins ....................................... 143
  Optimizing the PUT Function ................................................. 144
  Replacing References to the DATE, TIME, DATETIME, and TODAY Functions ........................................ 146
  Disabling the Remerging of Data When Using Summary Functions .... 146

Using Column Aliases ............................................. 146
  Overview of Column Aliases ............................................... 146
  Column Alias Extensions ..................................................... 147
  Using the CALCULATED Keyword with Column Aliases .................. 148

Accessing SAS System Information By Using DICTIONARY Tables ........ 149
  What Are DICTIONARY Tables? .............................................. 149
  Retrieving Information about DICTIONARY Tables and Sashelp Views ... 151
  Using DICTIONARY.Tables .................................................... 153
  Using DICTIONARY.Columns ................................................. 154
  DICTIONARY Tables and Performance ..................................... 155

Using SAS Data Set Options with PROC SQL ......................... 156

Using PROC SQL with the SAS Macro Facility ......................... 157
  Overview of Using PROC SQL with the SAS Macro Facility ............. 157
  Creating Macro Variables in PROC SQL .................................... 157
  Concatenating Values in Macro Variables .................................. 160
  Defining Macros to Create Tables ........................................... 161
  Using the PROC SQL Automatic Macro Variables ......................... 162
Introduction

This section shows you how to do the following:

• use PROC SQL options to create and debug queries
• improve query performance
• access DICTIONARY tables and how they are useful in gathering information about the elements of SAS
• use PROC SQL with the SAS macro facility
• use PROC SQL with the REPORT procedure
• access DBMSs by using SAS/ACCESS software
• format PROC SQL output by using the SAS Output Delivery System (ODS)

Using PROC SQL Options to Create and Debug Queries

Overview of Using PROC SQL Options to Create and Debug Queries

PROC SQL supports options that can give you greater control over PROC SQL while you are developing a query:

• The INOBS=, OUTOBS=, and LOOPS= options reduce query execution time by limiting the number of rows and the number of iterations that PROC SQL processes.
• The EXEC and VALIDATE statements enable you to quickly check the syntax of a query.
• The FEEDBACK option expands a SELECT * statement into a list of columns that the statement represents.
• The PROC SQL STIMER option records and displays query execution time.

You can set an option initially in the PROC SQL statement, and then use the RESET statement to change the same option's setting without ending the current PROC SQL step.
Restricting Row Processing with the INOBS= and OUTOBS= Options

When you are developing queries against large tables, you can reduce the time that it takes for the queries to run by reducing the number of rows that PROC SQL processes. Subsetting the tables with WHERE statements is one way to do this. Using the INOBS= option and the OUTOBS= option are other ways.

The INOBS= option restricts the number of rows that PROC SQL takes as input from any single source. For example, if you specify INOBS=10, then PROC SQL uses only 10 rows from any table or view that is specified in a FROM clause. If you specify INOBS=10 and join two tables without using a WHERE clause, then the resulting table (Cartesian product) contains a maximum of 100 rows. The INOBS= option is similar to the SAS system option OBS=.

The OUTOBS= option restricts the number of rows that PROC SQL displays or writes to a table. For example, if you specify OUTOBS=10 and insert values into a table by using a query, then PROC SQL inserts a maximum of 10 rows into the resulting table. OUTOBS= is similar to the SAS data set option OBS=.

In a simple query, there might be no apparent difference between using INOBS or OUTOBS. However, at other times it is important to choose the correct option. For example, taking the average of a column with INOBS=10 returns an average of only 10 values from that column.

Limiting Iterations with the LOOPS= Option

The LOOPS= option restricts PROC SQL to the number of iterations that are specified in this option through its inner loop. By setting a limit, you can prevent queries from consuming excessive computer resources. For example, joining three large tables without meeting the join-matching conditions could create a huge internal table that would be inefficient to process. Use the LOOPS= option to prevent this from happening.

You can use the number of iterations that are reported in the SQLOOPS macro variable (after each PROC SQL statement is executed) to gauge an appropriate value for the LOOPS= option. For more information, see “Using the PROC SQL Automatic Macro Variables” on page 162.

If you use the PROMPT option with the INOBS=, OUTOBS=, or LOOPS= options, you are prompted to stop or continue processing when the limits set by these options are reached.

Checking Syntax with the NOEXEC Option and the VALIDATE Statement

To check the syntax of a PROC SQL step without actually executing it, use the NOEXEC option or the VALIDATE statement. The NOEXEC option can be used once in the PROC SQL statement, and the syntax of all queries in that PROC SQL step will be checked for accuracy without executing them. The VALIDATE statement must be specified before each SELECT statement in order for that statement to be checked for accuracy without executing. If the syntax is valid, then a message is written to the SAS log to that effect. If the syntax is invalid, then an error message is displayed. The automatic macro variable SQLRC contains an error code that indicates the validity of the syntax. For an example of the VALIDATE statement used in PROC SQL, see “Validating a Query” on page 71. For an example of using the VALIDATE statement in a
Expanding SELECT * with the FEEDBACK Option

The FEEDBACK option expands a SELECT * (ALL) statement into the list of columns that the statement represents. Any PROC SQL view is expanded into the underlying query, all expressions are enclosed in parentheses to indicate their order of evaluation, and the PUT function optimizations that are performed on the query are displayed. The FEEDBACK option also displays the resolved values of macros and macro variables.

For example, the following query is expanded in the SAS log:

```
libname sql 'SAS-library';

proc sql feedback;
  select * from sql.countries;
```

Log 5.1 Expanded SELECT * Statement

```
NOTE: Statement transforms to:

  select COUNTRIES.Name, COUNTRIES.Capital, COUNTRIES.Population,
  COUNTRIES.Area, COUNTRIES.Continent, COUNTRIES.UNDate
  from SQL.COUNTRIES;
```

Timing PROC SQL with the STIMER Option

Certain operations can be accomplished in more than one way. For example, there is often a join equivalent to a subquery. Consider factors such as readability and maintenance, but generally you will choose the query that runs fastest. The SAS system option STIMER shows you the cumulative time for an entire procedure. The PROC SQL STIMER option shows you how fast the individual statements in a PROC SQL step are running. This enables you to optimize your query.

Note: For the PROC SQL STIMER option to work, the SAS system option STIMER must also be specified.

This example compares the execution times of two queries. Both queries list the names and populations of states in the UnitedStates table that have a larger population than Belgium. The first query does this with a join; the second with a subquery. Log 5.2 on page 141 shows the STIMER results from the SAS log.

```
libname sql 'SAS-library';

proc sql stimer ;
  select us.name, us.population
  from sql.unitedstates as us, sql.countries as w
  where us.population gt w.population and
    w.name = 'Belgium';

  select Name, population
  from sql.unitedstates
```
where population gt
   (select population from sql.countries
       where name = 'Belgium');

Log 5.2  Comparing Run Times of Two Queries

```
4  proc sql stimer ;
NOTE: SQL Statement used:
   real time           0.00 seconds
   cpu time            0.01 seconds
5   select us.name, us.population
6       from sql.unitedstates as us, sql.countries as w
7       where us.population gt w.population and
8           w.name = 'Belgium';
NOTE: The execution of this query involves performing one or more Cartesian
    product joins that can not be optimized.
NOTE: SQL Statement used:
   real time           0.10 seconds
   cpu time            0.05 seconds
9
10  select Name, population
11      from sql.unitedstates
12      where population gt
13              (select population from sql.countries
14                  where name = 'Belgium');
NOTE: SQL Statement used:
   real time           0.09 seconds
   cpu time            0.09 seconds
```

Compare the CPU time of the first query (that uses a join), 0.05 seconds, with 0.09
seconds for the second query (that uses a subquery). Although there are many factors
that influence the run times of queries, generally a join runs faster than an equivalent
subquery.

**Resetting PROC SQL Options with the RESET Statement**

Use the RESET statement to add, drop, or change the options in the PROC SQL
statement. You can list the options in any order in the PROC SQL and RESET
statements. Options stay in effect until they are reset.

This example first uses the NOPRINT option to prevent the SELECT statement from
displaying its result table in SAS output. The RESET statement then changes the
NOPRINT option to PRINT (the default) and adds the NUMBER option, which displays
the row number in the result table.

```
proc sql noprint;
   title 'Countries with Population Under 20,000';
   select Name, Population from sql.countries;
reset print number;
   select Name, Population from sql.countries
       where population lt 20000;
```
Improving Query Performance

Overview of Improving Query Performance

There are several ways to improve query performance, including the following:

• using indexes and composite indexes
• using the keyword ALL in set operations when you know that there are no duplicate rows, or when it does not matter if you have duplicate rows in the result table
• omitting the ORDER BY clause when you create tables and views
• using in-line views instead of temporary tables (or vice versa)
• using joins instead of subqueries
• using WHERE expressions to limit the size of result tables that are created with joins
• using either PROC SQL options, SAS system options, or both to replace a PUT function in a query with a logically equivalent expression
• replacing references to the DATE, TIME, DATETIME, and TODAY functions in a query with their equivalent constant values before the query executes
• disabling the remerging of data when summary functions are used in a query

Using Indexes to Improve Performance

Indexes are created with the CREATE INDEX statement in PROC SQL or with the MODIFY and INDEX CREATE statements in the DATASETS procedure. Indexes are stored in specialized members of a SAS library and have a SAS member type of INDEX. The values that are stored in an index are automatically updated if you make a change to the underlying data.

Indexes can improve the performance of certain classes of retrievals. For example, if an indexed column is compared to a constant value in a WHERE expression, then the index will likely improve the query's performance. Indexing the column that is specified in a correlated reference to an outer table also improves a subquery's (and hence, query's) performance. Composite indexes can improve the performance of queries that compare
the columns that are named in the composite index with constant values that are linked using the AND operator. For example, if you have a compound index in the columns CITY and STATE, and the WHERE expression is specified as WHERE CITY='xxx' AND STATE='yy', then the index can be used to select that subset of rows more efficiently. Indexes can also benefit queries that have a WHERE clause in this form:

\[ \ldots \text{where var1 in (select item1 from table1)} \ldots \]

The values of VAR1 from the outer query are found in the inner query by using the index. An index can improve the processing of a table join, if the columns that participate in the join are indexed in one of the tables. This optimization can be done for equijoin queries only—that is, when the WHERE expression specifies that table1.X=table2.Y.

**Using the Keyword ALL in Set Operations**

Set operators such as UNION, OUTER UNION, EXCEPT, and INTERSECT can be used to combine queries. Specifying the optional ALL keyword prevents the final process that eliminates duplicate rows from the result table. You should use the ALL form when you know that there are no duplicate rows or when it does not matter whether the duplicate rows remain in the result table.

**Omitting the ORDER BY Clause When Creating Tables and Views**

If you specify the ORDER BY clause when a table or view is created, then the data is always displayed in that order unless you specify another ORDER BY clause in a query that references that table or view. As with any sorting procedure, using ORDER BY when retrieving data has certain performance costs, especially on large tables. If the order of your output is not important for your results, then your queries will typically run faster without an ORDER BY clause.

**Using In-Line Views versus Temporary Tables**

It is often helpful when you are exploring a problem to break a query down into several steps and create temporary tables to hold the intermediate results. After you have worked through the problem, combining the queries into one query by using in-line views can be more efficient. However, under certain circumstances it is more efficient to use temporary tables. You should try both methods to determine which is more efficient for your case.

**Comparing Subqueries with Joins**

Many subqueries can also be expressed as joins. Generally, a join is processed at least as efficiently as the subquery. PROC SQL stores the result values for each unique set of correlation columns temporarily, thereby eliminating the need to calculate the subquery more than once.

**Using WHERE Expressions with Joins**

When joining tables, you should specify a WHERE expression. Joins without WHERE expressions are often time-consuming to evaluate because of the multiplier effect of the Cartesian product. For example, joining two tables of 1,000 rows each without specifying a WHERE expression or an ON clause, produces a result table with one million rows.
PROC SQL executes and obtains the correct results in unbalanced WHERE expressions (or ON join expressions) in an equijoin, as shown here, but handles them inefficiently:

    where table1.columnA-table2.columnB=0

It is more efficient to rewrite this clause to balance the expression so that columns from each table are on alternate sides of the equals condition:

    where table1.columnA=table2.columnB

PROC SQL sequentially processes joins that do not have an equijoin condition evaluating each row against the WHERE expression: that is, joins without an equijoin condition are not evaluated using sort-merge or index-lookup techniques. Evaluating left and right outer joins is generally comparable to, or only slightly slower than, a standard inner join. A full outer join usually requires two passes over both tables in the join, although PROC SQL tries to store as much data as possible in buffers. Thus, for small tables, an outer join might be processed with only one physical read of the data.

Optimizing the PUT Function

Reducing the PUT Function

There are several ways that you can improve the performance of a query by optimizing the PUT function. If you reference tables in a database, eliminating references to PUT functions can enable more of the query to be passed to the database. It can simplify SELECT statement evaluation for the default Base SAS engine.

There are five possible evaluations that are performed when optimizing the PUT function:

- Functions, including PUT, that contain literal values.
- PUT functions in the WHERE and HAVING clauses that contain formats that are supplied by SAS.
- PUT functions in the WHERE and HAVING clauses that contain user-defined formats.
- PUT functions in any part of the SELECT statement that contain user-defined formats that are defined with an OTHER= clause.
- PUT functions that are deployed inside the database.

Controlling PUT Function Optimization

- If you specify either the PROC SQL REDUCEPUT= option or the SQLREDUCEPUT= system option, SAS optimizes the PUT function before the query is executed.

    The following SELECT statements are examples of queries that would be optimized:

    select x, y from sqllibb where (PUT(x, abc.) in ('yes', 'no'));
    select x from sqlliba where (PUT(x, udfmt.) = trim(left('small')));

- For databases that allow implicit pass-through when the row count for a table is not known, PROC SQL allows the optimization in order for the query to be executed by the database. When the PROC SQL REDUCEPUT= option or the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, PROC SQL considers the value of the PROC SQL REDUCEPUTOBS= option or the SQLREDUCEPUTOBS= system option and determines whether to optimize the PUT function. The PROC SQL REDUCEPUTOBS= option or the SQLREDUCEPUTOBS= system option specifies the minimum number of rows that...
must be in a table in order for PROC SQL to consider optimizing the PUT function in a query. For databases that do not allow implicit pass-through, PROC SQL does not perform the optimization, and more of the query is performed by SAS.

• Some formats, especially user-defined formats, can contain many format values. Depending on the number of matches for a given PUT function expression, the resulting expression can list many format values. If the number of format values becomes too large, the query performance can degrade. When the PROC SQL REDUCEPUT= option or the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, PROC SQL considers the value of the PROC SQL REDUCEPUTVALUES= option or the SQLREDUCEPUTVALUES= system option and determines whether to optimize the PUT function in a query. For databases that do not allow implicit pass-through, PROC SQL does not perform the optimization, and more of the query is performed by SAS.

For more information, see the REDUCEPUT=, REDUCEPUTOBS=, and REDUCEPUTVALUES= options in Chapter 7, “SQL Procedure,” on page 215, and the SQLREDUCEPUT=, SQLREDUCEPUTOBS=, and SQLREDUCEPUTVALUES= system options in Appendix 1, “SQL Macro Variables and System Options,” on page 373.

Note: PROC SQL can consider both the REDUCEPUTOBS= option and the REDUCEPUTVALUES= option (or the SQLREDUCEPUTOBS= system option and the SQLREDUCEPUTVALUES= system option) when trying to determine whether to optimize the PUT function.

Deploying the PUT Function and SAS Formats inside a DBMS
SAS/ACCESS software for relational databases enables you to use the format publishing macro to deploy or publish the PUT function implementation to the database as a function named SAS_PUT(). As with any other programming function, the SAS_PUT() function can take one or more input parameters and return an output value. The default value for the SQLMAPPUTTO system option is SAS_PUT. After the SAS_PUT() function is deployed in the database, you can use the SAS_PUT() function as you would use any standard SQL function inside the database.

In addition, the SAS_PUT() function supports the use of SAS formats in SQL queries that are submitted to the database. You can use the format publishing macro to publish to the database both the formats that are supplied by SAS and the custom formats that you create with the FORMAT procedure.

By publishing the PUT function implementation to the database as the SAS_PUT() function to support the use of SAS formats, and by packaging both the formats that are supplied by SAS and the custom formats that you create with the FORMAT procedure, the following advantages are realized:

• The entire SQL query can be processed inside the database.
• The SAS format processing leverages the DBMS’s scalable architecture.
• The results are grouped by the formatted data, and are extracted from the database.

Note: If you use the SQL_FUNCTIONS= LIBNAME statement option to remap the PUT function (for example, SAS_PUT( )), then the SQL_FUNCTIONS= LIBNAME option takes precedence over the SQLMAPPUTTO= system option. For more information, see “SQL_FUNCTIONS= LIBNAME Option” in SAS/ACCESS for Relational Databases: Reference.

Tip: Using both the SQLREDUCEPUT= system option (or the PROC SQL REDUCEPUT= option) and the SAS_PUT() function can result in a significant performance boost.
Replacing References to the DATE, TIME, DATETIME, and TODAY Functions

When the PROC SQL CONSTDATETIME option or the SQLCONSTDATETIME system option is set, PROC SQL evaluates the DATE, TIME, DATETIME, and TODAY functions in a query once, and uses those values throughout the query. Computing these values once ensures consistent results when the functions are used multiple times in a query, or when the query executes the functions close to a date or time boundary. When referencing database tables, performance is enhanced because it allows more of the query to be passed down to the database.

For more information, see the “SQLCONSTDATETIME System Option” on page 373 or the CONSTDATETIME option in the Base SAS Procedures Guide.

Note: If you specify both the PROC SQL REDUCEPUT option or the SQLREDUCEPUT= system option and the PROC SQL CONSTDATETIME option or the SQLCONSTDATETIME system option, PROC SQL replaces the DATE, TIME, DATETIME, and TODAY functions with their respective values in order to determine the PUT function value before the query executes.

Disabling the Remerging of Data When Using Summary Functions

When you use a summary function in a SELECT clause or a HAVING clause, PROC SQL might remerge the data. Remerging the data involves two passes through the data. If you set the PROC SQL NOREMERGE option or the NOSQLREMERGE system option, PROC SQL will not process the remerging of data. When referencing database tables, performance is enhanced because it enables more of the query to be passed down to the database.

For more information, see the PROC SQL statement REMERGE option in the Base SAS Procedures Guide and the SQLREMERGE system option in Appendix 1, “SQL Macro Variables and System Options,” on page 373.

Using Column Aliases

Overview of Column Aliases

A column alias is a temporary, alternate name for a column. Aliases are specified in the SELECT clause to name or rename columns so that the result table is clearer or easier to read. Aliases are often used to name a column that is the result of an arithmetic expression or summary function. An alias is one word only. If you need a longer column name, then use the LABEL= column-modifier, as described in “column-modifier” on page 321. The keyword AS is required with a column alias to distinguish the column alias from column names in the SELECT clause.

Column aliases are optional, and each column name in the SELECT clause can have an alias. After you assign an alias to a column, you can use the alias to refer to that column in other clauses.

If you use a column alias when creating a PROC SQL view, then the alias becomes the permanent name of the column for each execution of the view.
Note: For the maximum portability of SQL code that is used outside of the SAS SQL procedure, avoid writing code that refers to column aliases in a WHERE clause, GROUP BY clause, or HAVING clause.

**Column Alias Extensions**

The development scope of PROC SQL and its aliasing rules predate the scope and rules of the first ANSI SQL standard and the ISO SQL standard. In PROC SQL, a column alias can be used in a WHERE clause, ON clause, GROUP BY clause, HAVING clause, or ORDER BY clause. In the ANSI SQL standard and ISO SQL standard, the value that is associated with a column alias does not need to be available until the ORDER BY clause is executed. As a result, there is no guarantee that an SQL processor can resolve a column alias in time for it to be referenced in a WHERE clause, GROUP BY clause, or HAVING clause. Because the ANSI SQL standard and ISO SQL standard require that a column alias needs to be available only for reference when the ORDER BY clause is executed, avoid writing code that refers to a column alias in a WHERE clause, GROUP BY clause, or HAVING clause.

If you refer to a column alias in an SQL expression (other than as part of an SQL expression that occurs in an ORDER BY clause), then the alias might not work. If you refer to a column alias in a context other than in an ORDER BY clause, then you should preface each column alias reference with the CALCULATED keyword. For more information, see “Using the CALCULATED Keyword with Column Aliases” on page 148.

There are six parts in the conceptual order of execution of a SELECT statement from the ANSI SQL standard or ISO SQL standard perspective. If all six parts exist, the sequence is the following:

1. The FROM part is executed first.
2. The WHERE or ON part is executed second.
3. The GROUP BY part is executed third.
4. The HAVING part is executed fourth.
5. The SELECT part is executed fifth.
6. The ORDER BY part is executed last.

The only required parts of an SQL query are the SELECT clause and FROM clause. The other four parts might be optional, depending on what type of query you are performing.

Here is a high-level template of an SQL query. The number enclosed in parentheses to the right of each part represents its position in the conceptual order of execution.

```sql
select <SELECT list> (5)
from <FROM clause> (1)
where <WHERE clause> (2)
group by <GROUP BY clause> (3)
having <HAVING clause> (4)
order by <ORDER BY clause>; (6)
```

In the following code examples, the first alias in each SELECT statement is just a rename of a table column. The second alias refers to a calculated expression. The first and second SQL statements writes the expected results in PROC SQL.

Here is the preferred SQL code example because a column alias is not referenced in the WHERE clause. This code example is portable to other SQL processors.

```sql
select qty as Quantity, cost, cost+100 as ListPrice
```
from calc
where qty > 5;

This code example will work in PROC SQL, but it might not work with other SQL processors.

select qty as Quantity, cost, cost+100 as ListPrice
from calc
where Quantity > 5;

Using the CALCULATED Keyword with Column Aliases

An early extension to PROC SQL development was the CALCULATED keyword. The CALCULATED keyword enables PROC SQL users to reference column aliases that are associated with calculated expressions. The column alias referenced by the CALCULATED keyword can be in the WHERE clause, ON clause, GROUP BY clause, HAVING clause, or ORDER BY clause. Using the CALCULATED keyword can be redundant if it is used in the ORDER BY clause to refer to a column alias. That column alias is resolved before the ORDER BY clause is executed. The CALCULATED keyword cannot be used with the ON clause for outer joins and inner joins. For more information, see “Referring to a Calculated Column by Alias” on page 31.

Here is a PROC SQL code example that uses the CALCULATED keyword to subset the rows by the values that are associated with the second alias (ListPrice).

Example Code 1  CALCULATED Keyword and the PROC SQL Use of Column Aliases

/*-- PROC SQL use of the CALCULATED keyword --*/
select qty as Quantity, cost, cost+100 as ListPrice
from calc
where CALCULATED ListPrice > 1500;

Here is an ISO SQL standard-approved and ANSI SQL standard-approved way of accomplishing this task:

Example Code 2  CALCULATED Keyword and the PROC SQL Use of Column Aliases

/*-- PROC SQL use of the CALCULATED keyword --*/
select qty as Quantity, cost, cost+100 as ListPrice
from calc
where cost+100 > 1500;

The code in the previous example is portable.

See Also

• “Creating New Columns” on page 27
• “Assigning a Column Alias” on page 30
• “Referring to a Calculated Column by Alias” on page 31
• “CALCULATED” on page 317
• “SELECT Statement” on page 252
• “WHERE Clause” on page 262
• “GROUP BY Clause” on page 263
• “HAVING Clause” on page 264
• “ORDER BY Clause” on page 265
Accessing SAS System Information By Using DICTIONARY Tables

What Are DICTIONARY Tables?

DICTIONARY tables are special read-only PROC SQL tables or views. They retrieve information about all the SAS libraries, SAS data sets, SAS system options, and external files that are associated with the current SAS session. For example, the DICTIONARY.Columns table contains information such as name, type, length, and format, about all columns in all tables that are known to the current SAS session.

PROC SQL automatically assigns the DICTIONARY libref. To get information from DICTIONARY tables, specify DICTIONARY.table-name in the FROM clause in a SELECT statement in PROC SQL.

DICTIONARY.table-name is valid in PROC SQL only. However, SAS provides PROC SQL views, based on the DICTIONARY tables, that can be used in other SAS procedures and in the DATA step. These views are stored in the Sashelp library and are commonly called “Sashelp views.”

For an example of a DICTIONARY table, see “Example 7: Reporting from DICTIONARY Tables” on page 282.

The following table describes the DICTIONARY tables that are available and shows the associated Sashelp views for each table.

<table>
<thead>
<tr>
<th>DICTIONARY Table</th>
<th>Sashelp View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATALOGS</td>
<td>Vcatalg</td>
<td>Contains information about known SAS catalogs.</td>
</tr>
<tr>
<td>CHECK_CONSTRAINTS</td>
<td>Vchkcon</td>
<td>Contains information about known check constraints.</td>
</tr>
<tr>
<td>COLUMNS</td>
<td>Vcolumn</td>
<td>Contains information about columns in all known tables.</td>
</tr>
<tr>
<td>CONSTRAINT_COLUMN_USAGE</td>
<td>Vcnco1u</td>
<td>Contains information about columns that are referred to by integrity constraints.</td>
</tr>
<tr>
<td>CONSTRAINT_TABLE_USAGE</td>
<td>Vcntabu</td>
<td>Contains information about tables that have integrity constraints defined on them.</td>
</tr>
<tr>
<td>DATAITEMS</td>
<td>Vdatait</td>
<td>Contains information about known information map data items.</td>
</tr>
<tr>
<td>DESTINATIONS</td>
<td>Vdest</td>
<td>Contains information about known ODS destinations.</td>
</tr>
<tr>
<td>DICTIONARIES</td>
<td>Vdetnry</td>
<td>Contains information about all DICTIONARY tables.</td>
</tr>
<tr>
<td>ENGINES</td>
<td>Vengine</td>
<td>Contains information about SAS engines.</td>
</tr>
<tr>
<td>DICTIONARY Table</td>
<td>SasHelp View</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>EXTFILES</td>
<td>Vextfl</td>
<td>Contains information about known external files.</td>
</tr>
<tr>
<td>FILTERS</td>
<td>Vfilter</td>
<td>Contains information about known information map filters.</td>
</tr>
<tr>
<td>FORMATS</td>
<td>Vformat</td>
<td>Contains information about currently accessible formats and informat.</td>
</tr>
<tr>
<td></td>
<td>Vcformat</td>
<td></td>
</tr>
<tr>
<td>FUNCTIONS</td>
<td>Vfunc</td>
<td>Contains information about currently accessible functions.</td>
</tr>
<tr>
<td>GOPTIONS</td>
<td>Vgopt</td>
<td>Contains information about currently defined graphics options (SAS/GRAFH software). SasHelp.Vallopt includes SAS system options as well as graphics options.</td>
</tr>
<tr>
<td></td>
<td>Vallopt</td>
<td></td>
</tr>
<tr>
<td>INDEXES</td>
<td>Vindex</td>
<td>Contains information about known indexes.</td>
</tr>
<tr>
<td>INFOMAPS</td>
<td>Vinfomp</td>
<td>Contains information about known information maps.</td>
</tr>
<tr>
<td>LIBNAMES</td>
<td>Vlibnam</td>
<td>Contains information about currently defined SAS libraries.</td>
</tr>
<tr>
<td>MACROS</td>
<td>Vmacro</td>
<td>Contains information about currently defined macro variables.</td>
</tr>
<tr>
<td>MEMBERS</td>
<td>Vmember</td>
<td>Contains information about all objects that are in currently defined SAS libraries. SasHelp.Vmember contains information for all member types; the other SasHelp views are specific to particular member types (such as tables or views).</td>
</tr>
<tr>
<td></td>
<td>Vsacces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vscatlg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vslib</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vstable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vstabvw</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vsview</td>
<td></td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Voption</td>
<td>Contains information about SAS system options. SasHelp.Vallopt includes graphics options as well as SAS system options.</td>
</tr>
<tr>
<td></td>
<td>Vallopt</td>
<td></td>
</tr>
<tr>
<td>REFERENTIAL_CONSTRAINTS</td>
<td>Vrefcon</td>
<td>Contains information about referential constraints.</td>
</tr>
<tr>
<td>REMEMBER</td>
<td>Vrememb</td>
<td>Contains information about known remembers.</td>
</tr>
<tr>
<td>STYLES</td>
<td>Vstyle</td>
<td>Contains information about known ODS styles.</td>
</tr>
<tr>
<td>TABLE_CONSTRAINTS</td>
<td>Vtabcon</td>
<td>Contains information about integrity constraints in all known tables.</td>
</tr>
<tr>
<td>TABLES</td>
<td>Vtable</td>
<td>Contains information about known tables.</td>
</tr>
<tr>
<td>DICTIONARY Table</td>
<td>Sashelp View</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TITLES</td>
<td>Vtitle</td>
<td>Contains information about currently defined titles and footnotes.</td>
</tr>
<tr>
<td>VIEWS</td>
<td>Vview</td>
<td>Contains information about known data views.</td>
</tr>
<tr>
<td>VIEW_SOURCES</td>
<td>Not available</td>
<td>Contains a list of tables (or other views) referenced by the SQL or DATASTEP view, and a count of the number of references.</td>
</tr>
<tr>
<td>XATTRS</td>
<td>Vxattr</td>
<td>Contains information about extended attributes.</td>
</tr>
</tbody>
</table>

**Retrieving Information about DICTIONARY Tables and Sashelp Views**

To see how each DICTIONARY table is defined, submit a DESCRIBE TABLE statement. This example shows the definition of DICTIONARY.Tables:

```sql
proc sql;
    describe table dictionary.tables;
```

The results are written to the SAS log.
Log 5.3  Definition of DICTIONARY.Tables

```
create table DICTIONARY.TABLES
(
  libname char(8) label='Library Name',
  memname char(32) label='Member Name',
  memtype char(8) label='Member Type',
  dbms_memtype char(32) label='DBMS Member Type',
  memlabel char(256) label='Data Set Label',
  typemem char(8) label='Data Set Type',
  crdate num format=DATETIME informat=DATETIME label='Date Created',
  modate num format=DATETIME informat=DATETIME label='Date Modified',
  nobs num label='Number of Physical Observations',
  obslen num label='Observation Length',
  nvar num label='Number of Variables',
  protect char(3) label='Type of Password Protection',
  compress char(8) label='Compression Routine',
  encrypt char(8) label='Encryption',
  npage num label='Number of Pages',
  filesize num label='Size of File',
  pcompress num label='Percent Compression',
  reuse char(3) label='Reuse Space',
  bufsize num label='Bufsize',
  delobs num label='Number of Deleted Observations',
  nlob num label='Number of Logical Observations',
  maxvar num label='Longest variable name',
  maxlabel num label='Longest label',
  maxgen num label='Maximum number of generations',
  gen num label='Generation number',
  attr char(3) label='Data Set Attributes',
  indxtype char(9) label='Type of Indexes',
  datarep char(32) label='Data Representation',
  sortname char(8) label='Name of Collating Sequence',
  sorttype char(4) label='Sorting Type',
  sortchar char(8) label='Charset Sorted By',
  regvector char(24) format=$HEX48 informat=$HEX48 label='Requirements Vector',
  datarepname char(170) label='Data Representation Name',
  encoding char(256) label='Data Encoding',
  audit char(8) label='Audit Trail Active?',
  audit_before char(8) label='Audit Before Image?',
  audit_admin char(8) label='Audit Admin Image?',
  audit_error char(8) label='Audit Error Image?',
  audit_data char(8) label='Audit Data Image?',
  num_character num label='Number of Character Variables',
  num_numeric num label='Number of Numeric Variables'
);
```

Similarly, you can use the DESCRIBE VIEW statement in PROC SQL to determine how a Sashelp view is defined. Here is an example:

```
proc sql;
   describe view sashelp.vstabvw;
```

Log 5.4  Description of Sashelp.Vstabvw

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

   select libname, memname, memtype
   from DICTIONARY.MEMBERS
   where (memtype='VIEW') or (memtype='DATA')
   order by libname asc, memname asc;
```
**Using DICTIONARY.Tables**

DICTIONARY tables are commonly used to monitor and manage SAS sessions because the data is more easily manipulated than the output from other sources such as PROC DATASETS. You can query DICTIONARY tables the same way you query any other table, including subsetting with a WHERE clause, ordering the results, and creating PROC SQL views.

Note that many character values in the DICTIONARY tables are stored as all-uppercase characters; you should design your queries accordingly.

Because DICTIONARY tables are read-only objects, you cannot insert rows or columns, alter column attributes, or add integrity constraints to them.

*Note:* For DICTIONARY.Tables and Sashelp.Vtable, if a table is read-protected with a password, then the only information that is listed for that table is the library name, member name, member type, and type of password protection. All other information is set to missing.

*Note:* An error occurs if DICTIONARY.Tables is used to retrieve information about an SQL view that exists in one library but has an input table from a second library that has not been assigned.

The following query uses a SELECT and subsetting WHERE clause to retrieve information about permanent tables and views that appear in the SQL library:

```sas
libname sql 'SAS-library';

proc sql;
  title 'All Tables and Views in the SQL Library';
  select libname, memname, memtype, nobs
    from dictionary.tables
    where libname='SQL';
```

```sas```
Using DICTIONARY.Columns

DICTIONARY tables are useful when you want to find specific columns to include in reports. The following query shows which of the tables that are used in this document contain the Country column:

```
libname sql 'SAS-library';
```
### All Tables That Contain the Country Column

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Member Name</th>
<th>Column Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>OILPROD</td>
<td>Country</td>
</tr>
<tr>
<td>SQL</td>
<td>OILRSRVS</td>
<td>Country</td>
</tr>
<tr>
<td>SQL</td>
<td>WORLDCITYCOORDS</td>
<td>Country</td>
</tr>
<tr>
<td>SQL</td>
<td>WORLDTEMPS</td>
<td>Country</td>
</tr>
</tbody>
</table>

**DICTIONARY Tables and Performance**

When querying a DICTIONARY table, SAS launches a discovery process that gathers information that is pertinent to that table. Depending on the DICTIONARY table that is being queried, this discovery process can search libraries, open tables, and execute views. Unlike other SAS procedures and the DATA step, PROC SQL can mitigate this process by optimizing the query before the discovery process is launched. Therefore, although it is possible to access DICTIONARY table information with SAS procedures or the DATA step by using the Sashelp views, it is often more efficient to use PROC SQL instead.

*Note*: You cannot use data set options with DICTIONARY tables.

For example, the following programs produce the same result, but the PROC SQL step runs much faster because the WHERE clause is processed before the tables that are referenced by the Sashelp.Vcolumn view are opened:

```sas
data mytable;
  set sashelp.vcolumn;
  where libname='WORK' and memname='SALES';
run;
proc sql;
  create table mytable as
  select * from sashelp.vcolumn
  where libname='WORK' and memname='SALES';
quit;
```

*Note*: SAS does not maintain DICTIONARY table information between queries. Each query of a DICTIONARY table launches a new discovery process.

If you are querying the same DICTIONARY table several times in a row, then you can get even faster performance by creating a temporary SAS data set (with the DATA step SET statement or the PROC SQL CREATE TABLE AS statement) with the information that you want and running your query against that data set.
When you query DICTIONARY.Tables or Sashelp.Vtable, all of the tables and views in all libraries that are assigned to the SAS session are opened to retrieve the requested information.

You can use a WHERE clause to help restrict which libraries are searched. However, the WHERE clause does not process most function calls such as UPCASE.

For example, if `WHERE UPCASE(libname) = 'WORK'` is used, the UPCASE function prevents the WHERE clause from optimizing this condition. All libraries that are assigned to the SAS session are searched. Searching all of the libraries could cause an unexpected increase in search time, depending on the number of libraries that are assigned to the SAS session.

All librefs and SAS table names are stored in uppercase. If you supply values for LIBNAME and MEMNAME in uppercase and you remove the UPCASE function, the WHERE clause will be optimized and performance will be improved. In the previous example, the code would be changed to `WHERE libname='WORK'`.

**Note:** Searching all librefs might cause unexpected results. If all librefs are searched, a view might exist that contains a libref that is not currently assigned to the SAS session. When this view is opened to retrieve information for the query, an error occurs.

**Note:** If you query table information from a library that is assigned to an external database, and you use the LIBNAME statement PRESERVE_TAB_NAMES=YES option or the PRESERVE_COL_NAMES=YES option, and you provide the table or column name as it appears in the database, you do not need to use the UPCASE function.

---

### Using SAS Data Set Options with PROC SQL

In PROC SQL, you can apply most of the SAS data set options, such as KEEP= and DROP=, to tables or SAS/ACCESS views anytime you specify a table or SAS/ACCESS view. In the SQL procedure, SAS data set options that are separated by spaces are enclosed in parentheses. The data set options immediately follow the table or SAS/ACCESS view name. In the following PROC SQL step, the RENAME= data set option renames LNAME to LASTNAME for the Staff1 table. The OBS= data set option restricts the number of rows that are read from Staff1 to 15:

```sql
proc sql;
  create table staff1(rename=(lname=lastname)) as
    select *
    from staff(obs=15);
```

SAS data set options can be combined with SQL statement arguments. In the following PROC SQL step, the PW= data set option assigns a password to the Test table, and the ALTER= data set option assigns an ALTER password to the Staff1 table:

```sql
proc sql;
  create table test
    (a character, b numeric, pw=cat);
  create index staffidx on
    staff1(lastname, alter=dog);
```

In this PROC SQL step, the PW= data set option assigns a password to the ONE table. The password is used when inserting a row and updating the table.
Using PROC SQL with the SAS Macro Facility

Overview of Using PROC SQL with the SAS Macro Facility

The macro facility is a programming tool that you can use to extend and customize SAS software. The macro facility reduces the amount of text that you must enter to perform common or repeated tasks and improves the efficiency and usefulness of your SQL programs.

The macro facility enables you to assign a name to character strings or groups of SAS programming statements. Thereafter, you can work with the names rather than with the text itself. For more information about the SAS macro facility, see SAS Macro Language: Reference.

Macro variables provide an efficient way to replace text strings in SAS code. The macro variables that you create and name are called user-defined macro variables. The macros variables that are defined by SAS are called automatic macro variables. PROC SQL produces six automatic macro variables (SQLOBS, SQLRC, SQLLOOPS, SLEXITCODE, SQLXRC, and SQLXMSG) to help you troubleshoot your programs. For more information, see “Using the PROC SQL Automatic Macro Variables” on page 162.

Creating Macro Variables in PROC SQL

Overview of Creating Macro Variables in PROC SQL

Other software vendors' SQL products allow the embedding of SQL into another language. References to variables (columns) of that language are termed host-variable references. They are differentiated from references to columns in tables by names that are prefixed with a colon. The host-variable stores the values of the object-items that are listed in the SELECT clause.
The only host language that is currently available in SAS is the macro language, which is part of Base SAS software. When a calculation is performed on a column's value, its result can be stored, using a macro-variable, in the macro facility. The result can then be referenced by that name in another PROC SQL query or SAS procedure. Host-variable can be used only in the outer query of a SELECT statement, not in a subquery. Host-variable cannot be used in a CREATE statement.

If the query produces more than one row of output, then the macro variable will contain only the value from the first row. If the query has no rows in its output, then the macro variable is not modified. If the macro variable does not exist yet, it will not be created. The PROC SQL macro variable SQLOBS contains the number of rows that are produced by the query.

Note: The SQLOBS automatic macro variable is assigned a value after the SQL SELECT statement executes.

Creating Macro Variables from the First Row of a Query Result
If you specify a single macro variable in the INTO clause, then PROC SQL assigns the variable the value from the first row only of the appropriate column in the SELECT list. In this example, &country1 is assigned the value from the first row of the Country column, and &barrels1 is assigned the value from the first row of the Barrels column. The NOPRINT option prevents PROC SQL from displaying the results of the query. The %PUT statement writes the contents of the macro variables to the SAS log.

```
libname sql 'SAS-library';

proc sql noprint;
  select country, barrels
  into :country1, :barrels1
  from sql.oilrsrvs;
%put &country1 &barrels1;
```

Log 5.5  Creating Macro Variables from the First Row of a Query Result

```
4  proc sql noprint;
5   select country, barrels
6      into :country1, :barrels1
7      from sql.oilrsrvs;
8
9  %put &country1 &barrels1;
Algeria                            9,200,000,000
NOTE: PROCEDURE SQL used:
   real time           0.12 seconds
```

Creating a Macro Variable from the Result of an Aggregate Function
A useful feature of macro variables is that they enable you to display data values in SAS titles. The following example prints a subset of the Worldtemps table and lists the highest temperature in Canada in the title:

```
libname sql 'SAS-library';

proc sql outobs=12;
  reset noprint;
  select max(AvgHigh)
  into :maxtemp
  from sql.worldtemps
  where country = 'Canada';
```
reset print;
    title "The Highest Temperature in Canada: &maxtemp";
select city, AvgHigh format 4.1
    from sql.worldtemps
    where country = 'Canada';

Note: You must use double quotation marks in the TITLE statement to resolve the reference to the macro variable.

Note: By default, macro variables that contain large numeric values are formatted using the BEST8. format. This format can cause the value to be displayed using scientific notation. You can use another format, such as the w. format, to display the value without using scientific notation:

select sum(population) format=16.
    into :totpop from sql.countries;

Output 5.4  Including a Macro Variable Reference in the Title

<table>
<thead>
<tr>
<th>City</th>
<th>AvgHigh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montreal</td>
<td>77.0</td>
</tr>
<tr>
<td>Quebec</td>
<td>76.0</td>
</tr>
<tr>
<td>Toronto</td>
<td>80.0</td>
</tr>
</tbody>
</table>

Creating Multiple Macro Variables
You can create one new macro variable per row from the result of a SELECT statement. Use the keywords THROUGH, THRU, or a hyphen (-) in an INTO clause to create a range of macro variables.

Note: When you specify a range of macro variables, the SAS macro facility creates only the number of macro variables that are needed. For example, if you specify :var1-:var9999 and only 55 variables are needed, only :var1-:var55 is created. The SQLOBS automatic variable is useful if a subsequent part of your program needs to know how many variables were actually created. In this example, SQLOBS would have a value of 55.

This example assigns values to macro variables from the first four rows of the Name column and the first three rows of the Population column. The %PUT statements write the results to the SAS log.

libname sql 'SAS-library';

proc sql noprint;
    select name, Population
        into :country1 - :country4, :pop1 - :pop3
        from sql.countries;

    %put &country1 &pop1;
    %put &country2 &pop2;
    %put &country3 &pop3;
    %put &country4;
Creating Multiple Macro Variables

4 proc sql noprint;
5   select name, Population
6      into :country1 - :country4, :pop1 - :pop3
7      from sql.countries;
8
9  %put &country1 &pop1;
  Afghanistan 17070323
10 %put &country2 &pop2;
  Albania 3407400
11 %put &country3 &pop3;
  Algeria 28171132
12 %put &country4;
  Andorra

Concatenating Values in Macro Variables

You can concatenate the values of one column into one macro variable. This form is useful for building a list of variables or constants. Use the SEPARATED BY keywords to specify a character to delimit the values in the macro variable.

This example assigns the first five values from the Name column of the Countries table to the &countries macro variable. The INOBS option limits PROC SQL to using the first five rows of the Countries table. A comma and a space are used to delimit the values in the macro variable.

libname sql 'SAS-library';

proc sql noprint inobs=5;
   select Name
        into :countries separated by ', ';
   from sql.countries;

   %put &countries;

Log 5.7 Concatenating Values in Macro Variables

4 proc sql noprint inobs=5;
5   select Name
6      into :countries separated by ', '
7     from sql.countries;
WARNING: Only 5 records were read from SQL.COUNTRIES due to INOBS= option.
8
9  %put &countries;
  Afghanistan, Albania, Algeria, Andorra, Angola

The leading and trailing blanks are trimmed from the values before the macro variables are created. If you do not want the blanks to be trimmed, then add NOTRIM to the INTO clause. Here is the previous example with NOTRIM added:

libname sql 'SAS-library';

proc sql noprint inobs=5;
   select Name
        into :countries separated by ', ', NOTRIM
   from sql.countries;
Log 5.8  Concatenating Values in Macro Variables

```sas
1 proc sql noprint inobs=5;
2 select Name
3 into :countries separated by ',' NOTRIM
4 from sql.countries;
WARNING: Only 5 records were read from SQL.COUNTRIES due to INOBS= option.
5 %put &countries;
Afghanistan                        ,Albania                            ,Algeria
,Andorra                            ,Angola
```

**Defining Macros to Create Tables**

Macros are useful as interfaces for table creation. You can use the SAS macro facility to help you create new tables and add rows to existing tables.

The following example creates a table that lists people to serve as referees for reviews of academic papers. No more than three people per subject are allowed in a table. The macro that is defined in this example checks the number of referees before it inserts a new referee's name into the table. The macro has two parameters: the referee's name and the subject matter of the academic paper.

```sas
libname sql 'SAS-library';

proc sql;
create table sql.referee
  (Name      char(15),
   Subject   char(15));
/* define the macro */
%macro addref(name,subject);
%local count;
/* are there three referees in the table? */
reset noprint;
select count(*)
into :count
from sql.referee
where subject="&subject";
%if &count ge 3 %then %do;
reset print;
title "ERROR: &name not inserted for subject – &subject."
title2 "There are 3 referees already.";
select * from sql.referee where subject="&subject";
reset noprint;
%end;
%else %do;
insert into sql.referee(name,subject) values("&name","&subject");
%put NOTE: &name has been added for subject – &subject.;
%end;
```
Submit the `%ADDREF()` macro with its two parameters to add referee names to the table. Each time you submit the macro, a message is written to the SAS log.

```sas
%addref(Conner,sailing);
%addref(Fay,sailing);
%addref(Einstein,relativity);
%addref(Smythe,sailing);
%addref(Naish,sailing);
```

**Log 5.9  Defining Macros to Create Tables**

The output has a row added with each execution of the `%ADDREF()` macro. When the table contains three referee names, it is displayed in SAS output with the message that it can accept no more referees.

**Output 5.5  Result Table and Message Created with SAS Macro Language Interface**

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conner</td>
<td>sailing</td>
</tr>
<tr>
<td>Fay</td>
<td>sailing</td>
</tr>
<tr>
<td>Smythe</td>
<td>sailing</td>
</tr>
</tbody>
</table>
```

**Using the PROC SQL Automatic Macro Variables**

PROC SQL sets up macro variables with certain values after it executes each statement. These macro variables can be tested inside a macro to determine whether to continue executing the PROC SQL step.

After each PROC SQL statement has executed, the following macro variables are updated with these values:
SQLEXITCODE
contains the highest return code that occurred from some types of SQL insert failures. This return code is written to the SYSERR macro variable when PROC SQL terminates.

SQLOBS
contains the number of rows that were processed by an SQL procedure statement. For example, the SQLOBS macro variable contains the number of rows that were formatted and displayed in SAS output by a SELECT statement or the number of rows that were deleted by a DELETE statement.

When the NOPRINT option is specified, the value of the SQLOBS macro variable depends on whether an output table, single macro variable, macro variable list, or macro variable range is created:

• If no output table, macro variable list, or macro variable range is created, then SQLOBS contains the value 1.
• If an output table is created, then SQLOBS contains the number of rows in the output table.
• If a single macro variable is created, then SQLOBS contains the value 1.
• If a macro variable list or macro variable range is created, then SQLOBS contains the number of rows that are processed to create the macro variable list or range.

If an SQL view is created, then SQLOBS contains the value 0.

Note: The SQLOBS automatic macro variable is assigned a value after the SQL SELECT statement executes.

SQLOOPS
contains the number of iterations that the inner loop of PROC SQL processes. The number of iterations increases proportionally with the complexity of the query. For more information, see “Limiting Iterations with the LOOPS= Option” on page 139 and LOOPS= in the Base SAS Procedures Guide.

SQLXOPENERRS
contains the number of DICTIONARY tables that failed to open. A new SQLXOPENERRS macro variable is created for each DICTIONARY request. The macro variable is created regardless of whether the DICTDIAG option is in effect. Information from the SQLXOPENERRS macro complements the information that you can retrieve with other macro variables such as SYSRC and SYSERR.

SQLRC
contains the following status values that indicate the success of the SQL procedure statement:

0
PROC SQL statement completed successfully with no errors.
4
PROC SQL statement encountered a situation for which it issued a warning. The statement continued to execute.
8
PROC SQL statement encountered an error. The statement stopped execution at this point.
PROC SQL statement encountered an internal error, indicating a bug in PROC SQL that should be reported to SAS Technical Support. These errors can occur only during compile time.

PROC SQL statement encountered a user error. For example, this error code is used, when a subquery (that can return only a single value) evaluates to more than one row. These errors can be detected only during run time.

PROC SQL statement encountered a system error. For example, this error is used, if the system cannot write to a PROC SQL table because the disk is full. These errors can occur only during run time.

PROC SQL statement encountered an internal error, indicating a bug in PROC SQL that should be reported to SAS Technical Support. These errors can occur only during run time.

The value of SQLRC can vary based on the value of the PROC SQL statement UNDO_POLICY= option or the SQLUNDOPOLICY system option.

For example, the values for the SQLRC return code differ based on the value of the UNDO_POLICY= option or the SQLUNDOPOLICY system option if you attempt to insert duplicate values into an index that is defined using the CREATE UNIQUE INDEX statement:

- If you set the UNDO_POLICY= option or the SQLUNDOPOLICY system option to either REQUIRED or OPTIONAL, and you attempt to insert a duplicate index value, SAS creates and tries to maintain a copy of the table before and after updates are applied. SAS detects an error condition and supplies a return code to PROC SQL, which stops execution as soon as the error condition is received. SQLRC contains the value 24.

- If you set the UNDO_POLICY= option or the SQLUNDOPOLICY system option to NONE and you attempt to insert a duplicate index value, SAS does not create a before-and-after copy of the table. SAS does not detect an error condition and does not supply a return code to PROC SQL, which attempts to continue to process the updates. SQLRC contains the value 8.

SQLXMSG contains descriptive information and the DBMS-specific return code for the error that is returned by the pass-through facility.

Note: Because the value of the SQLXMSG macro variable can contain special characters (such as &, %, /, *, and ;), use the %SUPERQ macro function when printing the following value: %put %superq(sqlxmsg); For information about the %SUPERQ function, see SAS Macro Language: Reference.

SQLXRC contains the DBMS-specific return code that is returned by the pass-through facility.

Macro variables that are generated by PROC SQL follow the scoping rules for %LET. For more information about macro variable scoping, see SAS Macro Language: Reference.

Users of SAS/AF software can access these automatic macro variables in SAS Component Language (SCL) programs by using the SYMGET function. The following example uses the VALIDATE statement in a SAS/AF software application to check the syntax of a block of code. Before it issues the CREATE VIEW statement, the application checks that the view is accessible.
submit sql immediate;
  validate &viewdef;
end submit;

if symget('SQLRC') gt 4 then
do;
  ... the view is not valid ...
end;
else do;
  submit sql immediate;
    create view &viewname as &viewdef;
  end submit;
end;

The following example retrieves the data from the Countries table, but does not display the table because the NOPRINT option is specified in the PROC SQL statement. The %PUT macro language statement displays the three automatic macro variable values in the SAS log. For more information about the %PUT statement and the SAS macro facility, see SAS Macro Language: Reference.

libname sql 'SAS-library';

proc sql noprint;
select * from sql.countries;
%put SQLOBS=*&sqlobs* SQLOOPS=*&sqloops* SQLRC=*&sqlrc*;

Log 5.10 Using the PROC SQL Automatic Macro Variables

SQLOBS=1* SQLOOPS=11* SQLRC=0*

Notice that the value of SQLOBS is 1. When the NOPRINT option is used and no table or macro variables are created, SQLOBS returns a value of 1 because only one row is processed.

Note: You can use the _AUTOMATIC_ option in the %PUT statement to list the values of all automatic macro variables. The list depends on the SAS products that are installed at your site.

Formatting PROC SQL Output By Using the REPORT Procedure

SQL provides limited output formatting capabilities. Some SQL vendors add output formatting statements to their products to address these limitations. SAS has reporting tools that enhance the appearance of PROC SQL output.

For example, SQL cannot display only the first occurrence of a repeating value in a column in its output. The following example lists cities in the USCityCoords table. Notice the repeating values in the State column.

libname sql 'SAS-library';

proc sql outobs=10;
  title 'US Cities';
  select State, City, latitude, Longitude
The following code uses PROC REPORT to format the output so that the state codes appear only once for each state group. A WHERE clause subsets the data so that the report lists the coordinates of cities in Pacific Rim states only. For more information about PROC REPORT, see the *Base SAS Procedures Guide*.

```
libname sql 'SAS-library';

proc sql noprint;
  create table sql.cityreport as
  select *
  from sql.uscitycoords
  order by state;
  
  proc report data=sql.cityreport
  headline nowd
  headskip;
  title 'Coordinates of U.S. Cities in Pacific Rim States';
  column state city ('Coordinates' latitude longitude);
  define state / order format=$2. width=5 'State';
  define city / order format=$15. width=15 'City';
  define latitude / display format=4. width=8 'Latitude';
  define longitude / display format=4. width=9 'Longitude';
  where state='AK' or
  state='HI' or
  state='WA' or
  state='OR' or
  state='CA';
  run;
```
Accessing a DBMS with SAS/ACCESS Software

Overview of Accessing a DBMS with SAS/ACCESS Software

SAS/ACCESS software for relational databases provides an interface between SAS software and data in other vendors' database management systems (DBMSs). SAS/ACCESS software provides dynamic access to DBMS data through the
SAS/ACCESS LIBNAME statement and the PROC SQL pass-through facility. The LIBNAME statement enables you to assign SAS librefs to DBMS objects such as schemas and databases. The pass-through facility enables you to interact with a DBMS by using its SQL syntax without leaving your SAS session.

It is recommended that you use the SAS/ACCESS LIBNAME statement to access your DBMS data because it is usually the fastest and most direct method of accessing DBMS data. The LIBNAME statement offers the following advantages:

- Significantly fewer lines of SAS code are required to perform operations in your DBMS. For example, a single LIBNAME statement establishes a connection to your DBMS, enables you to specify how your data is processed, and enables you to easily browse your DBMS tables in SAS.

- You do not need to know your DBMS's SQL language to access and manipulate your DBMS data. You can use SAS procedures, such as PROC SQL, or DATA step programming on any libref that references DBMS data. You can read, insert, update, delete, and append data, as well as create and drop DBMS tables by using normal SAS syntax.

- The LIBNAME statement provides more control over DBMS operations such as locking, spooling, and data type conversion through the many LIBNAME options and data set options.

- The LIBNAME engine optimizes the processing of joins and WHERE clauses by passing these operations directly to the DBMS to take advantage of the indexing and other processing capabilities of your DBMS.

An exception to this recommendation occurs when you need to use SQL that does not conform to the ANSI standard. The SAS/ACCESS LIBNAME statement accepts only ANSI standard for SQL, but the PROC SQL pass-through facility accepts all the extensions to SQL that are provided by your DBMS. Another advantage of this access method is that pass-through facility statements enable the DBMS to optimize queries when the queries have summary functions (such as AVG and COUNT), GROUP BY clauses, or columns that were created by expressions (such as the COMPUTED function).

For more information about SAS/ACCESS software, see SAS/ACCESS for Relational Databases: Reference.

Connecting to a DBMS By Using the LIBNAME Statement

Overview of Connecting to a DBMS By Using the LIBNAME Statement

Use the LIBNAME statement to read from and write to a DBMS object as if it were a SAS data set. After connecting to a DBMS table or view using the LIBNAME statement, you can use PROC SQL to interact with the DBMS data.

For many DBMSs, you can directly access DBMS data by assigning a libref to the DBMS using the SAS/ACCESS LIBNAME statement. Once you have associated a libref with the DBMS, you can specify a DBMS table in a two-level SAS name and work with the table like it is a SAS data set. You can also embed the LIBNAME statement in a PROC SQL view. For more information, see the “CREATE VIEW Statement” on page 243.

PROC SQL takes advantage of the capabilities of a DBMS by passing it certain operations whenever possible. For example, before implementing a join, PROC SQL checks to determine whether the DBMS can perform the join. If it can, then PROC SQL passes the join to the DBMS, which enhances performance by reducing data movement.
and translation. If the DBMS cannot perform the join, then PROC SQL processes the join. Using the SAS/ACCESS LIBNAME statement can often provide you with the performance benefits of the SQL procedure pass-through facility without writing DBMS-specific code.

**Note:** You can use the DBIDIRECTEXEC system option to send a PROC SQL CREATE TABLE AS SELECT statement, CREATE VIEW statement, DELETE statement, INSERT statement, and UPDATE statement directly to the database for execution, which can result in CPU and I/O performance improvement. For more information, see the SAS/ACCESS documentation for your DBMS.

To use the SAS/ACCESS LIBNAME statement, SAS/ACCESS software must be installed for your DBMS. For more information about the SAS/ACCESS LIBNAME statement, see the SAS/ACCESS documentation for your DBMS.

### Querying a DBMS Table

This example uses PROC SQL to query the Oracle table PAYROLL. The PROC SQL query retrieves all job codes and provides a total salary amount for each job code.

**Note:** By default, Oracle does not order the output results. To specify the order in which rows are displayed in the output results, you must use the ORDER BY clause in the SELECT statement.

```sql
libname mydblib oracle user=user-id password=password
    path=path-name schema=schema-name;

proc sql;
    select jobcode label='Jobcode',
        sum(salary) as total
        label='Total for Group'
        format=dollar11.2
    from mydblib.payroll
    group by jobcode;
quit;
```
Creating a PROC SQL View of a DBMS Table

PROC SQL views are stored query expressions that read data values from their underlying files, which can include SAS/ACCESS views of DBMS data. Although DATA step views of DBMS data can be used only to read the data, PROC SQL views of DBMS data can be used to update the underlying data if the following conditions are met:

- The PROC SQL view is based on only one DBMS table (or on a DBMS view that is based on only one DBMS table).
- The PROC SQL view has no calculated fields.

The following example uses the LIBNAME statement to connect to an ORACLE database, create a temporary PROC SQL view of the ORACLE table SCHEDULE, and print the view by using the PRINT procedure. The LIBNAME engine optimizes the processing of joins and WHERE clauses by passing these operations directly to the DBMS to take advantage of DBMS indexing and processing capabilities.

```plaintext
libname mydblib oracle user=user-id password=password;
proc sql;
create view LON as
  select flight, dates, idnum
```
from mydblib.schedule
where dest='LON';
quit;

proc print data=work.LON noobs;
run;

Output 5.9  PRINT Procedure Output

The SAS System

<table>
<thead>
<tr>
<th>FLIGHT</th>
<th>DATES</th>
<th>IDNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>219</td>
<td>01MAR1998:00:00:00</td>
<td>1407</td>
</tr>
<tr>
<td>219</td>
<td>01MAR1998:00:00:00</td>
<td>1777</td>
</tr>
<tr>
<td>219</td>
<td>01MAR1998:00:00:00</td>
<td>1103</td>
</tr>
<tr>
<td>219</td>
<td>01MAR1998:00:00:00</td>
<td>1125</td>
</tr>
<tr>
<td>219</td>
<td>01MAR1998:00:00:00</td>
<td>1350</td>
</tr>
<tr>
<td>219</td>
<td>01MAR1998:00:00:00</td>
<td>1332</td>
</tr>
<tr>
<td>219</td>
<td>02MAR1998:00:00:00</td>
<td>1407</td>
</tr>
<tr>
<td>219</td>
<td>02MAR1998:00:00:00</td>
<td>1118</td>
</tr>
<tr>
<td>219</td>
<td>02MAR1998:00:00:00</td>
<td>1132</td>
</tr>
<tr>
<td>219</td>
<td>02MAR1998:00:00:00</td>
<td>1135</td>
</tr>
<tr>
<td>219</td>
<td>02MAR1998:00:00:00</td>
<td>1441</td>
</tr>
<tr>
<td>219</td>
<td>02MAR1998:00:00:00</td>
<td>1332</td>
</tr>
<tr>
<td>219</td>
<td>03MAR1998:00:00:00</td>
<td>1428</td>
</tr>
<tr>
<td>219</td>
<td>03MAR1998:00:00:00</td>
<td>1442</td>
</tr>
<tr>
<td>219</td>
<td>03MAR1998:00:00:00</td>
<td>1130</td>
</tr>
<tr>
<td>219</td>
<td>03MAR1998:00:00:00</td>
<td>1411</td>
</tr>
<tr>
<td>219</td>
<td>03MAR1998:00:00:00</td>
<td>1115</td>
</tr>
<tr>
<td>219</td>
<td>03MAR1998:00:00:00</td>
<td>1332</td>
</tr>
</tbody>
</table>

Connecting to a DBMS By Using the SQL Procedure Pass-Through Facility

What Is the Pass-Through Facility?
The SQL procedure pass-through facility enables you to send DBMS-specific SQL statements directly to a DBMS for execution. The pass-through facility uses a SAS/ACCESS interface engine to connect to the DBMS. Therefore, SAS/ACCESS software must be installed for your DBMS.
You submit SQL statements that are DBMS-specific. For example, you pass Transact-SQL statements to a SAP ASE database. The pass-through facility’s basic syntax is the same for all DBMSs. Only the statements that are used to connect to the DBMS and the SQL statements are DBMS-specific.

With the pass-through facility, you can perform the following tasks:

• Establish a connection with the DBMS by using a CONNECT statement and terminate the connection with the DISCONNECT statement.
• Send nonquery DBMS-specific SQL statements to the DBMS by using the EXECUTE statement.
• Retrieve data from the DBMS to be used in a PROC SQL query with the CONNECTION TO component in a SELECT statement’s FROM clause.

You can use the pass-through facility statements in a query, or you can store them in a PROC SQL view. When a view is stored, any options that are specified in the corresponding CONNECT statement are also stored. Thus, when the PROC SQL view is used in a SAS program, SAS can automatically establish the appropriate connection to the DBMS.

For more information, see the CONNECT statement, the DISCONNECT statement, the EXECUTE statement, and the CONNECTION TO statement in Appendix 1, “SQL Macro Variables and System Options,” on page 373, and the pass-through facility for relational databases in SAS/ACCESS for Relational Databases: Reference.

Note: SAS procedures that perform multipass processing cannot operate on PROC SQL views that store pass-through facility statements, because the pass-through facility does not allow reopening of a table after the first record has been retrieved. To work around this limitation, create a SAS data set from the view and use the SAS data set as the input data set.

Return Codes
As you use PROC SQL statements that are available in the pass-through facility, any errors are written to the SAS log. The return codes and messages that are generated by the pass-through facility are available to you through the SQLXRC and SQLXMSG macro variables. Both macro variables are described in “Using the PROC SQL Automatic Macro Variables” on page 162.

Pass-Through Example
In this example, SAS/ACCESS connects to an ORACLE database by using the alias ora2, selects all rows in the Staff table, and displays the first 15 rows of data by using PROC SQL.

```sql
proc sql outobs=15;
   connect to oracle as ora2 (user=user-id password=password);
   select * from connection to ora2 (select lname, fname, state from staff);
   disconnect from ora2;
quit;
```
You can update PROC SQL and SAS/ACCESS views by using the INSERT, DELETE, and UPDATE statements, under the following conditions:

- If the view accesses a DBMS table, then you must have been granted the appropriate authorization by the external database management system (for example, DB2). You must have installed the SAS/ACCESS software for your DBMS. For more information about SAS/ACCESS views, see the SAS/ACCESS interface guide for your DBMS.

- You can update only a single table through a view. The table cannot be joined to another table or linked to another table with a set-operator. The view cannot contain a subquery.

- You can update a column in a view by using the column's alias, but you cannot update a derived column—that is, a column that is produced by an expression. In the following example, you can update the column SS, but not WeeklySalary:

```
create view EmployeeSalaries as
    select Employee, SSNumber as SS,
           Salary/52 as WeeklySalary
    from employees;
```
- You cannot update a view that contains an ORDER BY.

Note: Beginning with SAS 9, PROC SQL views, the pass-through facility, and the SAS/ACCESS LIBNAME statement are the preferred ways to access relational DBMS data. SAS/ACCESS views are no longer recommended. You can convert existing SAS/ACCESS views to PROC SQL views by using the CV2VIEW procedure. For more information, see “CV2VIEW Procedure” in SAS/ACCESS for Relational Databases: Reference.

Using the Output Delivery System with PROC SQL

The Output Delivery System (ODS) enables you to produce the output from PROC SQL in a variety of different formats such as PostScript, HTML, or list output. ODS defines the structure of the raw output from SAS procedures and from the SAS DATA step. The combination of data with a definition of its output structure is called an output object. Output objects can be sent to any of the various ODS destinations, which include listing, HTML, output, and printer. When new destinations are added to ODS, they automatically become available to PROC SQL, to all other SAS procedures that support ODS, and to the DATA step. For more information about ODS, see the SAS Output Delivery System: User’s Guide.

The following example opens the HTML destination and specifies ODSOUT.HTM as the file that will contain the HTML output. The output from PROC SQL is sent to ODSOUT.HTM.

Note: This example uses filenames that might not be valid in all operating environments. To run the example successfully in your operating environment, you might need to change the file specifications.

Note: Some browsers require an extension of HTM or HTML on the filename.

```plaintext
libname sql 'SAS-library';

ods html body='odsout.htm';
 proc sql outobs=12;
   title 'Coordinates of U.S. Cities';
   select *
     from sql.uscitycoords;
```
### Coordinates of U.S. Cities

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td>NY</td>
<td>43</td>
<td>-74</td>
</tr>
<tr>
<td>Albuquerque</td>
<td>NM</td>
<td>36</td>
<td>-106</td>
</tr>
<tr>
<td>Amarillo</td>
<td>TX</td>
<td>35</td>
<td>-102</td>
</tr>
<tr>
<td>Anchorage</td>
<td>AK</td>
<td>61</td>
<td>-150</td>
</tr>
<tr>
<td>Annapolis</td>
<td>MD</td>
<td>39</td>
<td>-77</td>
</tr>
<tr>
<td>Atlanta</td>
<td>GA</td>
<td>34</td>
<td>-84</td>
</tr>
<tr>
<td>Augusta</td>
<td>ME</td>
<td>44</td>
<td>-70</td>
</tr>
<tr>
<td>Austin</td>
<td>TX</td>
<td>30</td>
<td>-98</td>
</tr>
<tr>
<td>Baker</td>
<td>OR</td>
<td>45</td>
<td>-118</td>
</tr>
<tr>
<td>Baltimore</td>
<td>MD</td>
<td>39</td>
<td>-76</td>
</tr>
</tbody>
</table>

Measurements are in metric tons.
Overview ................................................................. 178
Computing a Weighted Average .................................................. 178
  Problem ................................................................. 178
  Background Information ............................................... 178
  Solution ................................................................. 179
  How It Works ......................................................... 180
Comparing Tables ............................................................... 180
  Problem ................................................................. 180
  Background Information ............................................... 181
  Solution ................................................................. 181
  How It Works ......................................................... 182
Overlaying Missing Data Values ................................................ 182
  Problem ................................................................. 182
  Background Information ............................................... 182
  Solution ................................................................. 183
  How It Works ......................................................... 184
Computing Percentages within Subtotals ...................................... 185
  Problem ................................................................. 185
  Background Information ............................................... 185
  Solution ................................................................. 186
  How It Works ......................................................... 187
Counting Duplicate Rows in a Table ........................................... 187
  Problem ................................................................. 187
  Background Information ............................................... 187
  Solution ................................................................. 188
  How It Works ......................................................... 189
Expanding Hierarchical Data in a Table ....................................... 189
  Problem ................................................................. 189
  Background Information ............................................... 189
  Solution ................................................................. 190
  How It Works ......................................................... 191
Summarizing Data in Multiple Columns ....................................... 192
  Problem ................................................................. 192
  Background Information ............................................... 192
  Solution ................................................................. 193
  How It Works ......................................................... 193
Overview

This section shows you examples of solutions that PROC SQL can provide. Each example includes a statement of the problem to solve, background information that you must know to solve the problem, the PROC SQL solution code, and an explanation of how the solution works.

Computing a Weighted Average

Problem

You want to compute a weighted average of a column of values.

Background Information

There is one input table, called Sample, that contains the following data:
data Sample;
  do i=1 to 10;
    Value=2983*ranuni(135);
    Weight=120 + 33*rannor(135);
    if mod(i,2)=0 then Gender='M';
    else Gender='F';
    output;
  end;
  drop i;
  proc print data=Sample;
    title 'Sample Data for Weighted Average';
  run;

Output 6.1 Sample Input Table for Weighted Averages

Sample Data for Weighted Average

<table>
<thead>
<tr>
<th>Obs</th>
<th>Value</th>
<th>Weight</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2893.36</td>
<td>129.087</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>56.13</td>
<td>146.217</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>901.43</td>
<td>115.940</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>2942.68</td>
<td>114.344</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>621.16</td>
<td>144.331</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>361.50</td>
<td>133.897</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>2575.09</td>
<td>149.373</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>2157.07</td>
<td>127.069</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>690.73</td>
<td>79.873</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>2085.80</td>
<td>144.480</td>
<td>M</td>
</tr>
</tbody>
</table>

Solution

Use the following PROC SQL code to obtain weighted averages that are shown in the following output:

proc sql;
  title 'Weighted Averages from Sample Data';
  select Gender, sum(Value*Weight)/sum(Weight) as WeightedAverage
  from (select Gender, Value,
            case
                when Weight gt 0 then Weight
                else 0
            end as Weight
  from Sample)
  group by Gender;

This solution uses an in-line view to create a temporary table that eliminates the negative data values in the Weight column. The in-line view is a query that performs the following tasks:

- selects the Gender and Value columns.
- uses a CASE expression to select the value from the Weight column. If Weight is greater than zero, then it is retrieved. If Weight is less than zero, then a value of zero is used in place of the Weight value.

\[
\text{(select Gender, Value,}
\begin{align*}
\text{case} \\
\text{\quad when Weight>0 then Weight} \\
\text{\quad else 0} \\
\text{\quad end as Weight} \\
\text{from Sample)}
\end{align*}
\]

The first, or outer, SELECT statement in the query, performs the following tasks:

- selects the Gender column
- constructs a weighted average from the results that were retrieved by the in-line view

The weighted average is the sum of the products of Value and Weight divided by the sum of the Weights.

\[
\text{select Gender, sum(Value*Weight)/sum(Weight) as WeightedAverage}
\]

Finally, the query uses a GROUP BY clause to combine the data so that the calculation is performed for each gender.

\[
\text{group by Gender;}
\]

### Comparing Tables

**Problem**

You have two copies of a table. One of the copies has been updated. You want to see which rows have been changed.
Background Information

There are two tables, the OldStaff table and NewStaff table. The NewStaff table is a copy of the OldStaff table. Changes have been made to the NewStaff table. You want to find out what changes have been made.

Output 6.3  Sample Input Tables for Table Comparison

<table>
<thead>
<tr>
<th>Old Staff Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>5463</td>
</tr>
<tr>
<td>6574</td>
</tr>
<tr>
<td>7896</td>
</tr>
<tr>
<td>4352</td>
</tr>
<tr>
<td>5674</td>
</tr>
<tr>
<td>7902</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>9786</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New Staff Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>5463</td>
</tr>
<tr>
<td>6574</td>
</tr>
<tr>
<td>7896</td>
</tr>
<tr>
<td>4352</td>
</tr>
<tr>
<td>5674</td>
</tr>
<tr>
<td>7002</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>9786</td>
</tr>
<tr>
<td>2123</td>
</tr>
</tbody>
</table>

Solution

To display only the rows that have changed in the new version of the table, use the EXCEPT set operator between two SELECT statements.
proc sql;
  title 'Updated Rows';
  select * from newstaff
  except
  select * from oldstaff;

Output 6.4  Rows That Have Changed

<table>
<thead>
<tr>
<th>id</th>
<th>Last</th>
<th>First</th>
<th>Middle</th>
<th>Phone</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>2123</td>
<td>Chen</td>
<td>Bill</td>
<td>W.</td>
<td>661-8099</td>
<td>R4432</td>
</tr>
<tr>
<td>7896</td>
<td>Bridges</td>
<td>Georgina</td>
<td>W.</td>
<td>661-2231</td>
<td>S2987</td>
</tr>
<tr>
<td>9786</td>
<td>Thompson</td>
<td>John</td>
<td>C.</td>
<td>661-6781</td>
<td>R2343</td>
</tr>
</tbody>
</table>

How It Works

The EXCEPT operator returns rows from the first query that are not part of the second query. In this example, the EXCEPT operator displays only the rows that have been added or changed in the NewStaff table.

Note: Any rows that were deleted from the OldStaff table will not appear.

Overlaying Missing Data Values

Problem

You are forming teams for a new league by analyzing the averages of bowlers when they were members of other bowling leagues. When possible, you will use each bowler's most recent league average. However, if a bowler was not in a league last year, then you will use the bowler's average from the prior year.

Background Information

There are two tables, League1 and League2, that contain bowling averages for last year and the prior year respectively. The structure of the tables is not identical because the data was compiled by two different secretaries. However, the tables do contain essentially the same type of data.

data league1;
  input @1 Fullname $20. @21 Bowler $4. @29 AvgScore 3.;
cards;
  Alexander Delarge  4224    164
  John T Chance       4425
  Jack T Colton       4264
  1412    141
  Andrew Shepherd     4189    185
;
The following PROC SQL code combines the information from two tables, League1 and League2. The program uses all the values from the League1 table, if available, and replaces any missing values with the corresponding values from the League2 table. The results are shown in the following output.

```sql
proc sql;
  title "Averages from Last Year's League When Possible";
```
title2 "Supplemented when Available from Prior Year’s League"
select coalesce(lastyr.fullname,trim(prioryr.firstname)
       ||' '||prioryr.lastname)as Name format=$26.,
coalesce(lastyr.bowler,prioryr.amfno)as Bowler,
coalesce(lastyr.avgscore,prioryr.avgscore)as Average format=8.
from league1 as lastyr full join league2 as prioryr
on lastyr.bowler=prioryr.amfno
order by Bowler;

Output 6.6 PROC SQL Output for Overlaying Missing Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Bowler</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mickey Raymond</td>
<td>1412</td>
<td>141</td>
</tr>
<tr>
<td>Patrick O’Malley</td>
<td>4118</td>
<td>164</td>
</tr>
<tr>
<td>Andrew Shepherd</td>
<td>4189</td>
<td>185</td>
</tr>
<tr>
<td>Alexander Delarge</td>
<td>4224</td>
<td>164</td>
</tr>
<tr>
<td>Jack T Colton</td>
<td>4264</td>
<td>174</td>
</tr>
<tr>
<td>John T Chance</td>
<td>4425</td>
<td></td>
</tr>
</tbody>
</table>

How It Works

This solution uses a full join to obtain all rows from League1 as well as all rows from League2. The program uses the COALESCE function on each column so that, whenever possible, there is a value for each column of a row. Using the COALESCE function on a list of expressions that is enclosed in parentheses returns the first nonmissing value that is found. For each row, the following code returns the AvgScore column from League1 for Average:

\[
\text{coalesce(lastyr.avgscore,prioryr.avgscore) as Average format=8.}
\]

If this value of AvgScore is missing, then COALESCE returns the AvgScore column from League2 for Average. If this value of AvgScore is missing, then COALESCE returns a missing value for Average.

In the case of the Name column, the COALESCE function returns the value of FullName from League1 if it exists. If not, then the value is obtained from League2 by using both the TRIM function and concatenation operators to combine the first name and last name columns:

\[
\text{trim(prioryr.firstname)||' '||prioryr.lastname}
\]

Finally, the table is ordered by Bowler. The Bowler column is the result of the COALESCE function.

\[
\text{coalesce(lastyr.bowler,prioryr.amfno) as Bowler}
\]

Because the value is obtained from either table, you cannot confidently order the output by either the value of Bowler in League1 or the value of AMFNo in League 2, but only by the value that results from the COALESCE function.
Computing Percentages within Subtotals

Problem

You want to analyze answers to a survey question to determine how each state responded. Then you want to compute the percentage of each answer that a given state contributed. For example, what percentage of all NO responses came from North Carolina?

Background Information

There is one input table, called Survey, that contains the following data (the first ten rows are shown):

```latex
data survey;
  input State $ Answer $ @@;
datalines;
  NY YES NY YES NY YES NY YES NY YES NY YES NY YES NY NO NY NO NC YES
  NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC YES
  NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO PA YES PA YES PA YES PA YES PA YES PA YES PA YES
  PA YES PA YES PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO
  PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO
  VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES
  VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA NO
  VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO
  VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO

proc print data=Survey(obs=10);
title 'Sample Data for Subtotal Percentages';
run;
```
**Solution**

Use the following PROC SQL code to compute the subtotal percentages:

```sql
proc sql;
  title 'Survey Responses';
  select survey.Answer, State, count(State) as Count,
       calculated Count/Subtotal as Percent format=percent8.2
  from survey,
       (select Answer, count(*) as Subtotal from survey
    group by Answer) as survey2
  where survey.Answer=survey2.Answer
  group by survey.Answer, State;
quit;
```
How It Works

This solution uses a subquery to calculate the subtotal counts for each answer. The code joins the result of the subquery with the original table and then uses the calculated state count as the numerator and the subtotal from the subquery as the denominator for the percentage calculation.

The query uses a GROUP BY clause to combine the data so that the calculation is performed for State within each answer.

group by survey.Answer, State;

Counting Duplicate Rows in a Table

Problem

You want to count the number of duplicate rows in a table and generate an output column that shows how many times each row occurs.

Background Information

There is one input table, called Duplicates, that contains the following data:

data Duplicates;
    input LastName $ FirstName $ City $ State $;
    datalines;
Smith John Richmond Virginia
Johnson Mary Miami Florida
Smith John Richmond Virginia
Reed Sam Portland Oregon
Davis Karen Chicago Illinois
Davis Karen Chicago Illinois
Thompson Jennifer Houston Texas
Smith John Richmond Virginia
Johnson Mary Miami Florida

proc print data=Duplicates;
title 'Sample Data for Counting Duplicates';
run;

Output 6.9  Sample Input Table for Counting Duplicates

<table>
<thead>
<tr>
<th>Obs</th>
<th>LastName</th>
<th>FirstName</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>John</td>
<td>Richmond</td>
<td>Virginia</td>
</tr>
<tr>
<td>2</td>
<td>Johnson</td>
<td>Mary</td>
<td>Miami</td>
<td>Florida</td>
</tr>
<tr>
<td>3</td>
<td>Smith</td>
<td>John</td>
<td>Richmond</td>
<td>Virginia</td>
</tr>
<tr>
<td>4</td>
<td>Reed</td>
<td>Sam</td>
<td>Portland</td>
<td>Oregon</td>
</tr>
<tr>
<td>5</td>
<td>Davis</td>
<td>Karen</td>
<td>Chicago</td>
<td>Illinois</td>
</tr>
<tr>
<td>6</td>
<td>Davis</td>
<td>Karen</td>
<td>Chicago</td>
<td>Illinois</td>
</tr>
<tr>
<td>7</td>
<td>Thompson</td>
<td>Jennifer</td>
<td>Houston</td>
<td>Texas</td>
</tr>
<tr>
<td>8</td>
<td>Smith</td>
<td>John</td>
<td>Richmond</td>
<td>Virginia</td>
</tr>
<tr>
<td>9</td>
<td>Johnson</td>
<td>Mary</td>
<td>Miami</td>
<td>Florida</td>
</tr>
</tbody>
</table>

Solution

Use the following PROC SQL code to count the duplicate rows:

    proc sql;
    title 'Duplicate Rows in Duplicates Table';
    select *, count(*) as Count
    from Duplicates
    group by LastName, FirstName, City, State
    having count(*) > 1;

Output 6.10  PROC SQL Output for Counting Duplicates

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
<th>City</th>
<th>State</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Davis</td>
<td>Karen</td>
<td>Chicago</td>
<td>Illinois</td>
<td>2</td>
</tr>
<tr>
<td>Johnson</td>
<td>Mary</td>
<td>Miami</td>
<td>Florida</td>
<td>2</td>
</tr>
<tr>
<td>Smith</td>
<td>John</td>
<td>Richmond</td>
<td>Virginia</td>
<td>3</td>
</tr>
</tbody>
</table>
**How It Works**

This solution uses a query that performs the following:

- selects all columns
- counts all rows
- groups all of the rows in the Duplicates table by matching rows
- excludes the rows that have no duplicates

*Note:* You must include all of the columns in your table in the GROUP BY clause to find exact duplicates.

---

**Expanding Hierarchical Data in a Table**

**Problem**

You want to generate an output column that shows a hierarchical relationship among rows in a table.

**Background Information**

There is one input table, called Employees, that contains the following data:

```sql
data Employees;
  input ID $ LastName $ FirstName $ Supervisor $;
  datalines;
1001 Smith John 1002
1002 Johnson Mary None
1003 Reed Sam None
1004 Davis Karen 1003
1005 Thompson Jennifer 1002
1006 Peterson George 1002
1007 Jones Sue 1003
1008 Murphy Janice 1003
1009 Garcia Joe 1002
;
```

```sql
proc print data=Employees;
  title 'Sample Data for Expanding a Hierarchy';
run;
```
Output 6.11  Sample Input Table for Expanding a Hierarchy

<table>
<thead>
<tr>
<th>Obs</th>
<th>ID</th>
<th>LastName</th>
<th>FirstName</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1001</td>
<td>Smith</td>
<td>John</td>
<td>1002</td>
</tr>
<tr>
<td>2</td>
<td>1002</td>
<td>Johnson</td>
<td>Mary</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>1003</td>
<td>Reed</td>
<td>Sam</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>1004</td>
<td>Davis</td>
<td>Karen</td>
<td>1003</td>
</tr>
<tr>
<td>5</td>
<td>1005</td>
<td>Thompson</td>
<td>Jennifer</td>
<td>1002</td>
</tr>
<tr>
<td>6</td>
<td>1006</td>
<td>Peterson</td>
<td>George</td>
<td>1002</td>
</tr>
<tr>
<td>7</td>
<td>1007</td>
<td>Jones</td>
<td>Sue</td>
<td>1003</td>
</tr>
<tr>
<td>8</td>
<td>1008</td>
<td>Murphy</td>
<td>Janice</td>
<td>1003</td>
</tr>
<tr>
<td>9</td>
<td>1009</td>
<td>Garcia</td>
<td>Joe</td>
<td>1002</td>
</tr>
</tbody>
</table>

You want to create output that shows the full name and ID number of each employee who has a supervisor, along with the full name and ID number of that employee's supervisor.

Solution

Use the following PROC SQL code to expand the data:

```sql
proc sql;
  title 'Expanded Employee and Supervisor Data';
  select A.ID label="Employee ID",
         trim(A.FirstName)||' '||A.LastName label="Employee Name",
         B.ID label="Supervisor ID",
         trim(B.FirstName)||' '||B.LastName label="Supervisor Name"
    from Employees A, Employees B
    where A.Supervisor=B.ID and A.Supervisor is not missing;
```
How It Works

This solution uses a self-join (reflexive join) to match employees and their supervisors. The SELECT clause assigns aliases of A and B to two instances of the same table and retrieves data from each instance. From instance A, the SELECT clause performs the following:

- selects the ID column and assigns it a label of Employee ID
- selects and concatenates the FirstName and LastName columns into one output column and assigns it a label of Employee Name

From instance B, the SELECT clause performs the following:

- selects the ID column and assigns it a label of Supervisor ID
- selects and concatenates the FirstName and LastName columns into one output column and assigns it a label of Supervisor Name

In both concatenations, the SELECT clause uses the TRIM function to remove trailing spaces from the data in the FirstName column, and then concatenates the data with a single space and the data in the LastName column to produce a single character value for each full name.

\[ \text{trim(A.FirstName)} || \text{'} | \text{'} || \text{A.LastName label="Employee Name"} \]

When PROC SQL applies the WHERE clause, the two table instances are joined. The WHERE clause conditions restrict the output to only those rows in table A that have a supervisor ID that matches an employee ID in table B. This operation provides a supervisor ID and full name for each employee in the original table, except for those who do not have a supervisor.

\[ \text{where A.Supervisor=B.ID and A.Supervisor is not missing;} \]

Note: Although there are no missing values in the Employees table, you should check for and exclude missing values from your results to avoid unexpected results. For example, if there were an employee with a blank supervisor ID number and an employee with a blank ID, then they would produce an erroneous match in the results.
Summarizing Data in Multiple Columns

Problem

You want to produce a grand total of multiple columns in a table.

Background Information

There is one input table, called Sales, that contains the following data:

```plaintext
data Sales;
   input Salesperson $ January February March;
datalines;
Smith 1000 650 800
Johnson 0 900 900
Reed 1200 700 850
Davis 1050 900 1000
Thompson 750 850 1000
Peterson 900 600 500
Jones 800 900 1200
Murphy 700 800 700
Garcia 400 1200 1150
;
```

```plaintext
proc print data=Sales;
   title 'Sample Data for Summarizing Data from Multiple Columns';
run;
```
Sample Data for Summarizing Data from Multiple Columns

<table>
<thead>
<tr>
<th>Obs</th>
<th>Salesperson</th>
<th>January</th>
<th>February</th>
<th>March</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>1000</td>
<td>650</td>
<td>800</td>
</tr>
<tr>
<td>2</td>
<td>Johnson</td>
<td>0</td>
<td>900</td>
<td>900</td>
</tr>
<tr>
<td>3</td>
<td>Reed</td>
<td>1200</td>
<td>700</td>
<td>850</td>
</tr>
<tr>
<td>4</td>
<td>Davis</td>
<td>1050</td>
<td>900</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>Thompson</td>
<td>750</td>
<td>850</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>Peterson</td>
<td>900</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>7</td>
<td>Jones</td>
<td>800</td>
<td>900</td>
<td>1200</td>
</tr>
<tr>
<td>8</td>
<td>Murphy</td>
<td>700</td>
<td>800</td>
<td>700</td>
</tr>
<tr>
<td>9</td>
<td>Garcia</td>
<td>400</td>
<td>1200</td>
<td>1150</td>
</tr>
</tbody>
</table>

You want to create output that shows the total sales for each month and the total sales for all three months.

**Solution**

Use the following PROC SQL code to produce the monthly totals and grand total:

```
proc sql;
  title 'Total First Quarter Sales';
  select sum(January) as JanTotal,
         sum(February) as FebTotal,
         sum(March)   as MarTotal,
         sum(calculated JanTotal, calculated FebTotal,
              calculated MarTotal) as GrandTotal format=dollar10.
  from Sales;
```

**How It Works**

Recall that when you specify one column as the argument to an aggregate function, the values in that column are calculated. When you specify multiple columns, the values in each row of the columns are calculated. This solution uses the SUM function to calculate...
the sum of each month’s sales, and then uses the SUM function a second time to total the monthly sums into one grand total.

```
sum(calculated JanTotal, calculated FebTotal, calculated MarTotal) as GrandTotal format=dollar10.
```

An alternative way to code the grand total calculation is to use nested functions:

```
sum(sum(January), sum(February), sum(March)) as GrandTotal format=dollar10.
```

### Creating a Summary Report

**Problem**

You have a table that contains detailed sales information. You want to produce a summary report from the detail table.

**Background Information**

There is one input table, called Sales, that contains detailed sales information. There is one record for each sale for the first quarter that shows the site, product, invoice number, invoice amount, and invoice date.

```plaintext
data sales;
  input Site $ Product $ Invoice $ InvoiceAmount InvoiceDate $;
  datalines;
  V1009  VID010  V7679 598.5  980126
  V1019  VID010  V7688 598.5  980126
  V1032  VID005  V7771 1070  980309
  V1043  VID014  V7780 1070  980309
  V421   VID003  V7831 2000  980330
  V421   VID010  V7832  750  980330
  V570   VID003  V7762 2000  980302
  V659   VID003  V7730 1000  980223
  V783   VID003  V7815  750  980323
  V985   VID003  V7733 2500  980223
  V966   VID001  V5020 1167  980215
  V98    VID003  V7750 2000  980223
;  
proc sql;
  title 'Sample Data to Create Summary Sales Report';
  select * from sales;
quit;
```
Output 6.15  Sample Input Table for Creating a Summary Report

<table>
<thead>
<tr>
<th>Site</th>
<th>Product</th>
<th>Invoice</th>
<th>InvoiceAmount</th>
<th>InvoiceDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1009</td>
<td>VID010</td>
<td>V7679</td>
<td>598.5</td>
<td>980126</td>
</tr>
<tr>
<td>V1019</td>
<td>VID010</td>
<td>V7688</td>
<td>598.5</td>
<td>980126</td>
</tr>
<tr>
<td>V1032</td>
<td>VID005</td>
<td>V7771</td>
<td>1070</td>
<td>980309</td>
</tr>
<tr>
<td>V1043</td>
<td>VID014</td>
<td>V7780</td>
<td>1070</td>
<td>980309</td>
</tr>
<tr>
<td>V421</td>
<td>VID003</td>
<td>V7831</td>
<td>2000</td>
<td>980330</td>
</tr>
<tr>
<td>V421</td>
<td>VID010</td>
<td>V7832</td>
<td>750</td>
<td>980330</td>
</tr>
<tr>
<td>V570</td>
<td>VID003</td>
<td>V7762</td>
<td>2000</td>
<td>980302</td>
</tr>
<tr>
<td>V659</td>
<td>VID003</td>
<td>V7730</td>
<td>1000</td>
<td>980223</td>
</tr>
<tr>
<td>V783</td>
<td>VID003</td>
<td>V7815</td>
<td>750</td>
<td>980323</td>
</tr>
<tr>
<td>V985</td>
<td>VID003</td>
<td>V7733</td>
<td>2500</td>
<td>980223</td>
</tr>
<tr>
<td>V966</td>
<td>VID001</td>
<td>V5020</td>
<td>1167</td>
<td>980215</td>
</tr>
<tr>
<td>V98</td>
<td>VID003</td>
<td>V7750</td>
<td>2000</td>
<td>980223</td>
</tr>
</tbody>
</table>

You want to use this table to create a summary report that shows the sales for each product for each month of the quarter.

**Solution**

Use the following PROC SQL code to create a column for each month of the quarter, and use the summary function SUM in combination with the GROUP BY statement to accumulate the monthly sales for each product:

```
proc sql;
  title 'First Quarter Sales by Product';
  select Product,
      sum(Jan) label='Jan',
      sum(Feb) label='Feb',
      sum(Mar) label='Mar'
  from (select Product,
            case
              when substr(InvoiceDate,3,2)='01' then InvoiceAmount end as Jan,
              case
              when substr(InvoiceDate,3,2)='02' then InvoiceAmount end as Feb,
              case
              when substr(InvoiceDate,3,2)='03' then InvoiceAmount end as Mar
  from work.sales)
  group by Product;
```
Output 6.16  PROC SQL Output for a Summary Report

How It Works

This solution uses an in-line view to create three temporary columns, Jan, Feb, and Mar, based on the month part of the invoice date column. The in-line view is a query that performs the following:

- selects the product column
- uses a CASE expression to assign the value of invoice amount to one of three columns, Jan, Feb, or Mar, depending on the value of the month part of the invoice date column

```sql
CASE
  WHEN SUBSTR(InvoiceDate, 3, 2) = '01' THEN InvoiceAmount END AS Jan,
  WHEN SUBSTR(InvoiceDate, 3, 2) = '02' THEN InvoiceAmount END AS Feb,
  WHEN SUBSTR(InvoiceDate, 3, 2) = '03' THEN InvoiceAmount END AS Mar
```

The first, or outer, SELECT statement in the query performs the following:

- selects the product
- uses the summary function SUM to accumulate the Jan, Feb, and Mar amounts
- uses the GROUP BY statement to produce a line in the table for each product

Notice that dates are stored in the input table as strings. If the dates were stored as SAS dates, then the CASE expression could be written as follows:

```sql
CASE
  WHEN MONTH(InvoiceDate) = 1 THEN InvoiceAmount END AS Jan,
  WHEN MONTH(InvoiceDate) = 2 THEN InvoiceAmount END AS Feb,
  WHEN MONTH(InvoiceDate) = 3 THEN InvoiceAmount END AS Mar
```

Note: Missing values in the matrix indicate that no sales occurred for that given product in that month.
Creating a Customized Sort Order

Problem

You want to sort data in a logical, but not alphabetical, sequence.

Background Information

There is one input table, called Chores, that contains the following data:

```plaintext
data chores;
  input Project $ Hours Season $;
  datalines;
  weeding 48 summer
  pruning 12 winter
  mowing 36 summer
  mulching 17 fall
  raking 24 fall
  raking 16 spring
  planting 8 spring
  planting 8 fall
  sweeping 3 winter
  edging 16 summer
  seeding 6 spring
  tilling 12 spring
  aerating 6 spring
  feeding 7 summer
  rolling 4 winter
;
```

```plaintext
proc sql;
title 'Garden Chores';
select * from chores;
quit;
```
You want to reorder this chore list so that all the chores are grouped by season, starting with spring and progressing through the year. Simply ordering by Season makes the list appear in alphabetical sequence: fall, spring, summer, winter.

**Solution**

Use the following PROC SQL code to create a new column, Sorter, that will have values of 1 through 4 for the seasons spring through winter. Use the new column to order the query, but do not select it to appear:

```sql
proc sql;
title 'Garden Chores by Season in Logical Order';
select Project, Hours, Season,
     case
         when Season = 'spring' then 1
         when Season = 'summer' then 2
         when Season = 'fall' then 3
         when Season = 'winter' then 4
         else .
     end as Sorter
from (select Project, Hours, Season,
     case
         when Season = 'spring' then 1
         when Season = 'summer' then 2
         when Season = 'fall' then 3
         when Season = 'winter' then 4
         else .
     end as Sorter
     from chores)
order by Sorter;
```
**How It Works**

This solution uses an in-line view to create a temporary column that can be used as an ORDER BY column. The in-line view is a query that performs the following:

- selects the Project, Hours, and Season columns
- uses a CASE expression to remap the seasons to the new column Sorter: spring to 1, summer to 2, fall to 3, and winter to 4

```
(select project, hours, season,
 case
  when season = 'spring' then 1
  when season = 'summer' then 2
  when season = 'fall' then 3
  when season = 'winter' then 4
  else .
 end as sorter
from chores)
```

The first, or outer, SELECT statement in the query performs the following:

- selects the Project, Hours, and Season columns

---

**Garden Chores by Season in Logical Order**

<table>
<thead>
<tr>
<th>Project</th>
<th>Hours</th>
<th>Season</th>
</tr>
</thead>
<tbody>
<tr>
<td>tilling</td>
<td>12</td>
<td>spring</td>
</tr>
<tr>
<td>raking</td>
<td>16</td>
<td>spring</td>
</tr>
<tr>
<td>planting</td>
<td>8</td>
<td>spring</td>
</tr>
<tr>
<td>seeding</td>
<td>6</td>
<td>spring</td>
</tr>
<tr>
<td>aerating</td>
<td>6</td>
<td>spring</td>
</tr>
<tr>
<td>mowing</td>
<td>36</td>
<td>summer</td>
</tr>
<tr>
<td>feeding</td>
<td>7</td>
<td>summer</td>
</tr>
<tr>
<td>edging</td>
<td>16</td>
<td>summer</td>
</tr>
<tr>
<td>weeding</td>
<td>48</td>
<td>summer</td>
</tr>
<tr>
<td>raking</td>
<td>24</td>
<td>fall</td>
</tr>
<tr>
<td>mulching</td>
<td>17</td>
<td>fall</td>
</tr>
<tr>
<td>planting</td>
<td>8</td>
<td>fall</td>
</tr>
<tr>
<td>rolling</td>
<td>4</td>
<td>winter</td>
</tr>
<tr>
<td>pruning</td>
<td>12</td>
<td>winter</td>
</tr>
<tr>
<td>sweeping</td>
<td>3</td>
<td>winter</td>
</tr>
</tbody>
</table>
orders rows by the values that were assigned to the seasons in the Sorter column that was created with the in-line view

Notice that the Sorter column is not included in the SELECT statement. That causes a note to be written to the log indicating that you have used a column in an ORDER BY statement that does not appear in the SELECT statement. In this case, that is exactly what you wanted to do.

### Conditionally Updating a Table

#### Problem

You want to update values in a column of a table, based on the values of several other columns in the table.

#### Background Information

There is one table, called Incentives, that contains information about sales data. There is one record for each salesperson that includes a department code, a base pay rate, and sales of two products, gadgets and whatnots.

```sql
data incentives;
  input @1 Name $18. @20 Department $2. Payrate Gadgets Whatnots;
datalines;
  Lao Che            M2   8.00 10193 1105
  Jack Colton        U2   6.00 9994 2710
  Mickey Raymond     M1  12.00 6103 1930
  Dean Proffit       M2  11.00 3000 1999
  Antoinette Lily    E1  20.00 2203 4610
  Sydney Wade        E2  15.00 4205 3010
  Alan Traherne      U2   4.00 5020 3000
  Elizabeth Bennett  E1  16.00 17003 3003
;
```

```sql
proc sql;
  title 'Sales Data for Incentives Program';
  select * from incentives;
quit;
```
Sample Input Data to Conditionally Change a Table

You want to update the table by increasing each salesperson’s pay rate (based on the total sales of gadgets and whatnots) and taking into consideration some factors that are based on department code.

Specifically, anyone who sells more than 10,000 gadgets merits an extra $5 per hour. Anyone selling between 5,000 and 10,000 gadgets also merits an incentive pay, but E Department salespersons are expected to be better sellers than those in the other departments, so their gadget sales incentive is $2 per hour compared to $3 per hour for those in other departments. Good sales of whatnots also entitle sellers to added incentive pay. The algorithm for whatnot sales is that the top level (level 1 in each department) salespersons merit an extra $.50 per hour for whatnot sales more than 2,000, and level 2 salespersons merit an extra $1 per hour for sales more than 2,000.

Solution

Use the following PROC SQL code to create a new value for the Payrate column. Actually Payrate is updated twice for each row, once based on sales of gadgets, and again based on sales of whatnots:

```sql
proc sql;
  update incentives
  set payrate = case
    when gadgets > 10000 then
      payrate + 5.00
    when gadgets > 5000 then
      case
        when department in ('E1', 'E2') then
          payrate + 2.00
        else payrate + 3.00
      end
    end;
  update incentives
  set payrate = case
    when whatnots > 2000 then
      .50
    else payrate
  end;
```

### Sales Data for Incentives Program

<table>
<thead>
<tr>
<th>Name</th>
<th>Department</th>
<th>Payrate</th>
<th>Gadgets</th>
<th>Whatnots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lao Che</td>
<td>M2</td>
<td>8</td>
<td>10193</td>
<td>1105</td>
</tr>
<tr>
<td>Jack Colton</td>
<td>U2</td>
<td>6</td>
<td>9994</td>
<td>2710</td>
</tr>
<tr>
<td>Mickey Raymond</td>
<td>M1</td>
<td>12</td>
<td>6103</td>
<td>1930</td>
</tr>
<tr>
<td>Dean Proft</td>
<td>M2</td>
<td>11</td>
<td>3000</td>
<td>1999</td>
</tr>
<tr>
<td>Antoinette Lily</td>
<td>E1</td>
<td>20</td>
<td>2203</td>
<td>4610</td>
</tr>
<tr>
<td>Sydney Wade</td>
<td>E2</td>
<td>15</td>
<td>4205</td>
<td>3010</td>
</tr>
<tr>
<td>Alan Traherne</td>
<td>U2</td>
<td>4</td>
<td>5020</td>
<td>3000</td>
</tr>
<tr>
<td>Elizabeth Bennett</td>
<td>E1</td>
<td>16</td>
<td>17003</td>
<td>3003</td>
</tr>
</tbody>
</table>
case
  when department in ('E2', 'M2', 'U2') then
    payrate + 1.00
  else payrate + 0.50
end
else payrate
end;

title 'Adjusted Payrates Based on Sales of Gadgets and Whatnots';
select * from incentives;

Output 6.20  PROC SQL Output for Conditionally Updating a Table

### Adjusted Payrates Based on Sales of Gadgets and Whatnots

<table>
<thead>
<tr>
<th>Name</th>
<th>Department</th>
<th>Payrate</th>
<th>Gadgets</th>
<th>Whatnots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lao Che</td>
<td>M2</td>
<td>13</td>
<td>10193</td>
<td>1105</td>
</tr>
<tr>
<td>Jack Colton</td>
<td>U2</td>
<td>10</td>
<td>9994</td>
<td>2710</td>
</tr>
<tr>
<td>Mickey Raymond</td>
<td>M1</td>
<td>15</td>
<td>6103</td>
<td>1930</td>
</tr>
<tr>
<td>Dean Profit</td>
<td>M2</td>
<td>11</td>
<td>3000</td>
<td>1999</td>
</tr>
<tr>
<td>Antoinette Lily</td>
<td>E1</td>
<td>20.5</td>
<td>2203</td>
<td>4610</td>
</tr>
<tr>
<td>Sydney Wade</td>
<td>E2</td>
<td>16</td>
<td>4205</td>
<td>3010</td>
</tr>
<tr>
<td>Alan Traherne</td>
<td>U2</td>
<td>8</td>
<td>5020</td>
<td>3000</td>
</tr>
<tr>
<td>Elizabeth Bennett</td>
<td>E1</td>
<td>21.5</td>
<td>17003</td>
<td>3003</td>
</tr>
</tbody>
</table>

**How It Works**

This solution performs consecutive updates to the Payrate column of the incentive table. The first update uses a nested case expression, first determining a bracket that is based on the amount of gadget sales: greater than 10,000 calls for an incentive of $5, between 5,000 and 10,000 requires an additional comparison. That is accomplished with a nested case expression that checks department code to choose between a $2 and $3 incentive.

```sql
update incentives
set payrate = case
  when gadgets > 10000 then
    payrate + 5.00
  when gadgets > 5000 then
    case
      when department in ('E1', 'E2') then
        payrate + 2.00
      else payrate + 3.00
    end
  else payrate
end;
```
The second update is similar, though simpler. All sales of whatnots more than 2,000 merit an incentive, either $.50 or $1 depending on the department level, that again is accomplished by means of a nested case expression.

```sql
update incentives
set payrate = case
    when whatnots > 2000 then
        case
            when department in ('E2', 'M2', 'U2') then payrate + 1.00
            else payrate + 0.50
        end
    else payrate
end;
```

---

**Updating a Table with Values from Another Table**

**Problem**

You want to update the Sql.United States table with updated population data.

**Background Information**

The Sql.NewPop table contains updated population data for some of the U.S. states.

```sql
libname sql 'SAS-library';

proc sql;
title 'Updated U.S. Population Data';
select state, population format=comma10. label='Population' from sql.newpop;
```
Solution

Use the following PROC SQL code to update the population information for each state in the Sql.United States table:

```sql
proc sql;
title 'UnitedStates';
update sql.unitedstates as u
    set population=(select population from sql.newpop as n
        where u.name=n.state)
    where u.name in (select state from sql.newpop);
select Name format=$17., Capital format=$15.,
    Population, Area, Continent format=$13., Statehood format=date9.
    from sql.unitedstates;
/* use this code to generate output so you don't
    overwrite the sql.unitedstates table */
options ls=84;
proc sql outobs=10;
title 'UnitedStates';
create table work.unitedstates as
    select * from sql.unitedstates;
update work.unitedstates as u
    set population=(select population from sql.newpop as n
        where u.name=n.state)
    where u.name in (select state from sql.newpop);
```

Output 6.21  Table with Updated Population Data

<table>
<thead>
<tr>
<th>state</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texas</td>
<td>20,851,820</td>
</tr>
<tr>
<td>Georgia</td>
<td>8,185,453</td>
</tr>
<tr>
<td>Washington</td>
<td>5,894,121</td>
</tr>
<tr>
<td>Arizona</td>
<td>5,130,632</td>
</tr>
<tr>
<td>Alabama</td>
<td>4,447,100</td>
</tr>
<tr>
<td>Oklahoma</td>
<td>3,450,654</td>
</tr>
<tr>
<td>Connecticut</td>
<td>3,405,565</td>
</tr>
<tr>
<td>Iowa</td>
<td>2,926,324</td>
</tr>
<tr>
<td>West Virginia</td>
<td>1,808,344</td>
</tr>
<tr>
<td>Idaho</td>
<td>1,293,953</td>
</tr>
<tr>
<td>Maine</td>
<td>1,274,923</td>
</tr>
<tr>
<td>New Hampshire</td>
<td>1,235,786</td>
</tr>
<tr>
<td>North Dakota</td>
<td>642,200</td>
</tr>
<tr>
<td>Alaska</td>
<td>626,932</td>
</tr>
</tbody>
</table>

where u.name=n.state)
where u.name in (select state from sql.newpop);
select Name format=$17., Capital format=$15.,
Population, Area, Continent format=$13., Statehood format=date9.
from work.unitedstates
;

Output 6.22  Sql.UnitedStates with Updated Population Data (Partial Output)

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
<th>Area</th>
<th>Continent</th>
<th>Statehood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>Montgomery</td>
<td>4447100</td>
<td>52423</td>
<td>North America</td>
<td>14DEC1819</td>
</tr>
<tr>
<td>Alaska</td>
<td>Juneau</td>
<td>626932</td>
<td>656400</td>
<td>North America</td>
<td>03JAN1959</td>
</tr>
<tr>
<td>Arizona</td>
<td>Phoenix</td>
<td>5130632</td>
<td>114000</td>
<td>North America</td>
<td>14FEB1912</td>
</tr>
<tr>
<td>Arkansas</td>
<td>Little Rock</td>
<td>2447996</td>
<td>53200</td>
<td>North America</td>
<td>15JUN1836</td>
</tr>
<tr>
<td>California</td>
<td>Sacramento</td>
<td>31518948</td>
<td>163700</td>
<td>North America</td>
<td>09SEP1850</td>
</tr>
<tr>
<td>Colorado</td>
<td>Denver</td>
<td>3601298</td>
<td>104100</td>
<td>North America</td>
<td>01AUG1876</td>
</tr>
<tr>
<td>Connecticut</td>
<td>Hartford</td>
<td>3405565</td>
<td>5500</td>
<td>North America</td>
<td>09JAN1878</td>
</tr>
<tr>
<td>Delaware</td>
<td>Dover</td>
<td>707232</td>
<td>2500</td>
<td>North America</td>
<td>07DEC1877</td>
</tr>
<tr>
<td>District of Colum</td>
<td>Washington</td>
<td>612907</td>
<td>100</td>
<td>North America</td>
<td>21FEB1871</td>
</tr>
<tr>
<td>Florida</td>
<td>Tallahassee</td>
<td>13814408</td>
<td>65800</td>
<td>North America</td>
<td>03MAR1845</td>
</tr>
</tbody>
</table>

How It Works

The UPDATE statement updates values in the Sql.UnitedStates table (here with the alias U). For each row in the Sql.UnitedStates table, the in-line view in the SET clause returns a single value. For rows that have a corresponding row in Sql.NewPop, this value is the value of the Population column from Sql.NewPop. For rows that do not have a corresponding row in Sql.NewPop, this value is missing. In both cases, the returned value is assigned to the Population column.

The WHERE clause ensures that only the rows in Sql.UnitedStates that have a corresponding row in Sql.NewPop are updated by checking each value of Name against the list of state names that is returned from the in-line view. Without the WHERE clause, the Population values are updated to “missing” for rows that do not have a corresponding row in Sql.NewPop.

Creating and Using Macro Variables

Problem

You want to create a separate data set for each unique value of a column.
Background Information

The Sql.Features data set contains information about various geographical features around the world.

```sql
libname sql 'SAS-library';
proc sql outobs=10;
title 'Features';
select Name format=$15., Type,Location format =$15.,Area,
    Height, Depth, Length
from sql.features;
```

Output 6.23  Features (Partial Output)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Location</th>
<th>Area</th>
<th>Height</th>
<th>Depth</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aconcagua</td>
<td>Mountain</td>
<td>Argentina</td>
<td>.</td>
<td>22834</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Amazon</td>
<td>River</td>
<td>South America</td>
<td>.</td>
<td>.</td>
<td>4000</td>
<td>.</td>
</tr>
<tr>
<td>Amur</td>
<td>River</td>
<td>Asia</td>
<td>.</td>
<td>.</td>
<td>2700</td>
<td>.</td>
</tr>
<tr>
<td>Andaman</td>
<td>Sea</td>
<td></td>
<td>218100</td>
<td>.</td>
<td>3667</td>
<td>.</td>
</tr>
<tr>
<td>Angel Falls</td>
<td>Waterfall</td>
<td>Venezuela</td>
<td>.</td>
<td>3212</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Annapurna</td>
<td>Mountain</td>
<td>Nepal</td>
<td>.</td>
<td>26504</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Aral Sea</td>
<td>Lake</td>
<td>Asia</td>
<td>25300</td>
<td>.</td>
<td>222</td>
<td>.</td>
</tr>
<tr>
<td>Ararat</td>
<td>Mountain</td>
<td>Turkey</td>
<td></td>
<td>16804</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Arctic</td>
<td>Ocean</td>
<td></td>
<td>510570</td>
<td>.</td>
<td>17880</td>
<td>.</td>
</tr>
<tr>
<td>Atlantic</td>
<td>Ocean</td>
<td></td>
<td>33420000</td>
<td>.</td>
<td>28374</td>
<td>.</td>
</tr>
</tbody>
</table>

Solution

To create a separate data set for each type of feature, you could go through the data set manually to determine all the unique values of Type, and then write a separate DATA step for each type (or a single DATA step with multiple OUTPUT statements). This approach is labor-intensive, error-prone, and impractical for large data sets. The following PROC SQL code counts the unique values of Type and puts each value in a separate macro variable. The SAS macro that follows the PROC SQL code uses these macro variables to create a SAS data set for each value. You do not need to know beforehand how many unique values there are or what the values are.

```sql
proc sql noprint;
    select count(distinct type)
    into :n
    from sql.features;
    select distinct type
```
into :type1 - :type%left(&n) from sql.features;
quit;

%macro makeds;
%do i=1 %to &n;
  data &&type&i (drop=type);
  set sql.features;
  if type="&&type&i";
  run;
%end;
%mend makeds;
%makeds;

Log 6.1  SAS Log After Creating a Separate Data Set for Each Unique Value of a Column

240  proc sql noprint;
241     select count(distinct type) into :n from sql.features;
242     select distinct type into :type1 - :type%left(&n) from sql.features;
243  quit;
244
245  %macro makeds;
246     %do i=1 %to &n;
247        data &&type&i (drop=type);
248           set sql.features;
249           if type="&&type&i";
250        run;
251     %end;
252  %mend makeds;
253  %makeds;
254
255  NOTE: PROCEDURE SQL used (Total process time):
256     real time         0.04 seconds
257     cpu time         0.03 seconds

258  NOTE: There were 74 observations read from the data set SQL.FEATURES.
259  NOTE: The data set WORK.DESERT has 7 observations and 6 variables.
260  NOTE: DATA statement used (Total process time):
261      real time         1.14 seconds
262      cpu time         0.41 seconds

263  NOTE: There were 74 observations read from the data set SQL.FEATURES.
264  NOTE: The data set WORK.ISLAND has 6 observations and 6 variables.
265  NOTE: DATA statement used (Total process time):
266      real time         0.02 seconds
267      cpu time         0.00 seconds

268  NOTE: There were 74 observations read from the data set SQL.FEATURES.
269  NOTE: The data set WORK.LAKE has 10 observations and 6 variables.
270  NOTE: DATA statement used (Total process time):
271      real time         0.01 seconds
272      cpu time         0.01 seconds

273  NOTE: There were 74 observations read from the data set SQL.FEATURES.
274  NOTE: The data set WORK.MOUNTAIN has 18 observations and 6 variables.
275  NOTE: DATA statement used (Total process time):
276      real time         0.02 seconds
277      cpu time         0.01 seconds
How It Works

This solution uses the INTO clause to store values in macro variables. The first SELECT statement counts the unique variables and stores the result in macro variable N. The second SELECT statement creates a range of macro variables, one for each unique value, and stores each unique value in one of the macro variables. Note the use of the %LEFT function, which trims leading blanks from the value of the N macro variable.

The MAKEDS macro uses all the macro variables that were created in the PROC SQL step. The macro uses a %DO loop to execute a DATA step for each unique value, writing rows that contain a given value of Type to a SAS data set of the same name. The Type variable is dropped from the output data sets.

For more information about SAS macros, see SAS Macro Language: Reference.

Using PROC SQL Tables in Other SAS Procedures

Problem

You want to show the average high temperatures in degrees Celsius for European countries on a map.

Background Information

The Sql.WorldTemps table has average high and low temperatures for various cities around the world.

```
proc sql outobs=10;
title 'WorldTemps';
```
select City, Country, avgHigh, avgLow
from sql.worldtemps
;

Output 6.24  WorldTemps (Partial Output)

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>AvgHigh</th>
<th>AvgLow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>90</td>
<td>45</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>Netherlands</td>
<td>70</td>
<td>33</td>
</tr>
<tr>
<td>Athens</td>
<td>Greece</td>
<td>89</td>
<td>41</td>
</tr>
<tr>
<td>Auckland</td>
<td>New Zealand</td>
<td>75</td>
<td>44</td>
</tr>
<tr>
<td>Bangkok</td>
<td>Thailand</td>
<td>95</td>
<td>69</td>
</tr>
<tr>
<td>Beijing</td>
<td>China</td>
<td>86</td>
<td>17</td>
</tr>
<tr>
<td>Belgrade</td>
<td>Yugoslavia</td>
<td>80</td>
<td>29</td>
</tr>
<tr>
<td>Berlin</td>
<td>Germany</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>Bogota</td>
<td>Colombia</td>
<td>69</td>
<td>43</td>
</tr>
<tr>
<td>Bombay</td>
<td>India</td>
<td>90</td>
<td>68</td>
</tr>
</tbody>
</table>

Solution

Use the following PROC SQL and PROC GMAP code to produce the map. You must license SAS/GRAPH software to use PROC GMAP.

```sas
options fmtsearch=(sashelp.mapfmts);

proc sql;
create table extremetemps as
select country, round(((mean(avgHigh)-32)/1.8) as High,
input(put(country,$glcsmn.), best.) as ID
from sql.worldtemps
where calculated id is not missing and country in
(select name from sql.countries where continent='Europe')
group by country;
quit;

proc gmap map=maps.europe data=extremetemps all;
  id id;
  block high / levels=3;
  title 'Average High Temperatures for European Countries';
  title2 'Degrees Celsius'
run;
quit;
```
How It Works

The SAS system option FMTSEARCH= tells SAS to search in the Sashelp.Mapfmts catalog for map-related formats. In the PROC SQL step, a temporary table is created with Country, High, and ID columns. The calculation $\text{round}((\text{mean}(	ext{avgHigh}) - 32) / 1.8)$ does the following:

1. For countries that are represented by more than one city, the mean of the cities' average high temperatures is used for that country.
2. That value is converted from degrees Fahrenheit to degrees Celsius.
3. The result is rounded to the nearest degree.

The PUT function uses the $\text{GLCSMN.}$ format to convert the country name to a country code. The INPUT function converts this country code, which is returned by the PUT function as a character value, into a numeric value that can be understood by the GMAP procedure. See SAS Functions and CALL Routines: Reference for details about the PUT and INPUT functions.

The WHERE clause limits the output to European countries by checking the value of the Country column against the list of European countries that is returned by the in-line view. Also, rows with missing values of ID are eliminated. Missing ID values could be produced if the $\text{GLCSMN.}$ format does not recognize the country name.

The GROUP BY clause is required so that the mean temperature can be calculated for each country rather than for the entire table.
The PROC GMAP step uses the ID variable to identify each country and places a block representing the High value on each country on the map. The ALL option ensures that countries (such as the United Kingdom in this example) that do not have High values are also drawn on the map. In the BLOCK statement, the LEVELS= option specifies how many response levels are used in the graph. For more information about the GMAP procedure, see *SAS/GRAPH: Reference*. 
Part 2

SQL Procedure Reference

Chapter 7
SQL Procedure ......................................................... 215

Chapter 8
SQL Procedure Components ................................. 315
Chapter 7
SQL Procedure

Overview
What Is the SQL Procedure? 216
What Are PROC SQL Tables? 216
What Are Views? 217
SQL Procedure Coding Conventions 217
Threaded Processing Using PROC SQL 218

Syntax: SQL Procedure 218
PROC SQL Statement 221
ALTER TABLE Statement 232
CONNECT Statement 236
CREATE INDEX Statement 237
CREATE TABLE Statement 238
CREATE VIEW Statement 243
DELETE Statement 245
DESCRIBE Statement 247
DISCONNECT Statement 248
DROP Statement 248
EXECUTE Statement 249
INSERT Statement 250
RESET Statement 252
SELECT Statement 252
UPDATE Statement 267
VALIDATE Statement 268

Examples: SQL Procedure 269
Example 1: Creating a Table and Inserting Data into It 269
Example 2: Creating a Table from a Query's Result 271
Example 3: Updating Data in a PROC SQL Table 273
Example 4: Joining Two Tables 276
Example 5: Using an Inner Join 278
Example 6: Combining Two Tables 280
Example 7: Reporting from DICTIONARY Tables 282
Example 8: Performing an Outer Join 284
Example 9: Creating a View from a Query's Result 290
Example 10: Joining Three Tables 293
Example 11: Querying an In-Line View 297
Example 12: Retrieving Values with the SOUNDS-LIKE Operator 299
Example 13: Joining Two Tables and Calculating a New Value 301
Example 14: Producing All the Possible Combinations of the Values in a Column 304
Example 15: Matching Case Rows and Control Rows 310
Overview

What Is the SQL Procedure?

The SQL procedure implements Structured Query Language (SQL) for SAS. SQL is a standardized, widely used language that retrieves data from and updates data in tables and the views that are based on those tables.

The SAS SQL procedure enables you to do the following:

• retrieve and manipulate data that is stored in tables or views.
• create tables, views, and indexes on columns in tables.
• create SAS macro variables that contain values from rows in a query’s result.
• add or modify the data values in a table’s columns or insert and delete rows. You can also modify the table itself by adding, modifying, or dropping columns.
• send DBMS-specific SQL statements to a database management system (DBMS) and retrieve DBMS data.

The following figure summarizes the variety of source material that you can use with PROC SQL and what the procedure can produce.

Figure 7.1 PROC SQL Input and Output

What Are PROC SQL Tables?

A PROC SQL table is synonymous with a SAS data file and has a member type of DATA. You can use PROC SQL tables as input into DATA steps and procedures.

You create PROC SQL tables from SAS data files, from SAS views, or from DBMS tables by using PROC SQL’s pass-through facility or the SAS/ACCESS LIBNAME statement. The pass-through facility is described in “Connecting to a DBMS By Using the SQL Procedure Pass-Through Facility” on page 171. The SAS/ACCESS LIBNAME statement is described in “Connecting to a DBMS By Using the LIBNAME Statement” on page 168.

In PROC SQL terminology, a row in a table is the same as an observation in a SAS data file. A column is the same as a variable.
What Are Views?

A SAS view defines a virtual data set that is named and stored for later use. A view contains no data but describes or defines data that is stored elsewhere. There are three types of SAS views:

- PROC SQL views
- SAS/ACCESS views
- DATA step views.

You can refer to views in queries as if they were tables. The view derives its data from the tables or views that are listed in its FROM clause. The data that is accessed by a view is a subset or superset of the data that is in its underlying tables or views.

A PROC SQL view is a SAS data set of type VIEW that is created by PROC SQL. A PROC SQL view contains no data. It is a stored query expression that reads data values from its underlying files, which can include SAS data files, SAS/ACCESS views, DATA step views, other PROC SQL views, or DBMS data. When executed, a PROC SQL view's output can be a subset or superset of one or more underlying files.

SAS/ACCESS views and DATA step views are similar to PROC SQL views in that they are both stored programs of member type VIEW. SAS/ACCESS views describe data in DBMS tables from other software vendors. DATA step views are stored DATA step programs.

Note: Starting in SAS 9, PROC SQL views, the pass-through facility, and the SAS/ACCESS LIBNAME statement are the preferred ways to access relational DBMS data; SAS/ACCESS views are no longer recommended. You can convert existing SAS/ACCESS views to PROC SQL views by using the CV2VIEW procedure. For more information, see “CV2VIEW Procedure” in SAS/ACCESS for Relational Databases: Reference.

You can update data through a PROC SQL or SAS/ACCESS view with certain restrictions. See “Updating PROC SQL and SAS/ACCESS Views” on page 173.

You can use all types of views as input to DATA steps and procedures.

Note: In this chapter, the term view collectively refers to PROC SQL views, DATA step views, and SAS/ACCESS views, unless otherwise noted.

Note: When the contents of an SQL view are processed (by a DATA step or a procedure), the referenced data set must be opened to retrieve information about the variables that is not stored in the view. If that data set is associated with a libref that is not defined in the current SAS code, then an error will result. You can avoid this error by specifying a USING clause in the CREATE VIEW statement. See “CREATE VIEW Statement” on page 243 for details.

Note: When you process PROC SQL views between a client and a server, getting the correct results depends on the compatibility between the client and server architecture. For more information, see “Access a SAS View” in SAS/CONNECT User’s Guide.

SQL Procedure Coding Conventions

Because PROC SQL implements Structured Query Language, it works somewhat differently from other Base SAS procedures, as described here:
When a PROC SQL statement is executed, PROC SQL continues to run until a QUIT statement, a DATA step, or another SAS procedure is executed. Therefore, you do not need to repeat the PROC SQL statement with each SQL statement. You need to repeat the PROC SQL statement only if you execute a QUIT statement, a DATA step, or another SAS procedure between SQL statements.

SQL procedure statements are divided into clauses. For example, the most basic SELECT statement contains the SELECT and FROM clauses. Items within clauses are separated with commas in SQL, not with blanks as in other SAS code. For example, if you list three columns in the SELECT clause, then the columns are separated with commas.

The SELECT statement, which is used to retrieve data, also automatically writes the output data to the Output window unless you specify the NOPRINT option in the PROC SQL statement. Therefore, you can display your output or send it to a list file without specifying the PRINT procedure.

The ORDER BY clause sorts data by columns. In addition, tables do not need to be presorted by a variable for use with PROC SQL. Therefore, you do not need to use the SORT procedure with your PROC SQL programs.

A PROC SQL statement runs when you submit it; you do not have to specify a RUN statement. If you follow a PROC SQL statement with a RUN statement, then SAS ignores the RUN statement and submits the statements as usual.

**Threaded Processing Using PROC SQL**

The THREADS option enables or disables parallel processing in PROC SQL. Threaded processing achieves a degree of parallelism in a processing operation. This parallelism is intended to reduce the real time to completion for a processing operation and therefore limit the cost of additional CPU resources. For more information, see “Support for Parallel Processing” in *SAS Language Reference: Concepts*.

The value of the SAS system option CPUCOUNT= affects the performance of the threaded sort. CPUCOUNT= suggests how many system CPUs are available for use by the threaded procedures.

For more information about the THREADS option, see “THREADS | NOTHREADS” on page 230.

For more information, see “CPUCOUNT= System Option” in *SAS System Options: Reference*.

**Syntax: SQL Procedure**

**Restriction:** This procedure is not supported on the CAS server.

**Tips:** You can use any global statements. For more information, see “Fundamental Concepts for Using Base SAS Procedures” in *Base SAS Procedures Guide*.

You can use data set options anytime a table name or view name is specified. For more information, see “Using SAS Data Set Options with PROC SQL” on page 156.

Regular type indicates the name of a component that is described in Chapter 8, “SQL Procedure Components,” on page 315. view-name indicates a SAS view of any type.
PROC SQL <option(s)>;
    ALTER TABLE table-name
    <ADD column-definition-1 <, column-definition-2, ...>>
    <ADD CONSTRAINT constraint-name-1 constraint-specification-1
        <, constraint-name-2 constraint-specification-2, ...>>
    <ADD constraint-specification-1 <, constraint-specification-2, ...>>
    <DROP column-1 <, column-2, ...>
    <DROP CONSTRAINT constraint-name-1 <, constraint-name-2, ...>>
    <DROP FOREIGN KEY constraint-name>
    <DROP PRIMARY KEY>
    <MODIFY column-definition-1 <, column-definition-2, ...>>

    CREATE <UNIQUE> INDEX index-name
        ON table-name (column-1 <, column-2, ...);
    CREATE TABLE table-name
        (column-specification-1 <, column-specification-2 | constraint-specification-1, ...
            >);
    CREATE TABLE table-name LIKE table-name-2;
    CREATE TABLE table-name AS query-expression
        <ORDER BY order-by-item-1 <, order-by-item-2, ...>>;
    CREATE VIEW proc-sql-view <(column-name-list)> AS query-expression
        <ORDER BY order-by-item-1 <, order-by-item-2, ...>>
        <USING libname-clause-1 <, libname-clause-2, ...> ;>
    DELETE FROM table-name | sas/access-view | proc-sql-view
        <AS alias>
        <WHERE sql-expression>;
    DESCRIBE TABLE table-name-1 <, table-name-2, ...>;
    DESCRIBE VIEW proc-sql-view-1 <, proc-sql-view-2, ...>;
    DESCRIBE TABLE CONSTRAINTS table-name-1 <, table-name-2, ...>;
    DROP INDEX index-name-1 <, index-name-2, ...> FROM table-name;
    DROP TABLE table-name-1 <, table-name-2, ...>;
    DROP VIEW view-name-1 <, view-name-2, ...>;
    INSERT INTO table-name | sas/access-view | proc-sql-view
        <(column-1 <, column-2, ...)>;
        SET column1=sql-expression-1 <, column-2=sql-expression-2, ...>
        <column1=sql-expression-1 <, column-2=sql-expression-2, ...> ...>;
    INSERT INTO table-name | sas/access-view | proc-sql-view
        <(column-1 <, column-2, ...)>;
        VALUES (value-1 <, value-2, ...)
        <VALUES (value-1 <, value-2, ...) ...>;
    INSERT INTO table-name | sas/access-view | proc-sql-view
        <(column-1 <, column-2, ...)> query-expression;
    RESET <option(s)>;
To connect to a DBMS and send it a DBMS-specific nonquery SQL statement, use this form:

**PROC SQL:**

```
CONNECT TO dbms-name <AS alias>
<connect-statement-argument-1=value-1
<connect-statement-argument-2=value-2 ...>>
<database-connection-argument-1=value-1
<database-connection-argument-2=value-2 ...> >;
EXECUTE (dbms-SQL-statement)
BY dbms-name | alias;
<DISCONNECT FROM dbms-name | alias >;
<QUIT >;
```

To connect to a DBMS and query the DBMS data, use this form:

**PROC SQL:**

```
CONNECT TO dbms-name <AS alias>
<connect-statement-argument-1=value-1
<connect-statement-argument-2=value-2 ...>>
<database-connection-argument-1=value-1
<database-connection-argument-2=value-2 ...> >;
SELECT column-list
FROM CONNECTION TO dbms-name | alias (dbms-query)
optional PROC SQL clauses;
<DISCONNECT FROM dbms-name | alias >;
<QUIT >;
```
### PROC SQL Statement

**Statement**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Task</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE INDEX</td>
<td>Create an index on a column</td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Create a PROC SQL table</td>
<td>Ex. 2</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>Create a PROC SQL view</td>
<td>Ex. 9</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete rows</td>
<td>Ex. 6</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Display a definition of a table or view</td>
<td>Ex. 7</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>Terminate the connection with a DBMS</td>
<td></td>
</tr>
<tr>
<td>DROP</td>
<td>Delete tables, views, or indexes</td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Send a DBMS-specific nonquery SQL statement to a DBMS</td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>Add rows</td>
<td>Ex. 1</td>
</tr>
<tr>
<td>RESET</td>
<td>Reset options that affect the procedure environment without restarting the procedure</td>
<td>Ex. 6</td>
</tr>
<tr>
<td>SELECT</td>
<td>Select and execute rows</td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>Modify values</td>
<td>Ex. 3</td>
</tr>
<tr>
<td>VALIDATE</td>
<td>Verify the accuracy of your query</td>
<td></td>
</tr>
</tbody>
</table>

**Restriction:** This procedure is not supported on the CAS server.

### Syntax

PROC SQL <option(s)>;

### Summary of Optional Arguments

**Control execution**

- CONSTDATETIME | NOCONSTDATETIME
- DQUOTE=ANSI | SAS
- ERRORSTOP | NOERRORSTOP
- EXEC | NOEXEC
- EXITCODE
- INOBS=<n>
- IPASSTHRU | NOIPASSTHRU
- LOOPS=<n>
NOCONSTDATETIME
NOERRORSTOP
NOEXEC
NOIPASSTHRU
NOPROMPT
NOREMERGE
NOSTIMER
NOTHREADS
OUTOBS=n
PROMPT | NOPROMPT
REDUCEPUT=ALL | NONE | DBMS | BASE
REDUCEPUTOBS=n
REDUCEPUTVALUES=n
REMERGE | NOREMERGE
STIMER | NOSTIMER
STOPONTRUNC
THREADS | NOTHREADS
UNDO_POLICY=NONE | OPTIONAL | REQUIRED

Control output

DICTDIAG | NODICTDIAG
DOUBLE | NODOUBLE
FEEDBACK | NOFEEDBACK
FLOW<=n <m>> | NOFLOW
NODICTDIAG
NODOUBLE
NOFEEDBACK
NOFLOW
NONUMBER
NOPRINT
NOSORTMSG
NOWARNRECURS
NUMBER | NONUMBER
PRINT | NOPRINT
SORTMSG | NOSORTMSG
SORTSEQ=sort-table | LINGUISTIC
UBUFSIZE=n | nK | nM | nG
WARNRECURS | NOWARNRECURS

Optional Arguments

CONSTDATETIME | NOCONSTDATETIME

specifies whether the SQL procedure replaces references to the DATE, TIME, DATETIME, and TODAY functions in a query with their equivalent constant values before the query executes. Computing these values once ensures consistency of results when the functions are used multiple times in a query or when the query executes the functions close to a date or time boundary.
When the NOCONSTDATETIME option is set, PROC SQL evaluates these functions in a query each time it processes an observation.

Default  

Interaction  

If both the CONSTDATETIME option and the REDUCEPUT= option on page 227 are specified, then PROC SQL replaces the DATE, TIME, DATETIME, and TODAY functions with their respective values in order to determine the PUT function value before the query executes.

Tip  

Alternatively, you can set the SQLCONSTDATETIME system option. The value that is specified in the SQLCONSTDATETIME system option is in effect for all SQL procedure statements, unless the PROC SQL CONSTDATETIME option is set. The value of the CONSTDATETIME option takes precedence over the SQLCONSTDATETIME system option. The RESET statement can also be used to set or reset the CONSTDATETIME option. However, changing the value of the CONSTDATETIME option does not change the value of the SQLCONSTDATETIME system option. For more information, see the “SQLCONSTDATETIME System Option” on page 373.

DICTDIAG | NODICTDIAG  

specifies whether to create a Diagnostic column in the report. The Diagnostic column in the report contains a message for each DICTIONARY table that failed to open.

Default  

DOUBLE | NODOUBLE  

specifies whether to double-space the report.

Default  

Example  

“Example 6: Combining Two Tables” on page 280

DQUOTE=ANSI | SAS  

specifies whether PROC SQL treats values within double quotation marks (" ") as variables or strings. With DQUOTE=ANSI, PROC SQL treats a quoted value as a variable. This feature enables you to use the following as table names, column names, or aliases:

• reserved words such as AS, JOIN, GROUP, and so on
• DBMS names and other names that are not normally permissible in SAS.

The quoted value can contain any character.

With DQUOTE=SAS, values within double quotation marks are treated as strings.

Default  

ERRORSTOP | NOERRORSTOP  

specifies whether PROC SQL stops executing if it encounters an error. In a batch or noninteractive session, ERRORSTOP instructs PROC SQL to stop executing the statements but to continue checking the syntax after it has encountered an error.

NOERRORSTOP instructs PROC SQL to execute the statements and to continue checking the syntax after an error occurs.
<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Interaction</th>
<th>Tips</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC</td>
<td>NOERRORSTOP in an interactive SAS session; ERRORSTOP in a batch or noninteractive session</td>
<td>This option is useful only when the EXEC option is in effect.</td>
<td>ERRORSTOP has an effect only when SAS is running in the batch or noninteractive execution mode.</td>
<td>ERRORSTOP on page 223</td>
</tr>
<tr>
<td>NOEXEC</td>
<td>specifies whether a statement should be executed after its syntax is checked for accuracy.</td>
<td></td>
<td>NOERRORSTOP is useful if you want a batch job to continue executing SQL procedure statements after an error is encountered.</td>
<td></td>
</tr>
<tr>
<td>EXITCODE</td>
<td>specifies whether PROC SQL clears an error code for any SQL statement. Error codes are assigned to the SQLEXITCODE macro variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>EXEC</td>
<td>NOEXEC is useful if you want to check the syntax of your SQL statements without executing the statements.</td>
<td>ERRORSTOP on page 223</td>
<td></td>
</tr>
<tr>
<td>Tip</td>
<td>The exit code can be reset to the default value between PROC SQL statements with the “RESET Statement” on page 252.</td>
<td>“Using the PROC SQL Automatic Macro Variables” on page 162</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FEEDBACK</td>
<td>specifies whether PROC SQL displays, in the SAS log, PROC SQL statements after view references are expanded or certain other transformations of the statement are made.</td>
<td></td>
<td>This option has the following effects:</td>
<td></td>
</tr>
<tr>
<td>NOFEEDBACK</td>
<td></td>
<td></td>
<td>• Any asterisk (for example, <code>SELECT *</code>) is expanded into the list of qualified columns that it represents.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Any PROC SQL view is expanded into the underlying query.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Macro variables are resolved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Parentheses are shown around all expressions to further indicate their order of evaluation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Comments are removed.</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>NOFEEDBACK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOW&lt;=n &lt;m&gt;&gt;</td>
<td>specifies that character columns longer than n are flowed to multiple lines. PROC SQL sets the column width at n and specifies that character columns longer than n are flowed to multiple lines. When you specify FLOW=n m, PROC SQL floats the width of the columns between these limits to achieve a balanced layout. Specifying FLOW without arguments is equivalent to specifying FLOW=12 200.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INOBS=n restricts the number of rows (observations) that PROC SQL retrieves from any single source.

Tip The INOBS= option is useful for debugging queries on large tables.

IPASSTHRU | NOIPASSTHRU specifies whether implicit pass through is enabled or disabled.

Implicit pass through is enabled when PROC SQL is invoked. You can disable it for a query or series of queries. The primary reasons that you might want to disable implicit pass through are as follows:

• DBMSs use SQL2 semantics for NULL values, which behave somewhat differently than SAS missing values.
• PROC SQL might do a better job of query optimization.

LOOPS=n restricts PROC SQL to n iterations through its inner loop. You use the number of iterations reported in the SQLOOPS macro variable (after each SQL statement is executed) to discover the number of loops. Set a limit to prevent queries from consuming excessive computer resources. For example, joining three large tables without meeting the join-matching conditions could create a huge internal table that would be inefficient to execute.

NOCONSTDATETIME

See “CONSTDATETIME | NOCONSTDATETIME” on page 222

NODICTDIAG

See “DICTDIAG | NODICTDIAG” on page 223

NODOUBLE

See “DOUBLE | NODOUBLE” on page 223

NOERRORSTOP

See “ERRORSTOP | NOERRORSTOP” on page 223

NOEXEC

See “EXEC | NOEXEC” on page 224

NOFEEDBACK

See “FEEDBACK | NOFEEDBACK” on page 224

NOFLOW

See “FLOW<=n<m> | NOFLOW” on page 224
NOIPASSTHRU
See “IPASSTHRU | NOIPASSTHRU” on page 225

NONUMBER
See “NUMBER | NONUMBER” on page 226

NOPRINT
See “PRINT | NOPRINT” on page 226

NOPROMPT
See “PROMPT | NOPROMPT” on page 227

NOREMERGE
See “REMERGE | NOREMERGE” on page 229

NOSORTMSG
See “SORTMSG | NOSORTMSG” on page 229

NOSTIMER
See “STIMER | NOSTIMER” on page 230

NOTHREADS
See “THREADS | NOTHREADS” on page 230

NOWARNRECURS
See “WARNRECURS | NOWARNRECURS” on page 231

NUMBER | NONUMBER
specifies whether the SELECT statement includes a column called ROW, which is the row (or observation) number of the data as the rows are retrieved.

Default NONUMBER

Example “Example 4: Joining Two Tables” on page 276

OUTOBS=n
restricts the number of rows (observations) in the output. For example, if you specify OUTOBS=10 and insert values into a table using a query expression, then the SQL procedure inserts a maximum of 10 rows. Likewise, OUTOBS=10 limits the output to 10 rows.

PRINT | NOPRINT
specifies whether the output from a SELECT statement is printed.

Default PRINT

Interaction NOPRINT affects the value of the SQLOBS automatic macro variable. For more information, see “Using the PROC SQL Automatic Macro Variables” on page 162.

Tip NOPRINT is useful when you are selecting values from a table into macro variables and do not want anything to be displayed.
PROMPT | NOPROMPT
modifies the effect of the INOBS=, OUTOBS=, and LOOPS= options. If you specify the PROMPT option and reach the limit specified by INOBS=, OUTOBS=, or LOOPS=, then PROC SQL prompts you to stop or continue. The prompting repeats if the same limit is reached again.

Default NOPROMPT

REDUCEPUT=ALL | NONE | DBMS | BASE
specifies the engine type to use to optimize a PUT function in a query. The PUT function is replaced with a logically equivalent expression. The engine type can be one of the following values:

ALL
specifies to consider the optimization of all PUT functions, regardless of the engine that is used by the query to access the data.

NONE
specifies to not optimize any PUT function.

DBMS
specifies to consider the optimization of all PUT functions in a query performed by a SAS/ACCESS engine.

Requirement The first argument to the PUT function must be a variable that is obtained by a table. The table must be accessed using a SAS/ACCESS engine.

BASE
specifies to consider the optimization of all PUT functions in a query performed by a SAS/ACCESS engine or a Base SAS engine.

Default DBMS

Interactions If both the REDUCEPUT= option and the CONSTDATETIME option are specified, then PROC SQL replaces the DATE, TIME, DATETIME, and TODAY functions with their respective values to determine the PUT function value before the query executes.

If the query also contains a WHERE or HAVING clause, then the evaluation of the WHERE or HAVING clause is simplified.

Tip Alternatively, you can set the SQLREDUCEPUT= system option. The value that is specified in the SQLREDUCEPUT= system option is in effect for all SQL procedure statements, unless the REDUCEPUT= option is set. The value of the REDUCEPUT= option takes precedence over the SQLREDUCEPUT= system option. The RESET statement can also be used to set or reset the REDUCEPUT= option. However, changing the value of the REDUCEPUT= option does not change the value of the SQLREDUCEPUT= system option. For more information, see the “SQLREDUCEPUT= System Option” on page 379.

REDUCEPUTOBS=n
when the REDUCEPUT= option is set to DBMS, BASE, or ALL, specifies the minimum number of observations that must be in a table for PROC SQL to consider optimizing the PUT function in a query.
Default 0, which indicates that there is no minimum number of observations in a table for PROC SQL to optimize the PUT function.

Range 0–2^63–1, or approximately 9.2 quintillion

Requirement n must be an integer

Interaction The REDUCEPUTOBS= option works only for DBMSs that record the number of observations in a table. If your DBMS does not record the number of observations, but you create row counts on your table, then the REDUCEPUTOBS= option will work.

Tip Alternatively, you can set the SQLREDUCEPUTOBS= system option. The value that is specified in the SQLREDUCEPUTOBS= system option is in effect for all SQL procedure statements, unless the REDUCEPUTOBS= option is set. The value of the REDUCEPUTOBS= option takes precedence over the SQLREDUCEPUTOBS= system option. The RESET statement can also be used to set or reset the REDUCEPUTOBS= option. However, changing the value of the REDUCEPUTOBS= option does not change the value of the SQLREDUCEPUTOBS= system option. For more information, see the “SQLREDUCEPUTOBS= System Option” on page 380.

REDUCEPUTVALUES=n

when the REDUCEPUT= option is set to DBMS, BASE, or ALL, specifies the maximum number of SAS format values that can exist in a PUT function expression for PROC SQL to consider optimizing the PUT function in a query.

Default 100

Range 100–3,000

Requirement n must be an integer

Interaction If the number of SAS format values in a PUT function expression is greater than this value, then PROC SQL does not optimize the PUT function.

Tips Alternatively, you can set the SQLREDUCEPUTVALUES= system option. The value that is specified in the SQLREDUCEPUTVALUES= system option is in effect for all SQL procedure statements, unless the REDUCEPUTVALUES= option is set. The value of the REDUCEPUTVALUES= option takes precedence over the SQLREDUCEPUTVALUES= system option. The RESET statement can also be used to set or reset the REDUCEPUTVALUES= option. However, changing the value of the REDUCEPUTVALUES= option does not change the value of the SQLREDUCEPUTVALUES= system option. For more information, see “SQLREDUCEPUTVALUES= System Option” on page 381.

The value for REDUCEPUTVALUES= is used for each individual optimization. For example, if you have a PUT function in a WHERE clause, and another PUT function in a SELECT statement, and both have user-defined formats with contained values, the value of
REDUCEPUTVALUES= is applied separately for the clause and the statement.

**REMERGE | NOREMERGE**

specifies whether PROC SQL can process queries that use remerging of data. The remerge feature of PROC SQL makes two passes through a table, using data in the second pass that was created in the first pass, in order to complete a query. When the NOREMERGE system option is set, PROC SQL cannot process remerging of data. If remerging is attempted when the NOREMERGE option is set, then an error is written to the SAS log.

Default: REMERGE

Tip: Alternatively, you can set the SQLREMERGE system option. The value that is specified in the SQLREMERGE system option is in effect for all SQL procedure statements, unless the PROC SQL REMERGE option is set. The value of the REMERGE option takes precedence over the SQLREMERGE system option. The RESET statement can also be used to set or reset the REMERGE option. However, changing the value of the REMERGE option does not change the value of the SQLREMERGE system option. For more information, see “SQLREMERGE System Option” on page 383.

See: “Remerging Data” on page 363

**SORTMSG | NOSORTMSG**

for certain operations, such as ORDER BY, that can sort tables internally using PROC SORT, specifies whether information about the sort can be requested from PROC SORT and displayed in the log.

Default: NOSORTMSG

**SORTSEQ=sort-table | LINGUISTIC**

specifies the collating sequence to use when a query contains an ORDER BY clause. Use this option only if you want a collating sequence other than your system or installation’s default collating sequence.

- **sort-table**: specifies a translation table that you created with PROC TRANTAB.
- **LINGUISTIC**: specifies that the collating sequence is determined from the session locale and that default collation options are used. If LINGUISTIC is specified for the SORTSEQ system option, then PROC SQL honors the setting. The setting of the PROC SQL SORTSEQ option overrides the setting of the SORTSEQ system option.

**CAUTION:**

Do not use the PROC SQL SORTSEQ=LINGUISTIC option or the SORTSEQ=LINGUISTIC system option when a SORTKEY function is used in an ORDER BY clause. It produces unintended ordering.


STIMER | NOSTIMER
specifies whether PROC SQL writes timing information to the SAS log for each
statement, rather than as a cumulative value for the entire procedure. For this option
to work, you must also specify the SAS system option STIMER. Some operating
environments require that you specify this system option when you invoke SAS. If
you use the system option alone, then you receive timing information for the entire
SQL procedure, not on a statement-by-statement basis.

Default  NOSTIMER

STOPONTRUNC
specifies to not insert or update a row that contains data larger than the column when
a truncation error occurs. This applies only when using the SET clause in an INSERT
or UPDATE statement.

THREADS | NOTHREADS
overrides the SAS system option THREADS | NOTHREADS for a particular
invocation of PROC SQL unless the system option is restricted. (See Restriction.)
THREADS | NOTHREADS can also be specified in a RESET statement for use in
particular queries. When THREADS is specified, PROC SQL uses parallel
processing in order to increase the performance of sorting operations that involve
large amounts of data. For more information about parallel processing, see SAS

Default value of SAS system option THREADS | NOTHREADS.

Restriction Your site administrator can create a restricted options table. A restricted
options table specifies SAS system option values that are established at
start-up and cannot be overridden. If the THREADS | NOTHREADS
system option is listed in the restricted options table, then any attempt
to set it is ignored and a warning message is written to the SAS log.

Interaction When THREADS | NOTHREADS has been specified in a PROC SQL
statement or a RESET statement, there is no way to reset the option to
its default (that is, the value of the SAS system option THREADS |
NOTHREADS) for that invocation of PROC SQL.

See “THREADS System Option” in SAS System Options: Reference

UNDO_POLICY=NONE | OPTIONAL | REQUIRED
specifies how PROC SQL handles updated data if errors occur while you are
updating data. You can use UNDO_POLICY= to control whether your changes are
permanent.

NONE
keeps any updates or inserts.

OPTIONAL
reverses any updates or inserts that it can reverse reliably.

REQUIRED
reverses all inserts or updates that have been done to the point of the error. In
some cases, the UNDO operation cannot be done reliably. For example, when a
program uses a SAS/ACCESS view, it might not be able to reverse the effects of
the INSERT and UPDATE statements without reversing the effects of other
changes at the same time. In that case, PROC SQL issues an error message and
does not execute the statement. Also, when a SAS data set is accessed through a
SAS/SHARE server and is opened with the data set option
CNTLLEV=RECORD, you cannot reliably reverse your changes.
This option can enable other users to update newly inserted rows. If an error occurs during the insert, then PROC SQL can delete a record that another user updated. In that case, the statement is not executed, and an error message is issued.

**Default**  REQUIRED

**Tips** If you are updating a data set using the SPD Engine, then you can significantly improve processing performance by setting UNDO_POLICY=NONE. However, ensure that NONE is an appropriate setting for your application.

Alternatively, you can set the SQLUNDOPOLICY system option. The value that is specified in the SQLUNDOPOLICY= system option is in effect for all SQL procedure statements, unless the PROC SQL UNDO_POLICY= option is set. The value of the UNDO_POLICY= option takes precedence over the SQLUNDOPOLICY= system option. The RESET statement can also be used to set or reset the UNDO_POLICY= option. However, changing the value of the UNDO_POLICY= option does not change the value of the SQLUNDOPOLICY= system option. After the procedure completes, it reverts to the value of the SQLUNDOPOLICY= system option. For more information, see the “SQLUNDOPOLICY= System Option” on page 384.

**UBUFSIZE=n | nK | nM | nG**

specifies the internal transient buffer page size for the PROC SQL paged memory subsystem. PROC SQL uses this subsystem to help implement operations such as joins, aggregations, and intersections. The output is in multiples of 1 (bytes), 1024 (kilobytes), 1,048,576 (megabytes), or 1,073,741,824 (gigabytes). For example, a value of 65536 specifies a page size of 65536 bytes, and a value of 64k specifies a page size of 65536 bytes.

UBUFSIZE can also be specified in a RESET statement for use in particular queries.

*Note:* The BUFFERSIZE option that was used prior to SAS 9.4 works the same as the UBUFSIZE option and is still supported. As of SAS 9.4, UBUFSIZE is the preferred option.

**Default** 0, which causes SAS to use the minimum optimal page size for the operating environment.

**WARNRECURS | NOWARNRECURS**

specifies whether a warning displays in the SAS log for recursive references.

NOWARNRECURS specifies to display recursive references in a note, instead of as a warning in the SAS log.

**Default**  WARNRECURS

**Details**

*Note:* Options can be added, removed, or changed between PROC SQL statements with the “RESET Statement” on page 252.
ALTER TABLE Statement

Adds columns to, drops columns from, and changes column attributes in an existing table. Adds, modifies, and drops integrity constraints from an existing table.

Restrictions: You cannot use any type of view in an ALTER TABLE statement.
You cannot use ALTER TABLE on a table that is accessed by an engine that does not support UPDATE processing.
You must use at least one ADD, DROP, or MODIFY clause in the ALTER TABLE statement.

See: “Example 3: Updating Data in a PROC SQL Table” on page 273

Syntax

ALTER TABLE table-name
    <ADD column-definition-1 <, column-definition-2, …>>
    <ADD CONSTRAINT constraint-name-1 constraint-specification-1
                <, constraint-name-2 constraint-specification-2, …>>
    <ADD constraint-specification-1 <, constraint-specification-2, …>>
    <DROP column-1 <, column-2, …>>
    <DROP CONSTRAINT constraint-name-1 <, constraint-name-2, …>>
    <DROP FOREIGN KEY constraint-name>
    <DROP PRIMARY KEY>
    <MODIFY column-definition-1 <, column-definition-2, …>>
;

Required Argument
table-name

- in the ALTER TABLE statement, refers to the name of the table that is to be altered.
- in the REFERENCES clause, refers to the name of table that contains the primary key that is referenced by the foreign key.

Table-name can be a one-level name, a two-level libref.table name, or a physical pathname that is enclosed in single quotation marks.

Optional Arguments

<ADD column-definition-1 <, column-definition-2, …>>
    adds the column or columns that are specified in each column-definition.

See “column-definition” on page 320

<ADD CONSTRAINT constraint-name-1 constraint-specification-1 <, constraint-name-2 constraint-specification-2, …>>
    adds the integrity constraint that is specified in constraint-specification and assigns constraint-name to it.

See “constraint-name” on page 233
constraint-name specifies a name for the constraint that is being specified. The name must be a valid SAS name.

Note: The names PRIMARY, FOREIGN, MESSAGE, UNIQUE, DISTINCT, CHECK, and NOT cannot be used as values for constraint-name.

collection-specification is a constraint definition in this form:

constraint <MESSAGE='message-string' <MSGTYPE=message-type>>

constraint is one of the following integrity constraints:

CHECK (WHERE-clause)
specifies that all rows in table-name satisfy the WHERE-clause. WHERE-clauses specify a SAS WHERE clause. Do not include the WHERE keyword in the WHERE clause.

DISTINCT (column-1 <, column-2, ...>)
specifies that the values of each column must be unique. This constraint is identical to UNIQUE.

FOREIGN KEY (column-1 <, column-2, ...>) REFERENCES table-name <ON DELETE referential-action> <ON UPDATE referential-action>
specifies a foreign key, that is, a set of columns whose values are linked to the values of the primary key variable in another table (the table-name that is specified for REFERENCES). The referential-actions are performed when the values of a primary key column that is referenced by the foreign key are updated or deleted.

column names a column in table-name.

referential-action can be the following actions:

CASCADE allows primary key data values to be updated, and updates matching values in the foreign key to the same values. This referential action is currently supported for updates only.

RESTRICT prevents the update or deletion of primary key data values if a matching value exists in the foreign key. This referential action is the default.

SET NULL allows primary key data values to be updated, and sets all matching foreign key values to NULL.

Restriction When defining overlapping primary key and foreign key constraints, the variables in a data file are part of both a primary key and a foreign key definition. If you use the exact same variables, then the variables must be defined in a different order. The foreign key’s update and delete referential actions must both be RESTRICT.

See “table-name” on page 232
NOT NULL (column)
specifies that column does not contain a null or missing value, including special missing values.

PRIMARY KEY (column-1 <, column-2, …> )
specifies one or more primary key columns (that is, columns that do not contain missing values and whose values are unique).

Restriction When you are defining overlapping primary key and foreign key constraints, the variables in a data file are part of both the primary key definition and the foreign key definition. If you use the exact same variables, then the variables must be defined in a different order.

Notes If a NOT NULL constraint exists for a variable that is being used to define a new primary key constraint, then the new primary key’s constraint replaces the existing primary key’s constraint.

If an attempt is made to define a NOT NULL constraint for a variable that is already defined for a primary key constraint, then the NOT NULL constraint definition will fail.

UNIQUE (column-1 <, column-2, …> )
specifies that the values of each column must be unique. This constraint is identical to DISTINCT.

message-string
specifies the text of an error message that is written to the log when the integrity constraint is not met. The maximum length of message-string is 250 characters.

message-type
specifies how the error message is displayed in the SAS log when an integrity constraint is not met. message-type can be one of the following values:

NEWLINE
the text that is specified for MESSAGE= is displayed as well as the default error message for that integrity constraint.

USER
only the text that is specified for MESSAGE= is displayed.

<ADD constraint-specification-1 <, constraint-specification-2, …>>
adds the integrity constraint that is specified in constraint-specification and assigns a default name to it. The default constraint name has the form that is shown in the following table:

<table>
<thead>
<tr>
<th>Default Name</th>
<th>Constraint Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NMxxxx</em></td>
<td>Not null</td>
</tr>
<tr>
<td><em>UNxxxx</em></td>
<td>Unique</td>
</tr>
<tr>
<td><em>CKxxxx</em></td>
<td>Check</td>
</tr>
<tr>
<td><em>PKxxxx</em></td>
<td>Primary key</td>
</tr>
<tr>
<td><em>FKxxxx</em></td>
<td>Foreign key</td>
</tr>
</tbody>
</table>
In these default names, xxx is a counter that begins at 0001.

See “constraint-specification” on page 233

<DROP column-1 < , column-2, …> >
deletes each column from the table.

deletes the integrity constraint that is referenced by each constraint-name. To find the name of an integrity constraint, use the DESCRIBE TABLE CONSTRAINTS clause. (See “DESCRIBE Statement” on page 247.)

See “constraint-name” on page 233

<DROP FOREIGN KEY constraint-name>
removes the foreign key constraint that is referenced by constraint-name.

Note: The DROP FOREIGN KEY clause is a DB2 extension.

See “constraint-name” on page 233

<DROP PRIMARY KEY>
removes the primary key constraint from table.

Note: The DROP PRIMARY KEY clause is a DB2 extension.

<MODIFY column-definition-1 < , column-definition-2, …> >
changes one or more attributes of the column that is specified in each column-definition.

See “column-definition” on page 320

Details

Specifying Initial Values of New Columns
When the ALTER TABLE statement adds a column to the table, it initializes the column's values to missing in all rows of the table. Use the UPDATE statement to add values to the new column or columns.

Changing Column Attributes
If a column is already in the table, then you can change the following column attributes by using the MODIFY clause: length, informat, format, and label. The values in a table are either truncated or padded with blanks (if character data) as necessary to meet the specified length attribute.

You cannot change a character column to numeric and vice versa. To change a column’s data type, drop the column and then add it (and its data) again, or use the DATA step.

Note: You cannot change the length of a numeric column with the ALTER TABLE statement. Use the DATA step instead.

Renaming Columns
You cannot use the RENAME= data set option with the ALTER TABLE statement to change a column’s name. However, you can use the RENAME= data set option with the CREATE TABLE or SELECT statement. For more information about the RENAME= data set option, see the section on SAS data set options in SAS Data Set Options: Reference.
Indexes on Altered Columns
When you alter the attributes of a column and an index has been defined for that column, then the values in the altered column keep the index that is defined for them. If you drop a column with the ALTER TABLE statement, then all the indexes (simple and composite) in which the column participates are also dropped. For more information about creating and using indexes, see “CREATE INDEX Statement” on page 237.

Integrity Constraints
Use ALTER TABLE to modify integrity constraints for existing tables. Use the CREATE TABLE statement to attach integrity constraints to new tables. For more information about integrity constraints, see the section on SAS files in SAS Language Reference: Concepts.

CONNECT Statement
Establishes a connection with a DBMS that SAS/ACCESS software supports.

Requirement: SAS/ACCESS software is required. For more information about this statement, see your SAS/ACCESS documentation.

See: “Connecting to a DBMS By Using the SQL Procedure Pass-Through Facility” on page 171

Syntax
CONNECT TO dbms-name <AS alias> <(connect-statement-argument-1=value-1 <connect-statement-argument-2=value-2 … >)> <(database-connection-argument-1=value-1 <database-connection-argument-2=value-2 … >)>;
CONNECT USING libref <AS alias>;

Required Arguments
dbms-name
identifies the DBMS that you want to connect to (for example, ORACLE or DB2).

libref
specifies the libref for which a DBMS connection has already been established through the LIBNAME statement. The libref connection parameters can be reused in the SQL procedure using the CONNECT statement.

Optional Arguments
AS alias
specifies an alias that has 1 to 32 characters. The keyword AS must precede alias. Some DBMSs allow more than one connection. The optional AS clause enables you to name the connections so that you can refer to them later.

connect-statement-argument=value
specifies values for arguments that indicate whether you can make multiple connections, shared or unique connections, and so on, to the database. These arguments are optional, but if they are included, then they must be enclosed in parentheses. For more information about these arguments, see SAS/ACCESS for Relational Databases: Reference.
`database-connection-argument=value`

specifies values for the DBMS-specific arguments that are needed by PROC SQL in order to connect to the DBMS. These arguments are optional for most databases, but if they are included, then they must be enclosed in parentheses. For more information, see the SAS/ACCESS documentation for your DBMS.

### CREATE INDEX Statement

Creates indexes on columns in tables.

**Restriction:** You cannot use CREATE INDEX on a table that is accessed with an engine that does not support UPDATE processing.

**Syntax**

```sql
CREATE <UNIQUE> INDEX index-name
    ON table-name (column-1 <, column-2, ...);
```

**Required Arguments**

- **INDEX index-name**
  names the index that you are creating. If you are creating an index on one column only, then `index-name` must be the same as `column`. If you are creating an index on more than one column, then `index-name` cannot be the same as any column in the table.

- **ON table-name**
  specifies a PROC SQL table.

**Optional Arguments**

- **UNIQUE**
  causes SAS to reject any change to a table that would cause more than one row to have the same index value. Unique indexes guarantee that data in one column, or in a composite group of columns, remains unique for every row in a table. A unique index can be defined for a column that includes NULL or missing values if each row has a unique index value.

- **column**
  specifies a column in `table-name`.

**Details**

### Indexes in PROC SQL

An index stores both the values of a table’s columns and a system of directions that enable access to rows in that table by index value. Defining an index on a column or set of columns enables SAS, under certain circumstances, to locate rows in a table more quickly and efficiently. Indexes enable PROC SQL to execute the following classes of queries more efficiently:

- comparisons against a column that is indexed
- an IN subquery where the column in the inner subquery is indexed
correlated subqueries, where the column being compared with the correlated reference is indexed

- join-queries, where the join-expression is an equals comparison and all the columns in the join-expression are indexed in one of the tables being joined.

SAS maintains indexes for all changes to the table, whether the changes originate from PROC SQL or from some other source. Therefore, if you alter a column's definition or update its values, then the same index continues to be defined for it. However, if an indexed column in a table is dropped, then the index on it is also dropped.

You can create simple or composite indexes. A simple index is created on one column in a table. A simple index must have the same name as that column. A composite index is one index name that is defined for two or more columns. The columns can be specified in any order, and they can have different data types. A composite index name cannot match the name of any column in the table. If you drop a composite index, then the index is dropped for all the columns named in that composite index.

Managing Indexes
You can use the CONTENTS statement in the DATASETS procedure to display a table's index names and the columns for which they are defined. You can also use the DICTIONARY tables INDEXES, TABLES, and COLUMNS to list information about indexes. For more information, see “Accessing SAS System Information By Using DICTIONARY Tables” on page 149.

See the section on SAS files in SAS Language Reference: Concepts for a further description of when to use indexes and how they affect SAS statements that handle BY-group processing.

CREATE TABLE Statement
Creates PROC SQL tables.

See:  “Example 1: Creating a Table and Inserting Data into It” on page 269
     “Example 2: Creating a Table from a Query's Result” on page 271

Syntax
CREATE TABLE table-name
   (column-specification-1 <, column-specification-2 | constraint-specification-1, …>);
CREATE TABLE table-name LIKE table-name-2;
CREATE TABLE table-name AS query-expression
   <ORDER BY order-by-item-1 <, order-by-item-2, …>>;

Required Arguments

- table-name
  - in the CREATE TABLE statement, refers to the name of the table that is to be created. You can use data set options by placing them in parentheses immediately after table-name. For more information, see “Using SAS Data Set Options with PROC SQL” on page 156.
  - in the REFERENCES clause, refers to the name of table that contains the primary key that is referenced by the foreign key.
column-specification

consists of a column definition and, as an option, constraints for the column using this form:

column-definition <column-constraint>

column-definition

See “column-definition” on page 320.

column-constraint

is one of the following:

CHECK (WHERE-clause)
specifies that all rows in table-name satisfy the WHERE-clause. WHERE-clause is a SAS WHERE clause. Do not include the WHERE keyword in the WHERE clause.

DISTINCT

specifies that the values of the column must be unique. This constraint is identical to UNIQUE.

NOT NULL

specifies that the column does not contain a null or missing value, including special missing values.

PRIMARY KEY

specifies that the column is a primary key column, that is, a column that does not contain missing values and whose values are unique.

Restriction

When you are defining overlapping primary key and foreign key constraints, the variables in a data file are part of both the primary key definition and the foreign key definition. If you use the exact same variables, then the variables must be defined in a different order.

Notes

If a NOT NULL constraint exists for a variable that is being used to define a new primary key constraint, then the new primary key’s constraint replaces the existing primary key’s constraint.

If an attempt is made to define a NOT NULL constraint for a variable that is already defined for a primary key constraint, then the NOT NULL constraint definition will fail.

REFERENCES table-name <ON DELETE referential-action> <ON UPDATE referential-action>
specifies that the column is a foreign key, that is, a column whose values are linked to the values of the primary key variable in another table (the table-name that is specified for REFERENCES). The referential-actions are performed when the values of a primary key column that is referenced by the foreign key are updated or deleted.

referential-actions can be the following actions:

CASCADE

allows primary key data values to be updated, and updates matching values in the foreign key to the same values. This referential action is currently supported for updates only.
RESTRICT
occurs only if there are matching foreign key values. This referential action is the default.

SET NULL
sets all matching foreign key values to NULL.

Restriction When you are defining overlapping primary key and foreign key constraints, the variables in a data file are part of both a primary key definition and a foreign key definition. If you use the exact same variables, then the variables must be defined in a different order. The foreign key’s update and delete referential actions must both be RESTRICT.

UNIQUE
specifies that the values of the column must be unique. This constraint is identical to DISTINCT.

Note: If you specify column-constraint, then SAS automatically assigns a name to the constraint. The constraint name has the form the following form, where xxx is a counter that begins at 0001.

<table>
<thead>
<tr>
<th>Default name</th>
<th>Constraint type</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CKxxxx</em></td>
<td>Check</td>
</tr>
<tr>
<td><em>FKxxxx</em></td>
<td>Foreign key</td>
</tr>
<tr>
<td><em>NMxxxx</em></td>
<td>Not Null</td>
</tr>
<tr>
<td><em>PKxxxx</em></td>
<td>Primary key</td>
</tr>
<tr>
<td><em>UNxxxx</em></td>
<td>Unique</td>
</tr>
</tbody>
</table>

**table-name-2**
creates *table-name* with the same column names and column attributes as *table-name2*, but with no rows.

**query-expression**
creates *table-name* from the results of a query.

See “query expression” on page 343

**Optional Arguments**

**constraint-specification**
is a constraint for one or more columns in this form:

```
CONSTRANTr constraint-name constraint <MESSAGE='message-string'
<MSGTYPE=message-type>>
```

**constraint-name**
specifies a name for the constraint that is being specified. The name must be a valid SAS name.

Note: The names PRIMARY, FOREIGN, MESSAGE, UNIQUE, DISTINCT, CHECK, and NOT cannot be used as values for constraint-name.
**constraint**

is one of the following constraint types:

**CHECK WHERE-clause**

specifies that all rows in *table-name* satisfy the *WHERE-clause*. *WHERE-clause* is a SAS WHERE clause. Do not include the WHERE keyword in the *WHERE* clause.

**DISTINCT (column-1 <, column-2, ...)**

specifies that the values of each *column* must be unique. This constraint is identical to UNIQUE.

**FOREIGN KEY (column-1 <, column-2, ...) REFERENCES table-name <ON DELETE referential-action> <ON UPDATE referential-action>**

specifies a foreign key, that is, a set of *columns* whose values are linked to the values of the primary key variable in another table (the *table-name* that is specified for REFERENCES). The *referential-actions* are performed when the values of a primary key column that is referenced by the foreign key are updated or deleted. *referential-actions* can be the following action:

**CASCADE**

allows primary key data values to be updated, and updates matching values in the foreign key to the same values. This referential action is currently supported for updates only.

**RESTRICT**

occurs only if there are matching foreign key values. This referential action is the default.

**SET NULL**

sets all matching foreign key values to NULL.

**NOT NULL (column)**

specifies that *column* does not contain a null or missing value, including special missing values.

**PRIMARY KEY (column-1 <, column-2, ...)**

specifies one or more primary key columns, that is, columns that do not contain missing values and whose values are unique.

**Restriction**

When you are defining overlapping primary key and foreign key constraints, the variables in a data file are part of both a primary key definition and a foreign key definition. If you use the exact same variables, then the variables must be defined in a different order. The foreign key's update and delete referential actions must both be RESTRICT.

**NOT NULL (column)**

specifies that *column* does not contain a null or missing value, including special missing values.

**PRIMARY KEY (column-1 <, column-2, ...)**

specifies one or more primary key columns, that is, columns that do not contain missing values and whose values are unique.

**Restriction**

When you are defining overlapping primary key and foreign key constraints, the variables in a data file are part of both a primary key and a foreign key definition. If you use the exact same variables, then the variables must be defined in a different order.

**Notes**

If a NOT NULL constraint exists for a variable that is being used to define a new primary key constraint, then the new primary key’s constraint replaces the existing primary key’s constraint.
If an attempt is made to define a NOT NULL constraint for a variable that is already defined for a primary key constraint, then the NOT NULL constraint definition will fail.

`UNIQUE (column-1 <, column-2, ...>)` specifies that the values of each `column` must be unique. This constraint is identical to DISTINCT.

`message-string` specifies the text of an error message that is written to the log when the integrity constraint is not met. The maximum length of `message-string` is 250 characters.

`message-type` specifies how the error message is displayed in the SAS log when an integrity constraint is not met.

`NEWLINE` the text that is specified for `MESSAGE=` is displayed as well as the default error message for that integrity constraint.

`USER` only the text that is specified for `MESSAGE=` is displayed.

`ORDER BY order-by-item-1 <, order-by-item-2, ...>` sorts the rows in `table-name` by the values of each `order-by-item`.

See “ORDER BY Clause” on page 265

Details

Creating a Table without Rows

- The first form of the CREATE TABLE statement creates tables that automatically map SQL data types to tables that are supported by SAS. Use this form when you want to create a new table with columns that are not present in existing tables. It is also useful if you are running SQL statements from an SQL application in another SQL-based database.

- The second form uses a LIKE clause to create a table that has the same column names and column attributes as another table. To drop any columns in the new table, you can specify the DROP= data set option in the CREATE TABLE statement. The specified columns are dropped when the table is created. Indexes are not copied to the new table.

Both of these forms create a table without rows. You can use an INSERT statement to add rows. Use an ALTER TABLE statement to modify column attributes or to add or drop columns.

Creating a Table from a Query Expression

- The third form of the CREATE TABLE statement stores the results of any query expression in a table and does not display the output. It is a convenient way to create temporary tables that are subsets or supersets of other tables.

When you use this form, a table is physically created as the statement is executed. The newly created table does not reflect subsequent changes in the underlying tables (in the query expression). If you want to continually access the most current data, then create a view from the query expression instead of a table. For more information, see “CREATE VIEW Statement” on page 243.

CAUTION:
Recursive table references can cause data integrity problems. Although it is possible to recursively reference the target table of a CREATE TABLE AS statement, doing so can cause data integrity problems and incorrect results. Constructions such as the following should be avoided:

```sql
proc sql;
create table a as select var1, var2 from a;
```

- If you create a new table from an existing table that has extended attributes, then some or all of the extended attributes are copied to the new table. If the new table contains all of the same columns as the existing table that has extended attributes, then all of the extended attributes are copied to the new table. If the new table contains only a subset of the columns from the existing table, then only the extended attributes from the subset of the columns are copied to the new table.

*Note:* Extended attributes are not copied to tables that are created using multi-table joins or outer joins. When UNION, INTERSECT, or minus operators are used, extended attributes are copied only if the table that is listed before the UNION, INTERSECT, or minus operator has extended attributes.

**Integrity Constraints**

You can attach integrity constraints when you create a new table. To modify integrity constraints, use the ALTER TABLE statement.

The following interactions apply to integrity constraints when they are part of a column specification.

- You cannot specify compound primary keys.
- The check constraint that you specify in a column specification does not need to reference the same column in its WHERE clause.
- You can specify more than one integrity constraint.
- You can specify the MSGTYPE= and MESSAGE= options on a constraint.

For more information, see “Understanding Integrity Constraints” in SAS Language Reference: Concepts.

### CREATE VIEW Statement

Creates a PROC SQL view from a query expression.

**See:**
- “What Are Views?” on page 217
- “Example 9: Creating a View from a Query’s Result” on page 290

**Syntax**

```sql
CREATE VIEW proc-sql-view <(column-name-list)> AS query-expression
  <ORDER BY order-by-item-1 <, order-by-item-2, ...>>
  <USING libname-clause-1 <, libname-clause-2, ...>>;
```

**Required Arguments**

`proc-sql-view`

specifies the name for the PROC SQL view that you are creating.

**See**
- “What Are Views?” on page 217
**Optional Arguments**

**column-name-list**

is a comma-separated list of column names for the view, to be used in place of the column names or aliases that are specified in the SELECT clause. The names in this list are assigned to columns in the order in which they are specified in the SELECT clause. If the number of column names in this list does not equal the number of columns in the SELECT clause, then a warning is written to the SAS log.

**ORDER BY order-by-item-1 <, order-by-item-2, …>**

See “ORDER BY Clause” on page 265.

**USING libname-clause-1 <, libname-clause-2, …>**

stores DBMS connection information in the view by embedding the SAS/ACCESS LIBNAME statement inside the view. *libname-clause* can be one of the following:

- **LIBNAME libref <engine> 'SAS-library' <option(s)> <engine-host-option(s)>**
- **LIBNAME libref SAS/ACCESS-engine-name SAS/ACCESS-engine-connection-option(s) <SAS/ACCESS-engine-LIBNAME-option(s)>**

See “LIBNAME Statement” in *SAS Global Statements: Reference*

“LIBNAME Statement Syntax for Relational Databases” in *SAS/ACCESS for Relational Databases: Reference*

**Details**

**Sorting Data Retrieved by Views**

PROC SQL enables you to specify the ORDER BY clause in the CREATE VIEW statement. When a view with an ORDER BY clause is accessed, and the ORDER BY clause directly affects the order of the results, its data is sorted and displayed as specified by the ORDER BY clause. However, if the ORDER BY clause does not directly affect the order of the results (for example, if the view is specified as part of a join), then PROC SQL ignores the ORDER BY clause in order to enhance performance.

*Note:* If the ORDER BY clause is omitted, then a particular order to the output rows, such as the order in which the rows are encountered in the queried table, cannot be guaranteed—even if an index is present. Without an ORDER BY clause, the order of the output rows is determined by the internal processing of PROC SQL, the default collating sequence of SAS, and your operating environment. Therefore, if you want your results to appear in a particular order, then use the ORDER BY clause.

*Note:* If you specify the NUMBER option in the PROC SQL statement when you create your view, then the ROW column appears in the output. However, you cannot order by the ROW column in subsequent queries. See the description of “NUMBER | NONUMBER” on page 226.

**Librefs and Stored Views**

You can refer to a table name alone (without the libref) in the FROM clause of a CREATE VIEW statement if the table and view reside in the same SAS library, as in this example:

```sql
create view proclib.view1 as
```
select *
    from invoice
    where invqty>10;

In this view, VIEW1 and INVOICE are stored permanently in the SAS library referenced by Proclib. Specifying a libref for INVOICE is optional.

**Updating Views**

You can update a view’s underlying data with some restrictions. See “Updating PROC SQL and SAS/ACCESS Views” on page 173.

**Embedded LIBNAME Statements**

The USING clause enables you to store DBMS connection information in a view by embedding the SAS/ACCESS LIBNAME statement inside the view. When PROC SQL executes the view, the stored query assigns the libref and establishes the DBMS connection using the information in the LIBNAME statement. The scope of the libref is local to the view, and will not conflict with any identically named librefs in the SAS session. When the query finishes, the connection to the DBMS is terminated and the libref is deassigned.

The USING clause must be the last clause in the CREATE VIEW statement. Multiple LIBNAME statements can be specified, separated by commas. In the following example, a connection is made and the libref ACCREC is assigned to an ORACLE database.

```sql
create view proclib.view1 as
    select *
        from accrec.invoices as invoices
        using libname accrec oracle
          user=USERNAME
          pass=PASSWORD
          path='dbms-path';
```

For more information about the SAS/ACCESS LIBNAME statement, see the SAS/ACCESS documentation for your DBMS.

**Note:** Starting in SAS System 9, PROC SQL views, the pass-through facility, and the SAS/ACCESS LIBNAME statement are the preferred ways to access relational DBMS data; SAS/ACCESS views are no longer recommended. You can convert existing SAS/ACCESS views to PROC SQL views by using the CV2VIEW procedure. For more information, see “CV2VIEW Procedure” in SAS/ACCESS for Relational Databases: Reference.

You can also embed a SAS LIBNAME statement in a view with the USING clause, which enables you to store SAS libref information in the view. Just as in the embedded SAS/ACCESS LIBNAME statement, the scope of the libref is local to the view, and it will not conflict with an identically named libref in the SAS session.

```sql
create view work.tableview as
    select * from proclib.invoices
    using libname proclib
    'SAS-library';
```

**DELETE Statement**

Removes one or more rows from a table or view that is specified in the FROM clause.
Restriction: You cannot use DELETE FROM on a table that is accessed by an engine that does not support UPDATE processing.

See: “Example 6: Combining Two Tables” on page 280

Syntax

DELETE FROM table-name | sas/access-view | proc-sql-view
  <AS alias>
  <WHERE sql-expression>;

Required Arguments

**table-name** specifies the table that you are deleting rows from. table-name can be a one-level name, a two-level libref.table name, or a physical pathname that is enclosed in single quotation marks.

**CAUTION:** Recursive table references can cause data integrity problems Although it is possible to recursively reference the target table of a DELETE statement, doing so can cause data integrity problems and incorrect results. Constructions such as the following should be avoided:

```sql
proc sql;
  delete from a
  where var1 > (select min(var2) from a);
```

**sas/access-view** specifies a SAS/ACCESS view that you are deleting rows from.

**proc-sql-view** specifies a PROC SQL view that you are deleting rows from. proc-sql-view can be a one-level name, a two-level libref.view name, or a physical pathname that is enclosed in single quotation marks.

Optional Arguments

**AS alias** assigns an alias to table-name, sas/access-view, or proc-sql-view.

**WHERE sql-expression** See “sql-expression” on page 351.

Details

**Deleting Rows through Views**
You can delete one or more rows from a view's underlying table, with some restrictions. See “Updating PROC SQL and SAS/ACCESS Views” on page 173.

**CAUTION:**
If you omit a WHERE clause, then the DELETE statement deletes all of the rows from the specified table or the table that is described by a view. The rows are not actually deleted from the table until it is re-created.
DESCRIBE Statement

Displays a PROC SQL definition in the SAS log.

**Restriction:** PROC SQL views are the only type of view allowed in a DESCRIBE VIEW statement.

**See:** “Example 7: Reporting from DICTIONARY Tables” on page 282

### Syntax

- **DESCRIBE TABLE** `table-name-1 <, table-name-2, ...>;`
- **DESCRIBE VIEW** `proc-sql-view-1 <, proc-sql-view-2, ...>;`
- **DESCRIBE TABLE CONSTRAINTS** `table-name-1 <, table-name-2, ...>;`

### Required Arguments

- **table-name** specifies a PROC SQL table. `table-name` can be a one-level name, a two-level `libref.table` name, or a physical pathname that is enclosed in single quotation marks.

- **proc-sql-view** specifies a PROC SQL view. `proc-sql-view` can be a one-level name, a two-level `libref.view` name, or a physical pathname that is enclosed in single quotation marks.

### Details

- The DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the table specified in the DESCRIBE TABLE statement, regardless of how the table was originally created (for example, with a DATA step). If applicable, SAS data set options are included with the table definition. If indexes are defined on columns in the table, then CREATE INDEX statements for those indexes are also written to the SAS log.

  When you are transferring a table to a DBMS that SAS/ACCESS software supports, it is helpful to know how it is defined. To find out more information about a table, use the FEEDBACK option or the CONTENTS statement in the DATASETS procedure.

- The DESCRIBE VIEW statement writes a view definition to the SAS log. If you use a PROC SQL view in the DESCRIBE VIEW statement that is based on or derived from another view, then you might want to use the FEEDBACK option in the PROC SQL statement. This option is displayed in the SAS log how the underlying view is defined and expands any expressions that are used in this view definition. The CONTENTS statement in DATASETS procedure can also be used with a view to find out more information.

  To define any password-protected SAS view, you must specify a password. If the SAS view was created with more than one password, then you must specify its most restrictive password if you want to access a definition of the view. For example, to define a SAS view that has both Read and Write protection, you must specify its Write password. Similarly, to define a view that has both Read and Alter protection, you must specify its Alter password. (Alter is the more restrictive of the two.) For more information, see “Using Passwords with Views” in SAS Language Reference: Concepts.
DISCONNECT Statement

Ends the connection with a DBMS that a SAS/ACCESS interface supports.

**Requirement:** SAS/ACCESS software is required. For more information about this statement, see your SAS/ACCESS documentation.

**See:** “Connecting to a DBMS By Using the SQL Procedure Pass-Through Facility” on page 171

**Syntax**

```sql
DISCONNECT FROM dbms-name | alias;
```

**Required Arguments**

- `dbms-name`
  - specifies the DBMS from which you want to end the connection (for example, DB2 or ORACLE). The name that you specify should match the name that is specified in the CONNECT statement.

- `alias`
  - specifies the alias that is defined in the CONNECT statement.

**Details**

- An implicit COMMIT is performed before the DISCONNECT statement ends the DBMS connection. If a DISCONNECT statement is not submitted, then implicit DISCONNECT and COMMIT actions are performed and the connection to the DBMS is broken when PROC SQL terminates.
- PROC SQL continues executing until you submit a QUIT statement, another SAS procedure, or a DATA step.

DROP Statement

Deletes tables, views, or indexes.

**Restriction:** You cannot use DROP TABLE or DROP INDEX on a table that is accessed by an engine that does not support UPDATE processing.

**Syntax**

```sql
DROP TABLE table-name-1 <, table-name-2, …>;
DROP VIEW view-name-1 <, view-name-2, …>;
DROP INDEX index-name-1 <, index-name-2, …> FROM table-name;
```
Required Arguments

**table-name**

specifies a PROC SQL table. *table-name* can be a one-level name, a two-level *libref.table* name, or a physical pathname that is enclosed in single quotation marks.

**view-name**

specifies a SAS view of any type: PROC SQL view, SAS/ACCESS view, or DATA step view. *view-name* can be a one-level name, a two-level *libref.view* name, or a physical pathname that is enclosed in single quotation marks.

**index-name**

specifies an index that exists on *table-name*.

Details

- If you drop a table that is referenced in a view definition and try to execute the view, then an error message is written to the SAS log that states that the table does not exist. Therefore, remove references in queries and views to any tables and views that you drop.

- If you drop a table with indexed columns, then all the indexes are automatically dropped. If you drop a composite index, then the index is dropped for all the columns that are named in that index.

- You can use the DROP statement to drop a table or view in an external database that is accessed with the pass-through facility or SAS/ACCESS LIBNAME statement, but not for an external database table or view that a SAS/ACCESS view describes.

EXECUTE Statement

Sends a DBMS-specific SQL statement to a DBMS that a SAS/ACCESS interface supports.

**Requirement:** SAS/ACCESS software is required. For more information about this statement, see your SAS/ACCESS documentation.

**See:** “Connecting to a DBMS By Using the SQL Procedure Pass-Through Facility” on page 171

SQL documentation for your DBMS

**Syntax**

EXECUTE (dbms-SQL-statement)

BY dbms-name | alias;

**Required Arguments**

**dbms-SQL-statement**

is any DBMS-specific SQL statement, except the SELECT statement, which can be executed by the DBMS-specific dynamic SQL. The SQL statement can contain a semicolon. The SQL statement can be case-sensitive, depending on your data source, and it is passed to the data source exactly as you enter it.

**dbms-name**

identifies the DBMS to which you want to direct the DBMS statement (for example, ORACLE or DB2).
**alias**

specifies an optional alias that is defined in the CONNECT statement. Note that *alias* must be preceded by the keyword *BY*.

**Details**

- If your DBMS supports multiple connections, then you can use the alias that is defined in the CONNECT statement. This alias directs the EXECUTE statements to a specific DBMS connection.
- Any return code or message that is generated by the DBMS is available in the macro variables SQLXRC and SQLXMSG after the statement completes.

**Example**

The following example, after the connection, uses the EXECUTE statement to drop a table, create a table, and insert a row of data.

```sql
proc sql;
  execute(drop table 'My Invoice') by db;
  execute(create table 'My Invoice'(
    'Invoice Number' LONG not null,
    'Billed To' VARCHAR(20),
    'Amount' CURRENCY,
    'BILLED ON' DATETIME)) by db;
  execute(insert into 'My Invoice'
          values(12345, 'John Doe', 123.45, #11/22/2003#)) by db;
quit;
```

**INSERT Statement**

Adds rows to a new or existing table or view.

**Restriction:** You cannot use INSERT INTO on a table that is accessed with an engine that does not support UPDATE processing.

**See:** "Example 1: Creating a Table and Inserting Data into It" on page 269

**Syntax**

```
INSERT INTO table-name | sas/access-view | proc-sql-view <(column-1, column-2, ...)>;

  SET column1=sql-expression-1 <, column-2=sql-expression-2, ...>

INSERT INTO table-name | sas/access-view | proc-sql-view <(column-1 <, column-2, ...)>;

  VALUES (value-1 <, value-2, ...>)
  <VALUES (value-1 <, value-2, ...) ...>

INSERT INTO table-name | sas/access-view | proc-sql-view <(column-1 <, column-2, ...)> query-expression;
```
**Required Arguments**

*table-name*

specifies a PROC SQL table into which you are inserting rows. *table-name* can be a one-level name, a two-level libref.table name, or a physical pathname that is enclosed in single quotation marks.

*sas/access-view*

specifies a SAS/ACCESS view into which you are inserting rows.

*proc-sql-view*

specifies a PROC SQL view into which you are inserting rows. *proc-sql-view* can be a one-level name, a two-level libref.view name, or a physical pathname that is enclosed in single quotation marks.

*column*

specifies the column into which you are inserting rows.

*sql-expression*

See “sql-expression” on page 351.

**Restriction**

You cannot use a logical operator (AND, OR, or NOT) in an expression in a SET clause.

*value*

is a data value.

**CAUTION:**

Recursive table references can cause data integrity problems. Although it is possible to recursively reference the target table of an INSERT statement, doing so can cause data integrity problems and incorrect results. Constructions such as the following should be avoided:

```sql
proc sql;
  insert into a
  select var1, var2
  from a
  where var1 > 0;
```

*query-expression*

See “query expression” on page 343.

**Details**

**Methods for Inserting Values**

- The first form of the INSERT statement uses the SET clause, which specifies or alters the values of a column. You can use more than one SET clause per INSERT statement, and each SET clause can set the values in more than one column. Multiple SET clauses are not separated by commas. If you specify an optional list of columns, then you can set a value only for a column that is specified in the list of columns to be inserted.

- The second form of the INSERT statement uses the VALUES clause. This clause can be used to insert lists of values into a table. You can either give a value for each column in the table or give values just for the columns specified in the list of column names. One row is inserted for each VALUES clause. Multiple VALUES clauses are not separated by commas. The order of the values in the VALUES clause matches the order of the column names in the INSERT column list or, if no list was specified, the order of the columns in the table.
The third form of the INSERT statement inserts the results of a query expression into a table. The order of the values in the query expression matches the order of the column names in the INSERT column list or, if no list was specified, the order of the columns in the table.

Note: If the INSERT statement includes an optional list of column names, then only those columns are given values by the statement. Columns that are in the table but not listed are given missing values.

Inserting Rows through Views
You can insert one or more rows into a table through a view, with some restrictions. See “Updating PROC SQL and SAS/ACCESS Views” on page 173.

Adding Values to an Indexed Column
If an index is defined on a column and you insert a new row into the table, then that value is added to the index. You can display information about indexes with the following:

• the CONTENTS statement in the DATASETS procedure. For more information, see “CONTENTS Statement” in Base SAS Procedures Guide.
• the DICTIONARY.INDEXES table. For more information, see “Accessing SAS System Information By Using DICTIONARY Tables” on page 149.

For more information about creating and using indexes, see the “CREATE INDEX Statement” on page 237.

RESET Statement
Resets PROC SQL options without restarting the procedure.

See: “Example 6: Combining Two Tables” on page 280

Syntax

RESET <option(s)>;

Optional Argument

option(s)
the PROC SQL options that you want to add, drop, or change without restarting the procedure.

See For a list of options, see “PROC SQL Statement” on page 221.

SELECT Statement
Selects columns and rows of data from tables and views.

Restriction: The clauses in the SELECT statement must appear in the order shown.

See: “query expression” on page 343
"table-expression” on page 368
SELECT Clause
Lists the columns that will appear in the output.

**Syntax**

```
SELECT <DISTINCT | UNIQUE> object-item-1 <, object-item-2, ...>
    < INTO macro-variable-specification-1 <, macro-variable-specification-2, ...>>
   FROM from-list
   < WHERE sql-expression>
   < GROUP BY group-by-item-1 <, group-by-item-2, ...>>
   < HAVING sql-expression>
   < ORDER BY order-by-item-1 <, order-by-item-2 <ASC | DESC>, ...>>;
```

**SELECT Clause**
Lists the columns that will appear in the output.

See  “Using Column Aliases” on page 146
   “column-definition” on page 320
   “Example 1: Creating a Table and Inserting Data into It” on page 269
   “Example 2: Creating a Table from a Query's Result” on page 271

**Syntax**

```
SELECT <DISTINCT> object-item-1 <, object-item-2, ...>
```

**Required Arguments**

```
alias
```
assigns a temporary, alternate name to the column.

**DISTINCT**
eliminates duplicate rows. The DISTINCT argument is identical to UNIQUE.

**Alias**

```
UNIQUE
```

**Notes**

Although the UNIQUE argument is identical to DISTINCT, it is not an ANSI standard.

DISTINCT works on the internal or stored value, not necessarily on the value as it is displayed. Numeric precision can cause multiple rows to be returned with values that appear to be the same.

**Tips**

A row is considered a duplicate when all of its values are the same as the values of another row. The DISTINCT argument applies to all columns in the SELECT list. One row is displayed for each existing combination of values.

If available, PROC SQL uses index files when processing SELECT DISTINCT statements.

**Example**

“Example 14: Producing All the Possible Combinations of the Values in a Column” on page 304

**object-item**
is one of the following:

```
*  
```

represents all columns in the tables or views that are listed in the FROM clause.
**Details**

**Asterisk (*) Notation**

The asterisk (*) represents all columns of the table or tables listed in the FROM clause. When an asterisk is not prefixed with a table name, all the columns from all tables in the FROM clause are included; when it is prefixed (for example, table-name.* or table-alias.*), all the columns from only that table are included.

**Note:** A warning will occur if you create an output table using the SELECT * syntax when columns with the same name exist in the multiple tables that are listed on the FROM clause. You can avoid the warning by using one of the following actions:

- Individually list the desired columns in the SELECT statement at the same time as you omit the duplicate column names.
- Use the RENAME= and DROP= data set options. In this example, the ID column is renamed `tmpid`.

```sql
proc sql;
create table all(drop=tmpid) as
select * from
    one, two(rename=(id=tmpid))
where one.id=two.tmpid;
quit;
```

If table aliases are used, then place the RENAME= data set option after the table name and before the table alias. You can omit the DROP= data set option if you want to keep the renamed column in the final output table.
INTO Clause

Stores the value of one or more columns for use later in another PROC SQL query or SAS statement.

Restriction

An INTO clause cannot be used in a CREATE TABLE statement.

See

“Using the PROC SQL Automatic Macro Variables” on page 162

Syntax

INTO macro-variable-specification-1 <, macro-variable-specification-2, …>

Required Argument

macro-variable-specification

is one of the following:

macro-variable <SEPARATED BY 'character(s)' <NOTRIM> >
stores the values that are returned into a single macro variable.

macro-variable <TRIMMED>
stores the values that are returned into a single macro variable.

macro-variable-1–macro-variable-n <NOTRIM>
stores the values that are returned into a range of macro variables.

Tip When you specify a range of macro variables, the SAS Macro Facility creates only the number of macro variables that are needed. For example, if you specify :var1-:var9999 and only 55 variables are needed, only :var1-:var55 is created. The SQLOBS automatic variable is useful if a subsequent part of your program needs to know how many variables were actually created. In this example, SQLOBS would have the value of 55.

macro-variable-1 <NOTRIM>
stores the values that are returned into a range of macro variables.

Tip If you do not know how many variables you might need, then you can create a macro variable range without specifying an upper bound for the range. The SQLOBS macro variable can be used if a subsequent part of your program needs to know how many variables were actually created.

macro-variable
specifies a SAS macro variable that stores the values of the rows that are returned.

NOTRIM
protects the leading and trailing blanks from being deleted from values that are stored in a range of macro variables or multiple values that are stored in a single macro variable.

SEPARATED BY 'character'
specifies a character that separates the values of the rows.

TRIMMED
trims the leading and trailing blanks from values that are stored in a single macro variable.
Details

- Use the INTO clause only in the outer query of a SELECT statement, not in a subquery.

- When storing a value in a single macro variable, PROC SQL preserves leading or trailing blanks. The TRIMMED option can be used to trim the leading and trailing blanks from values that are stored in a single macro variable. However, if values are stored in a range of macro variables, or if the SEPARATED BY option is used to store multiple values in a single macro variable, PROC SQL trims leading or trailing blanks unless you specify the NOTRIM option.

- You can put multiple rows of the output in macro variables. You can use the PROC SQL macro variable SQLOBS to determine the number of rows that are produced by a query expression. For more information about SQLOBS, see “Using the PROC SQL Automatic Macro Variables” on page 162.

  Note: The SQLOBS macro variable is assigned a value after the SELECT statement executes.

- Values assigned by the INTO clause use the BEST8. format by default. Large numeric values might be displayed using scientific notation. To display large numeric values without scientific notation, use another format, such as the w. format:

  ```sql
  select sum(population) format=16. into :totpop from sql.countries;
  ```

Example: INTO Clause

**Example 1: INTO Clause**

These examples use the “Proclib.Houses” on page 443 table:

```sql
title 'Proclib.Houses Table';
proc sql;
  select * from proclib.houses;
```
Output 7.1  Proclib.HOUSES Table

With the macro-variable-specification, you can do the following:

- You can create macro variables based on the first row of the result.

```sas
proc sql noprint;
   select style, sqfeet
   into :style, :sqfeet
   from proclib.houses;
%
%put &style &sqfeet;
```

The results are written to the SAS log:

```
CONDO         900
```

- You can use the TRIMMED option to remove leading and trailing blanks from values that are stored in a single macro variable.

```sas
proc sql noprint;
   select distinct style, sqfeet
   into :s1, :s2 TRIMMED
   from proclib.houses;
%
%put &s1 &s2;
%put There were &sqlobs distinct values.;
```

The results are written to the SAS log:

```
INTO Clause 257
```
You can create one new macro variable per row in the result of the SELECT statement. This example shows how you can request more values for one column than for another. The hyphen is used in the INTO clause to imply a range of macro variables. You can use either of the keywords THROUGH or THRU instead of a hyphen.

The following PROC SQL step puts the values from the first four rows of the Proclib.HOUSES table into macro variables:

```sql
proc sql noprint;
select distinct Style, SqFeet
  into :style1 - :style3, :sqfeet1 - :sqfeet4
  from proclib.houses;

%put &style1 &sqfeet1;
%put &style2 &sqfeet2;
%put &style3 &sqfeet3;
%put &sqfeet4;
```

The `%PUT` statements write the results to the SAS log:

```
1 proc sql noprint;
2 select distinct style, sqfeet
3    into :s1, :s2 TRIMMED
4    from proclib.houses;
5 %put &s1 &s2;
CONDO  900
6 %put There were &sqlobs distinct values.;
There were 1 distinct values.
```

You can use a hyphen in the INTO clause to specify a range without an upper bound.

```sql
proc sql noprint;
select distinct Style, SqFeet
  into :style1 - , :sqfeet1 - :sqfeet4
  from proclib.houses;
```

```sql
%put &style1 &sqfeet1;
%put &style2 &sqfeet2;
%put &style3 &sqfeet3;
%put &sqfeet4;
```

The results are written to the SAS log:

```sql
1 proc sql noprint;
```
2 select distinct Style, SqFeet
3       into :style1 , :sqfeet1 -
4       from proclib.houses;
5
6   %put &style1 &sqfeet1;
CONDO 900
7   %put &style2 &sqfeet2;
CONDO 1000
8   %put &style3 &sqfeet3;
RANCH 1200
9   %put &sqfeet4;
1400

• You can concatenate the values of one column into one macro variable. This form is
useful for building up a list of variables or constants. The SQLOBS macro variable is
useful to reveal how many distinct variables there were in the data processed by the
query.

proc sql noprint;
  select distinct style
       into :s1 separated by ','
       from proclib.houses;
%put &s1;
%put There were &sqlobs distinct values.;

The results are written to the SAS log:

3   proc sql noprint;
4     select distinct style
5        into :s1 separated by ','
6        from proclib.houses;
7
8   %put &s1
  CONDO,RANCH,SPLIT,TWOSTORY
There were 4 distinct values.

• You can use leading zeros in order to create a range of macro variable names, as
shown in the following example:

proc sql noprint;
  select SqFeet
       into :sqfeet01 - :sqfeet10
       from proclib.houses;
%put &sqfeet01 &sqfeet02 &sqfeet03 &sqfeet04 &sqfeet05;
%put &sqfeet06 &sqfeet07 &sqfeet08 &sqfeet09 &sqfeet10;

The results are written to the SAS log:

11   proc sql noprint;
12      select sqfeet
13         into :sqfeet01 - :sqfeet10
14      from proclib.houses;
15   %put &sqfeet01 &sqfeet02 &sqfeet03 &sqfeet04 &sqfeet05;
900 1000 1200 1400 1600
16   %put &sqfeet06 &sqfeet07 &sqfeet08 &sqfeet09 &sqfeet10;
You can prevent leading and trailing blanks from being trimmed from values that are stored in macro variables. By default, when storing values in a range of macro variables, or when storing multiple values in a single macro variable (with the SEPARATED BY option), PROC SQL trims the leading and trailing blanks from the values before creating the macro variables. If you do not want leading and trailing blanks to be trimmed, then specify the NOTRIM option, as shown in the following example:

```sql
proc sql noprint;
  select style, sqfeet
    into :style1 - :style4 notrim,
        :sqfeet separated by ',' notrim
    from proclib.houses;
%
  %put *&style1* *&sqfeet*;
  %put *&style2* *&sqfeet*;
  %put *&style3* *&sqfeet*;
  %put *&style4* *&sqfeet*;
```

The results are written to the SAS log, as shown in the following output:

```text
3   proc sql noprint;
 4      select style, sqfeet
 5         into :style1 - :style4 notrim,
 6              :sqfeet separated by ',' notrim
 7         from proclib.houses;
 8
9   %put *&style1* *&sqfeet*;
**CONDO   * *     900,    1000,    1200,    1400,    1600,    1800,    2100, 3000,    1940,    1860**
10  %put *&style2* *&sqfeet*;
**CONDO   * *     900,    1000,    1200,    1400,    1600,    1800,    2100, 3000,    1940,    1860**
11  %put *&style3* *&sqfeet*;
**RANCH   * *     900,    1000,    1200,    1400,    1600,    1800,    2100, 3000,    1940,    1860**
12  %put *&style4* *&sqfeet*;
**RANCH   * *     900,    1000,    1200,    1400,    1600,    1800,    2100, 3000,    1940,    1860**
```
FROM Clause

Specifies source tables or views.

See "Example 1: Creating a Table and Inserting Data into It" on page 269
"Example 4: Joining Two Tables" on page 276
"Example 10: Joining Three Tables" on page 293
"Example 11: Querying an In-Line View" on page 297

Syntax

FROM from-list

Required Argument

from-list

is one of the following:

table-name < <AS> alias>
names a single PROC SQL table. table-name can be a one-level name, a two-level libref.table name, or a physical pathname that is enclosed in single quotation marks. alias specifies a temporary, alternate name for a table, view, or in-line view that is specified in the FROM clause.

view-name < <AS> alias>
names a single SAS view. view-name can be a one-level name, a two-level libref.view name, or a physical pathname that is enclosed in single quotation marks.

joined-table

specifies a join.

See "joined-table" on page 327

(query-expression) < <AS> alias> <(column-1 <, column-2, ...>)>
specifies an in-line view.

alias specifies a temporary, alternate name for a table, view, or in-line view that is specified in the FROM clause.

column names the column that appears in the output. The column names that you specify are matched by position to the columns in the output.

See "query expression" on page 343

CONNECTION TO

specifies a DBMS table.

See "CONNECTION TO" on page 324

Note With table-name and view-name, you can use data set options by placing them in parentheses immediately after table-name or view-name. For more information, see "Using SAS Data Set Options with PROC SQL" on page 156.
Details

**Table Aliases**

A table alias is a temporary, alternate name for a table that is specified in the FROM clause. Table aliases are prefixed to column names to distinguish between columns that are common to multiple tables. Column names in reflexive joins (joining a table with itself) must be prefixed with a table alias in order to distinguish which copy of the table the column comes from. Column names in other types of joins must be prefixed with table aliases or table names unless the column names are unique to those tables.

The optional keyword AS is often used to distinguish a table alias from other table names.

**In-Line Views**

The FROM clause can itself contain a query expression that takes an optional table alias. This type of nested query expression is called an in-line view. An in-line view is any query expression that would be valid in a CREATE VIEW statement. PROC SQL can support many levels of nesting, but it is limited to 256 tables in any one query. The 256-table limit includes underlying tables that can contribute to views that are specified in the FROM clause.

An in-line view saves you a programming step. Rather than creating a view and referring to it in another query, you can specify the view in-line in the FROM clause.

Characteristics of in-line views include the following:

- An in-line view is not assigned a permanent name, although it can take an alias.
- An in-line view can be referred to only in the query in which it is defined. It cannot be referenced in another query.
- You cannot use an ORDER BY clause in an in-line view.
- The names of columns in an in-line view can be assigned in the object-item list of that view or with a list of names enclosed in parentheses following the alias. This syntax can be useful for renaming columns. For an example, see “Example 11: Querying an In-Line View” on page 297.
- In order to visually separate an in-line view from the rest of the query, you can enclose the in-line view in any number of pairs of parentheses. If you specify an alias for the in-line view, then the alias specification must appear outside the outermost pair of parentheses for that in-line view.

**WHERE Clause**

Subsets the output based on specified conditions.

See “Example 4: Joining Two Tables” on page 276
“Example 10: Joining Three Tables” on page 293

**Syntax**

WHERE sql-expression
**Required Argument**

*sql-expression*

See “sql-expression” on page 351.

**Details**

- When a condition is met (that is, the condition resolves to true), those rows are displayed in the result table. Otherwise, no rows are displayed.
- You cannot use summary functions that specify only one column.

In this example, MAX is a summary function. Therefore, its context is that of a GROUP BY clause. It cannot be used to group, or summarize, data.

```sql
where max(measure1) > 50;
```

However, this WHERE clause will work.

```sql
where max(measure1,measure2) > 50;
```

In this case, MAX is a SAS function. It works with the WHERE clause because you are comparing the values of two columns within the same row. Consequently, it can be used to subset the data.

---

**GROUP BY Clause**

Specifies how to group the data for summarizing.

See “Example 9: Creating a View from a Query's Result” on page 290

“Example 13: Joining Two Tables and Calculating a New Value” on page 301

**Syntax**

```
GROUP BY group-by-item-1, group-by-item-2, ...
```

**Required Argument**

*group-by-item*

is one of the following:

- **integer**
  - is a positive integer that equates to a column’s position.

- **column-name**
  - is the name of a column or a column alias.

  See “column-name” on page 323

  “Using Column Aliases” on page 146

- **sql-expression**
  - See “sql-expression” on page 351.

**Details**

- You can specify more than one *group-by-item* to get more detailed reports. Both the grouping of multiple items and the BY statement of a PROC step are evaluated in similar ways. If more than one *group-by-item* is specified, then the first one determines the major grouping.
• Integers can be substituted for column names (that is, SELECT object-items) in the GROUP BY clause. For example, if the group-by-item is 2, then the results are grouped by the values in the second column of the SELECT clause list. Using integers can shorten your coding and enable you to group by the value of an unnamed expression in the SELECT list. If you use a floating-point value (for example, 2.3), then PROC SQL ignores the decimal portion.

• The data does not have to be sorted in the order of the group-by values because PROC SQL handles sorting automatically. You can use the ORDER BY clause to specify the order in which rows are displayed in the result table.

• If you specify a GROUP BY clause in a query that does not contain a summary function, then your clause is transformed into an ORDER BY clause and a message to that effect is written to the SAS log.

• You can group the output by the values that are returned by an expression. For example, if X is a numeric variable, then the output of the following is grouped by the integer portion of values of X:

```sql
select x, sum(y)
from table1
group by int(x);
```

Similarly, if Y is a character variable, then the output of the following is grouped by the second character of values of Y:

```sql
select sum(x), y
from table1
group by substring(y from 2 for 1);
```

An expression that contains only numeric literals (and functions of numeric literals) or only character literals (and functions of character literals) is ignored.

*Note:* If you use an expression such as those in the previous examples, SAS remerges the summary statistics with the original data. Remerging the summary statistics and data might cause unexpected results. For more information, see “Remerging Data” on page 363.

An expression in a GROUP BY clause cannot be a summary function. For example, the following GROUP BY clause is not valid:

```sql
group by sum(x)
```

**HAVING Clause**

Subsets grouped data based on specified conditions.

See  “Using Column Aliases” on page 146

“Example 9: Creating a View from a Query’s Result” on page 290

“Example 13: Joining Two Tables and Calculating a New Value” on page 301

**Syntax**

`HAVING sql-expression`

**Required Argument**

`sql-expression`

See “`sql-expression`” on page 351.
Details

The HAVING clause is used with at least one summary function and an optional GROUP BY clause to summarize groups of data in a table. A HAVING clause is any valid SQL expression that is evaluated as either true or false for each group in a query. Alternatively, if the query involves remerged data, then the HAVING expression is evaluated for each row that participates in each group. The query must include one or more summary functions.

Typically, the GROUP BY clause is used with the HAVING expression and defines the group or groups to be evaluated. If you omit the GROUP BY clause, then the summary function and the HAVING clause treat the table as one group.

The following PROC SQL step uses the Proclib.Payroll table (shown in “Example 2: Creating a Table from a Query's Result” on page 271) and groups the rows by Gender to determine the oldest employee of each gender. In SAS, dates are stored as integers. The lower the birthdate as an integer, the greater the age. The expression birth=min(birth) is evaluated for each row in the table. When the minimum birthdate is found, the expression becomes true and the row is included in the output.

```sql
proc sql;
  title 'Oldest Employee of Each Gender';
  select *
    from proclib.payroll
    group by gender
    having birth=min(birth);
```

Note: This query involves remerged data because the values returned by a summary function are compared to values of a column that is not in the GROUP BY clause. For more information about summary functions and remerging data, see “Remerging Data” on page 363.

ORDER BY Clause

Specifies the order in which rows are displayed in a result table.

See  
“Using Column Aliases” on page 146

“Example 12: Retrieving Values with the SOUNDS-LIKE Operator” on page 299

“query expression” on page 343

Syntax

ORDER BY order-by-item-1 <ASC | DESC> [, order-by-item-2 <ASC | DESC>, …];

Required Arguments

order-by-item

is one of the following:

integer

equates to a column’s position.

column-name

is the name of a column or a column alias.
Note: A column name is not the same as a column label. The only valid use of a column label in a query is as a column heading in the .LST output.

See “column-name” on page 323

sql-expression

See “sql-expression” on page 351.

ASC

orders the data in ascending order. This is the default order. If neither ASC nor DESC is specified, the data is ordered in ascending order.

DESC

orders the data in descending order.

Details

• The ORDER BY clause sorts the results of a query expression according to the order specified in that query. When this clause is used, the default ordering sequence is ascending, from the lowest value to the highest. You can use the SORTSEQ= option to change the collating sequence for your output. See “PROC SQL Statement” on page 221.

• The order of the output rows that are returned is guaranteed only for columns that are specified in the ORDER BY clause.

Note: The ORDER BY clause does not guarantee that the order of the rows generated is deterministic. The ANSI standard for SQL allows the SQL implementation to specify whether the ORDER BY clause is stable or unstable. If the joint combination of values that is referenced in an ORDER BY clause for a query are unique in all of the rows that are being ordered, then the order of rows that is generated by ORDER BY is always deterministic. However, if the ORDER BY clause does not reference a joint combination of unique values, then the order of rows is not deterministic if ORDER BY is unstable.

• If an ORDER BY clause is omitted, then a particular order to the output rows, such as the order in which the rows are encountered in the queried table, cannot be guaranteed—even if an index is present. Without an ORDER BY clause, the order of the output rows is determined by the internal processing of PROC SQL, the default collating sequence of SAS, and your operating environment.

• If more than one order-by-item is specified (separated by commas), then the first one determines the major sort order.

• Integers can be substituted for column names (that is, SELECT object-items) in the ORDER BY clause. For example, if the order-by-item is 2 (an integer), then the results are ordered by the values of the second column. If a query expression includes a set operator (for example, UNION), then use integers to specify the order. Doing so avoids ambiguous references to columns in the table expressions. Note that if you use a floating-point value (for example, 2.3) instead of an integer, then PROC SQL ignores the decimal portion.

• In the ORDER BY clause, you can specify any column of a table or view that is specified in the FROM clause of a query expression, regardless of whether that column has been included in the query's SELECT clause. For example, this query produces a report ordered by the descending values of the population change for each country from 1990 to 1995:

```sql
proc sql;
  select country
  from census
```
order by pop95-pop90 desc;

NOTE: The query as specified involves ordering by an item that doesn't appear in its SELECT clause.

- You can order the output by the values that are returned by an expression. For example, if \( X \) is a numeric variable, then the output of the following is ordered by the integer portion of values of \( X \):

```
select x, y
from table1
order by int(x);
```

Similarly, if \( Y \) is a character variable, then the output of the following is ordered by the second character of values of \( Y \):

```
select x, y
from table1
order by substring(y from 2 for 1);
```

Note that an expression that contains only numeric literals (and functions of numeric literals) or only character literals (and functions of character literals) is ignored.

---

**UPDATE Statement**

Modifies a column's values in existing rows of a table or view.

**Restriction:** You cannot use UPDATE on a table that is accessed by an engine that does not support UPDATE processing.

**See:** “Example 3: Updating Data in a PROC SQL Table” on page 273

**Syntax**

```
UPDATE table-name | sas/access-view | proc-sql-view <AS alias>
    SET column-1=sql-expression-1 <, column-2=sql-expression-2, ...>
    <WHERE sql-expression>;
```

**Required Arguments**

- **table-name** specifies a PROC SQL table. \( table-name \) can be a one-level name, a two-level \( libref.table \) name, or a physical pathname that is enclosed in single quotation marks.
- **sas/access-view** specifies a SAS/ACCESS view.
- **proc-sql-view** specifies a PROC SQL view. \( proc-sql-view \) can be a one-level name, a two-level \( libref.view \) name, or a physical pathname that is enclosed in single quotation marks.
- **alias** assigns an alias to \( table-name, sas/access-view, \) or \( proc-sql-view \).
- **column** specifies a column in \( table-name, sas/access-view, \) or \( proc-sql-view \).
Details

You can update one or more rows of a table through a view, with some restrictions. See “Updating PROC SQL and SAS/ACCESS Views” on page 173.

- Any column that is not modified retains its original values, except in certain queries using the CASE expression. See “CASE Expression” on page 318 for a description of CASE expressions.
- To add, drop, or modify a column’s definition or attributes, use the ALTER TABLE statement, described in “ALTER TABLE Statement” on page 232.
- In the SET clause, a column reference on the left side of the equal sign can also appear as part of the expression on the right side of the equal sign. For example, you could use this expression to give employees a $1,000 holiday bonus:

  set salary=salary + 1000

- If you omit the WHERE clause, then all rows are updated. When you use a WHERE clause, only the rows that meet the WHERE condition are updated.
- When you update a column that is used in an index, the new values are indexed based on the index that was defined on the column.

VALIDATE Statement

Checks the accuracy of a query expression's syntax and semantics without executing the expression.

Syntax

VALIDATE query-expression;

Required Argument

query-expression

See “query expression” on page 343.

Details

- The VALIDATE statement writes a message in the SAS log that states that the query is valid. If there are errors, then VALIDATE writes error messages to the SAS log.
- The VALIDATE statement can also be included in applications that use the macro facility. When used in such an application, VALIDATE returns a value that indicates the query expression’s validity. The value is returned through the macro variable SQLRC (a short form for SQL return code). For example, if a SELECT statement is valid, then the macro variable SQLRC returns a value of 0. For more information, see “Using the PROC SQL Automatic Macro Variables” on page 162.
Examples: SQL Procedure

Example 1: Creating a Table and Inserting Data into It

Features:
- CREATE TABLE statement
- column-modifier
- INSERT statement
- VALUES clause
- SELECT clause
- FROM clause

Table name: Proclib.Paylist

This example creates the table Proclib.Paylist and inserts data into it.

Program

```
libname proclib 'SAS-library';
proc sql;
  create table proclib.paylist
  (IdNum char(4),
   Gender char(1),
   Jobcode char(3),
   Salary num,
   Birth num informat=date7.
     format=date7.,
   Hired num informat=date7.
     format=date7.);
  insert into proclib.paylist
    values('1639','F','TA1',42260,'26JUN70'd,'28JAN91'd)
    values('1065','M','ME3',38090,'26JAN54'd,'07JAN92'd)
    values('1400','M','ME1',29769,'05NOV67'd,'16OCT90'd)
    values('1561','M',null,36514,'30NOV63'd,'07OCT87'd)
    values('1221','F','FA3',.,'22SEP63'd,'04OCT94'd);
title 'Proclib.Paylist Table';
select *
  from proclib.paylist;
proc printto; run;
```

Program Description

Declare the Proclib library. The Proclib library is used in these examples to store created tables.

```
libname proclib 'SAS-library';
```
Create the Proclib.Paylist table. The CREATE TABLE statement creates Proclib.Paylist with six empty columns. Each column definition indicates whether the column is character or numeric. The number in parentheses specifies the width of the column. INFORMAT= and FORMAT= assign date informats and formats to the Birth and Hired columns.

```
proc sql;
  create table proclib.paylist
  (IdNum char(4),
   Gender char(1),
   Jobcode char(3),
   Salary num,
   Birth num informat=date7. format=date7.,
   Hired num informat=date7. format=date7.);
```

Insert values into the Proclib.Paylist table. The INSERT statement inserts data values into Proclib.Paylist according to the position in the VALUES clause. Therefore, in the first VALUES clause, 1639 is inserted into the first column, F into the second column, and so on. Dates in SAS are stored as integers with 0 equal to January 1, 1960. Suffixing the date with a d is one way to use the internal value for dates.

```
insert into proclib.paylist
  values('1639','F','TA1',42260,'26JUN70'd,'28JAN91'd)
values('1065','M','ME3',38090,'26JAN54'd,'07JAN92'd)
values('1400','M','ME1',29769.'05NOV67'd,'16OCT90'd)
```

Include missing values in the data. The value null represents a missing value for the character column Jobcode. The period represents a missing value for the numeric column Salary.

```
values('1561','M',null,36514,'30NOV63'd,'07OCT87'd)
values('1221','F','FA3',.,'22SEP63'd,'04OCT94'd);
```

Specify the title.

```
title 'Proclib.Paylist Table';
```

Display the entire Proclib.Paylist table. The SELECT clause selects columns from Proclib.Paylist. The asterisk (*) selects all columns. The FROM clause specifies Proclib.Paylist as the table to select from.

```
select *
  from proclib.paylist;
```

proc printto; run;
Output: Inserting Data into a Table

Output 7.2  The Proclib.Paylist Table

Proclib.Paylist Table

<table>
<thead>
<tr>
<th>IdNum</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1639</td>
<td>F</td>
<td>TA1</td>
<td>42260</td>
<td>26JUN70</td>
<td>28JAN91</td>
</tr>
<tr>
<td>1065</td>
<td>M</td>
<td>ME3</td>
<td>38090</td>
<td>26JAN54</td>
<td>07JAN92</td>
</tr>
<tr>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>29769</td>
<td>05NOV67</td>
<td>16OCT90</td>
</tr>
<tr>
<td>1561</td>
<td>M</td>
<td></td>
<td>36514</td>
<td>30NOV63</td>
<td>07OCT87</td>
</tr>
<tr>
<td>1221</td>
<td>F</td>
<td>FA3</td>
<td></td>
<td>22SEP63</td>
<td>04OCT94</td>
</tr>
</tbody>
</table>

Example 2: Creating a Table from a Query's Result

Features: CREATE TABLE statement
          AS query expression
          SELECT clause
          columnalias
          FORMAT=column-modifier
          object-item

Other features: Data Set Option
                OBS=

Table names: Proclib.Payroll
            Proclib.Payroll

Details

This example builds a column with an arithmetic expression and creates the Proclib.Bonus table from the query's result.

```sql
proc sql outobs=10;
    title 'Proclib.Payroll';
    title2 'First 10 Rows Only';
    select * from proclib.payroll;
    title;
```
Figure 7.2  Query Result from Proclib.Payroll

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>34376</td>
<td>12SEP60</td>
<td>04JUN87</td>
</tr>
<tr>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>35108</td>
<td>15OCT64</td>
<td>09AUG90</td>
</tr>
<tr>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>29769</td>
<td>05NOV67</td>
<td>16OCT90</td>
</tr>
<tr>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>32886</td>
<td>31AUG65</td>
<td>29JUL90</td>
</tr>
<tr>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>38822</td>
<td>13DEC50</td>
<td>17NOV85</td>
</tr>
<tr>
<td>1499</td>
<td>M</td>
<td>ME3</td>
<td>43025</td>
<td>26APR54</td>
<td>07JUN80</td>
</tr>
<tr>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>18723</td>
<td>06JUN62</td>
<td>01OCT90</td>
</tr>
<tr>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>88606</td>
<td>30MAR61</td>
<td>10FEB81</td>
</tr>
<tr>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>32615</td>
<td>17JAN63</td>
<td>02DEC90</td>
</tr>
<tr>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>38785</td>
<td>22DEC68</td>
<td>05OCT89</td>
</tr>
</tbody>
</table>

Program

libname proclib 'SAS-library';

proc sql;
create table proclib.bonus as
  select IdNumber, Salary format=dollar8.,
         salary*.025 as Bonus format=dollar8.
  from proclib.payroll;

title 'Bonus Information';
select *
  from proclib.bonus(obs=10);

Program Description

Declare the Proclib library. The Proclib library is used in these examples to store created tables.

libname proclib 'SAS-library';

Create the Proclib.Bonus table. The CREATE TABLE statement creates the table Proclib.Bonus from the result of the subsequent query.

proc sql;
create table proclib.bonus as

Select the columns to include. The SELECT clause specifies that three columns will be in the new table: IdNumber, Salary, and Bonus. FORMAT= assigns the DOLLAR8. format to Salary. The Bonus column is built with the SQL expression salary*.025.
select IdNumber, Salary format=dollar8.,
salary*.025 as Bonus format=dollar8.
from proclib.payroll;

Specify the title.

title 'Bonus Information';

Display the first 10 rows of the Proclib.Bonus table. The SELECT clause selects columns from Proclib.Bonus. The asterisk (*) selects all columns. The FROM clause specifies Proclib.Bonus as the table to select from. The OBS= data set option limits the printing of the output to 10 rows.

select *
from proclib.bonus(obs=10);

Output: Creating a Table from a Query

Output 7.3 The Proclib.Bonus Table

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Salary</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td>$34,376</td>
<td>$859</td>
</tr>
<tr>
<td>1653</td>
<td>$35,108</td>
<td>$878</td>
</tr>
<tr>
<td>1400</td>
<td>$29,769</td>
<td>$744</td>
</tr>
<tr>
<td>1350</td>
<td>$32,886</td>
<td>$822</td>
</tr>
<tr>
<td>1401</td>
<td>$38,822</td>
<td>$971</td>
</tr>
<tr>
<td>1499</td>
<td>$43,025</td>
<td>$1,076</td>
</tr>
<tr>
<td>1101</td>
<td>$18,723</td>
<td>$468</td>
</tr>
<tr>
<td>1333</td>
<td>$88,606</td>
<td>$2,215</td>
</tr>
<tr>
<td>1402</td>
<td>$32,615</td>
<td>$815</td>
</tr>
<tr>
<td>1479</td>
<td>$38,785</td>
<td>$970</td>
</tr>
</tbody>
</table>

Example 3: Updating Data in a PROC SQL Table

Features:
- ALTER TABLE statement
- DROP clause
- MODIFY clause
- UPDATE statement
- SET clause
- CASE expression

Table name: Employees
This example updates data values in the Employees table and drops a column.

**Program to Create the Employee Table**

```sql
proc sql;
    title 'Employees Table';
    select * from Employees;
```

**Program Description**

**Display the entire Employees table.** The SELECT clause displays the table before the updates. The asterisk (*) selects all columns for display. The FROM clause specifies Employees as the table to select from.

```sql
proc sql;
    title 'Employees Table';
    select * from Employees;
```

**Output: Creating Employees Table**

**Output 7.4  The Employees Table**

<table>
<thead>
<tr>
<th>IdNum</th>
<th>LName</th>
<th>FName</th>
<th>JobCode</th>
<th>Salary</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1876</td>
<td>CHIN</td>
<td>JACK</td>
<td>TA1</td>
<td>42400</td>
<td>212/588-5634</td>
</tr>
<tr>
<td>1114</td>
<td>GREENWALD</td>
<td>JANICE</td>
<td>ME3</td>
<td>38000</td>
<td>212/588-1092</td>
</tr>
<tr>
<td>1566</td>
<td>PENNINGTON</td>
<td>MICHAEL</td>
<td>ME1</td>
<td>29860</td>
<td>718/383-5681</td>
</tr>
<tr>
<td>1354</td>
<td>PARKER</td>
<td>MARY</td>
<td>FA3</td>
<td>65800</td>
<td>914/455-2337</td>
</tr>
<tr>
<td>1130</td>
<td>WOOD</td>
<td>DEBORAH</td>
<td>PT2</td>
<td>36514</td>
<td>212/587-0013</td>
</tr>
</tbody>
</table>

**Program to Update the Employee Table**

```sql
proc sql;
    update employees
    set salary=salary*
    case when jobcode like '__1' then 1.04
         else 1.025
    end;
    alter table employees
    modify salary num format=dollar8.
    drop phone;
    title 'Updated Employees Table';
    select * from employees;
```
**Program Description**

**Update the values in the Salary column.** The UPDATE statement updates the values in Employees. The SET clause specifies that the data in the Salary column be multiplied by 1.04 when the job code ends with a 1 and 1.025 for all other job codes. (The two underscores represent any character.) The CASE expression returns a value for each row that completes the SET clause.

```sql
proc sql;
update employees
set salary=salary*
    case when jobcode like '__1' then 1.04
         else 1.025
    end;
```

**Modify the format of the Salary column and delete the Phone column.** The ALTER TABLE statement specifies Employees as the table to alter. The MODIFY clause permanently modifies the format of the Salary column. The DROP clause permanently drops the Phone column.

```sql
alter table employees
    modify salary num format=dollar8.
    drop phone;
```

**Specify the title.**

```sql
title 'Updated Employees Table';
```

**Display the entire updated Employees table.** The SELECT clause displays the Employees table after the updates. The asterisk (*) selects all columns.

```sql
select * from employees;
```

**Output: Updating Data in a PROC SQL Table**

**Output 7.5  Updated Employees Table**

<table>
<thead>
<tr>
<th>IdNum</th>
<th>LName</th>
<th>FName</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1876</td>
<td>CHIN</td>
<td>JACK</td>
<td>TA1</td>
<td>$44,096</td>
</tr>
<tr>
<td>1114</td>
<td>GREENWALD</td>
<td>JANICE</td>
<td>ME3</td>
<td>$38,960</td>
</tr>
<tr>
<td>1556</td>
<td>PENNINGTON</td>
<td>MICHAEL</td>
<td>ME1</td>
<td>$31,054</td>
</tr>
<tr>
<td>1354</td>
<td>PARKER</td>
<td>MARY</td>
<td>FA3</td>
<td>$67,445</td>
</tr>
<tr>
<td>1130</td>
<td>WOOD</td>
<td>DEBORAH</td>
<td>PT2</td>
<td>$37,427</td>
</tr>
</tbody>
</table>
Example 4: Joining Two Tables

Features:
- FROM clause
- table alias
- inner join
- joined-table component
- PROC SQL statement option
  - NUMBER
- WHERE clause
  - IN condition

Table names:
- Proclib.Staff
- Proclib.Payroll

Details
This example joins two tables in order to get more information about data that are common to both tables.

```sql
proc sql outobs=10;
  title 'Proclib.Staff';
  title2 'First 10 Rows Only';
  select * from proclib.staff;
  title;
```

Figure 7.3  Proclib.Staff Table

<table>
<thead>
<tr>
<th>idnum</th>
<th>Iname</th>
<th>fname</th>
<th>city</th>
<th>state</th>
<th>hphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td>ADAMS</td>
<td>GERALD</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-1255</td>
</tr>
<tr>
<td>1653</td>
<td>ALIBRANDI</td>
<td>MARIA</td>
<td>BRIDGEPORT</td>
<td>CT</td>
<td>203/675-7715</td>
</tr>
<tr>
<td>1400</td>
<td>ALHERTANI</td>
<td>ABDULLAH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-0808</td>
</tr>
<tr>
<td>1350</td>
<td>ALVAREZ</td>
<td>MERCEDES</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-1549</td>
</tr>
<tr>
<td>1401</td>
<td>ALVAREZ</td>
<td>CARLOS</td>
<td>PATERNON</td>
<td>NJ</td>
<td>201/732-8787</td>
</tr>
<tr>
<td>1499</td>
<td>BAREFOOT</td>
<td>JOSEPH</td>
<td>PRINCETON</td>
<td>NJ</td>
<td>201/812-5665</td>
</tr>
<tr>
<td>1101</td>
<td>BAUCOM</td>
<td>WALTER</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-8060</td>
</tr>
<tr>
<td>1333</td>
<td>BANADYGA</td>
<td>JUSTIN</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-1777</td>
</tr>
<tr>
<td>1402</td>
<td>BLALOCK</td>
<td>RALPH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-2849</td>
</tr>
<tr>
<td>1479</td>
<td>BALLETI</td>
<td>MARIE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-8816</td>
</tr>
</tbody>
</table>
proc sql outobs=10;
title 'Proclib.Payroll';
title2 'First 10 Rows Only';
select * from proclib.payroll;
title;

Figure 7.4 Proclib.Payroll Table

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>34376</td>
<td>12SEP60</td>
<td>04JUN87</td>
</tr>
<tr>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>35108</td>
<td>15OCT64</td>
<td>09AUG90</td>
</tr>
<tr>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>29769</td>
<td>05NOV67</td>
<td>16OCT90</td>
</tr>
<tr>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>32886</td>
<td>31AUG65</td>
<td>29JUL90</td>
</tr>
<tr>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>38822</td>
<td>13DEC50</td>
<td>17NOV85</td>
</tr>
<tr>
<td>1499</td>
<td>M</td>
<td>ME3</td>
<td>43025</td>
<td>26APR54</td>
<td>07JUN80</td>
</tr>
<tr>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>18723</td>
<td>06JUN62</td>
<td>01OCT90</td>
</tr>
<tr>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>88606</td>
<td>30MAR61</td>
<td>10FEB81</td>
</tr>
<tr>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>32615</td>
<td>17JAN63</td>
<td>02DEC90</td>
</tr>
<tr>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>38785</td>
<td>22DEC68</td>
<td>05OCT89</td>
</tr>
</tbody>
</table>

Program

libname proclib 'SAS-library';
proc sql number;
title 'Information for Certain Employees Only';
select Lname, Fname, City, State, IdNumber, Salary, Jobcode
from proclib.staff, proclib.payroll
where idnumber=idnum and idnum in
('1919', '1400', '1350', '1333');

Program Description

**Declare the Proclib library.** The Proclib library is used in these examples to store created tables.

libname proclib 'SAS-library';

**Add row numbers to PROC SQL output.** NUMBER adds a column that contains the row number.
proc sql number;

Specify the title.

title 'Information for Certain Employees Only';

Select the columns to display The SELECT clause selects the columns to show in the output.

select Lname, Fname, City, State,
       IdNumber, Salary, Jobcode

Specify the tables from which to obtain the data. The FROM clause lists the tables to select from.

from proclib.staff, proclib.payroll

Specify the join criterion and subset the query. The WHERE clause specifies that the tables are joined on the ID number from each table. WHERE also further subsets the query with the IN condition, which returns rows for only four employees.

where idnumber=idnum and idnum in
       ('1919', '1400', '1350', '1333');

Output: Joining Two Tables

Output 7.6 Information for Certain Employees Only

<table>
<thead>
<tr>
<th>Row</th>
<th>Lname</th>
<th>Fname</th>
<th>City</th>
<th>State</th>
<th>IdNumber</th>
<th>Salary</th>
<th>Jobcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADAMS</td>
<td>GERALD</td>
<td>STAMFORD</td>
<td>CT</td>
<td>1919</td>
<td>34376</td>
<td>TA2</td>
</tr>
<tr>
<td>2</td>
<td>ALHERTANI</td>
<td>ABDULLAH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>1400</td>
<td>29769</td>
<td>ME1</td>
</tr>
<tr>
<td>3</td>
<td>ALVAREZ</td>
<td>MERCEDES</td>
<td>NEW YORK</td>
<td>NY</td>
<td>1350</td>
<td>32886</td>
<td>FA3</td>
</tr>
<tr>
<td>4</td>
<td>BANADYGA</td>
<td>JUSTIN</td>
<td>STAMFORD</td>
<td>CT</td>
<td>1333</td>
<td>88606</td>
<td>PT2</td>
</tr>
</tbody>
</table>

Example 5: Using an Inner Join

Features: INNER JOIN keywords
Table names: Sql.Oilprod
            Sql.Oilrsrvs

Input Tables

This example uses an inner join to create a new table that contains data columns from two tables. The data is organized by the descending order of the values in the Production column.

Note: The source tables are available in a ZIP file at http://support.sas.com/documentation/onlinedoc/base/index.html. Under the “Related Documentation”
Example 5: Using an Inner Join

Download the ZIP file and extract the data files to a location that is accessible by SAS.

```
libname libref 'SAS-library';
proc sql ;
    select p.country, barrelsperday 'Production', barrels 'Reserves'
    from sql.oilprod p inner join sql.oilrsrvs r
    on p.country = r.country
    order by barrelsperday desc;
```

**Figure 7.5  Table Created with an Inner Join**

<table>
<thead>
<tr>
<th>Country</th>
<th>Production</th>
<th>Reserves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saudi Arabia</td>
<td>9,000,000</td>
<td>260,000,000,000</td>
</tr>
<tr>
<td>United States of America</td>
<td>8,000,000</td>
<td>30,000,000,000</td>
</tr>
<tr>
<td>Iran</td>
<td>4,000,000</td>
<td>90,000,000,000</td>
</tr>
<tr>
<td>Norway</td>
<td>3,500,000</td>
<td>11,000,000,000</td>
</tr>
<tr>
<td>Mexico</td>
<td>3,400,000</td>
<td>50,000,000,000</td>
</tr>
<tr>
<td>China</td>
<td>3,000,000</td>
<td>25,000,000,000</td>
</tr>
</tbody>
</table>

**Program**

```
libname libref 'SAS-library';
proc sql ;
    select p.country, barrelsperday 'Production', barrels 'Reserves'
    from sql.oilprod p inner join sql.oilrsrvs r
    on p.country = r.country
    order by barrelsperday desc;
```

**Program Description**

**Declare the library.** Create a LIBREF for the library that contains the downloaded files from the ZIP file.

```
libname libref 'SAS-library';
```

**Select the tables and specify headers for the columns.** Assign Production as the new title for the BarrelsPerDay column, and assign Reserves as the new title for the Barrels column.

```
proc sql ;
    select p.country, barrelsperday 'Production', barrels 'Reserves'
    from sql.oilprod p inner join sql.oilrsrvs r
    on p.country = r.country
    order by barrelsperday desc;
```

**Assign aliases for the tables, and use an inner join.** Assign the table Sql.Oilprod the alias P, and assign the table Sql.Oilrsrvs the alias R. Use the INNER JOIN keywords to extract the columns.

```
from sql.oilprod p inner join sql.oilrsrvs r
```
on p.country = r.country

**Assign the order of the data.** DESC specifies that the data should be organized by the descending order of the values in the Production column.

```sql
order by barrelsperday desc;
```

---

### Example 6: Combining Two Tables

**Features:**
- DELETE statement
- IS condition
- RESET statement option
  - DOUBLE
- UNION set operator

**Table names:**
- Proclib.Newpay
- Proclib.Paylist
- Proclib.Paylist2

---

#### Input Tables

This example creates a new table, Proclib.Newpay, by concatenating two other tables: Proclib.Paylist and Proclib.Paylist2.

```sql
proc sql;
  title 'Proclib.Paylist Table';
  select * from proclib.paylist;
  title;
```

**Figure 7.6 Proclib.Paylist Table**

<table>
<thead>
<tr>
<th>IdNum</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1639</td>
<td>F</td>
<td>TA1</td>
<td>42260</td>
<td>26JUN70</td>
<td>28JAN91</td>
</tr>
<tr>
<td>1065</td>
<td>M</td>
<td>ME3</td>
<td>38090</td>
<td>26JAN54</td>
<td>07JAN92</td>
</tr>
<tr>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>29769</td>
<td>05NOV67</td>
<td>16OCT90</td>
</tr>
<tr>
<td>1561</td>
<td>M</td>
<td>ME3</td>
<td>36514</td>
<td>30NOV63</td>
<td>07OCT87</td>
</tr>
<tr>
<td>1221</td>
<td>F</td>
<td>FA3</td>
<td>.</td>
<td>22SEP63</td>
<td>04OCT94</td>
</tr>
</tbody>
</table>

```sql
proc sql;
  title 'Proclib.Paylist2 Table';
  select * from proclib.Paylist2;
  title;
```
Program

libname proclib 'SAS-library';

proc sql;
   create table proclib.newpay as
      select * from proclib.paylist
      union
      select * from proclib.paylist2;
   delete
      from proclib.newpay
      where jobcode is missing or salary is missing;
   reset double;
   title 'Personnel Data';
   select *
      from proclib.newpay;

Program Description

Declare the Proclib library. The Proclib library is used in these examples to store created tables.

libname proclib 'SAS-library';

Create the Proclib.Newpay table. The SELECT clauses select all the columns from the tables that are listed in the FROM clauses. The UNION set operator concatenates the query results that are produced by the two SELECT clauses.

proc sql;
   create table proclib.newpay as
      select * from proclib.paylist
      union
      select * from proclib.paylist2;

Delete rows with missing Jobcode or Salary values. The DELETE statement deletes rows from Proclib.Newpay that satisfy the WHERE expression. The IS condition specifies rows that contain missing values in the Jobcode or Salary column.
delete
    from proclib.newpay
    where jobcode is missing or salary is missing;

Reset the PROC SQL environment and double-space the output. RESET changes
the procedure environment without stopping and restarting PROC SQL. The DOUBLE
option double-spaces the output. (The DOUBLE option has no effect on ODS output.)

    reset double;

Specify the title.

    title 'Personnel Data';

Display the entire Proclib.Newpay table. The SELECT clause selects all columns from
the newly created table, Proclib.Newpay.

    select *
        from proclib.newpay;

Output: Combining Two Tables

Output 7.7  The Proclib.Newpay Table

<table>
<thead>
<tr>
<th>IdNum</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1065</td>
<td>M</td>
<td>ME3</td>
<td>38090</td>
<td>26JAN54</td>
<td>07JAN92</td>
</tr>
<tr>
<td>1360</td>
<td>F</td>
<td>FA3</td>
<td>36886</td>
<td>31AUG55</td>
<td>29JUL91</td>
</tr>
<tr>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>29769</td>
<td>05NOV67</td>
<td>16OCT90</td>
</tr>
<tr>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>38822</td>
<td>13DEC55</td>
<td>17NOV93</td>
</tr>
<tr>
<td>1499</td>
<td>M</td>
<td>ME1</td>
<td>23025</td>
<td>26APR74</td>
<td>07JUN92</td>
</tr>
<tr>
<td>1639</td>
<td>F</td>
<td>TA1</td>
<td>42260</td>
<td>26JUN70</td>
<td>28JAN91</td>
</tr>
<tr>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>31896</td>
<td>15OCT64</td>
<td>09AUG92</td>
</tr>
<tr>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>34376</td>
<td>12SEP66</td>
<td>04JUN87</td>
</tr>
</tbody>
</table>

Example 7: Reporting from DICTIONARY Tables

Features: DESCRIBE TABLE statement

DICTIONARY.table-name component

Table name: DICTIONARY.Members

This example uses DICTIONARY tables to show a list of the SAS files in a SAS library.
If you do not know the names of the columns in the DICTIONARY table that you are
querying, then use a DESCRIBE TABLE statement with the table.
Program

libname proclib 'SAS-library';
proc sql;
  describe table dictionary.members;
title 'SAS Files in the Proclib Library';
select memname, memtype
  from dictionary.members
  where libname='PROCLIB';

Program Description

Declare the Proclib library. The Proclib library is used in these examples to store created tables.

libname proclib 'SAS-library';

List the column names from the DICTIONARY.Members table. DESCRIBE TABLE writes the column names from DICTIONARY.Members to the SAS log.

proc sql;
  describe table dictionary.members;

Specify the title.

title 'SAS Files in the Proclib Library';

Display a list of files in the Proclib library. The SELECT clause selects the MEMNAME and MEMTYPE columns. The FROM clause specifies DICTIONARY.Members as the table to select from. The WHERE clause subsets the output to include only those rows that have a libref of Proclib in the LIBNAME column.

select memname, memtype
  from dictionary.members
  where libname='PROCLIB';
Log

Log 7.1 Creating Table DICTIONARY.Members Log

```
277  options nodate pageno=1 source linesize=80 pagesize=60;
278  proc sql;
279     describe table dictionary.members;
280     title 'SAS Files in the Proclib Library';
281     select memname, memtype
282        from dictionary.members
283        where libname='PROCLIB';
```

Output: SAS Files in the Proclib Library

Output 7.8 The Proclib Library

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BONUS</td>
<td>DATA</td>
</tr>
<tr>
<td>NEWPAY</td>
<td>DATA</td>
</tr>
<tr>
<td>PAYLIST</td>
<td>DATA</td>
</tr>
<tr>
<td>PAYLIST2</td>
<td>DATA</td>
</tr>
<tr>
<td>PAYROLL</td>
<td>DATA</td>
</tr>
<tr>
<td>STAFF</td>
<td>DATA</td>
</tr>
</tbody>
</table>

Example 8: Performing an Outer Join

Features: joined-table component
left outer join
SELECT clause
  COALESCE function
WHERE clause
  CONTAINS condition

Table names: Proclib.Payroll
Proclib.Payroll2

Details
This example illustrates a left outer join of the Proclib.Payroll and Proclib.Payroll2 tables.

```
proc sql outobs=10;
  title 'Proclib.Payroll';
  title2 'First 10 Rows Only';
  select * from proclib.payroll
  order by idnumber;
  title;
```

Figure 7.8  Proclib.Payroll

```
proc sql;
  title 'Proclib.Payroll2';
  select * from proclib.payroll2
  order by idnum;
  title;
```
Program Using OUTER JOIN Based on ID Number

```
libname proclib 'SAS-library';
proc sql outobs=10;
title 'Most Current Jobcode and Salary Information';
select p.IdNumber, p.Jobcode, p.Salary,
P2.jobcode label='New Jobcode',
P2.salary label='New Salary' format=dollar8.
from proclib.payroll as p left join proclib.payroll2 as p2
on p.IdNumber=p2.idnum;
```

Program Description

 DECLARE the Proclib library. The Proclib library is used in these examples to store created tables.

```
libname proclib 'SAS-library';
```

LIMIT the number of output rows. OUTOBS= limits the output to 10 rows.

```
proc sql outobs=10;
```

SPECIFY the title for the first query.

```
title 'Most Current Jobcode and Salary Information';
```
Select the columns. The SELECT clause lists the columns to select. Some column names are prefixed with a table alias because they are in both tables. LABEL= and FORMAT= are column modifiers.

```
select p.IdNumber, p.Jobcode, p.Salary,
    p2.jobcode label='New Jobcode',
    p2.salary label='New Salary' format=dollar8.
```

Specify the type of join. The FROM clause lists the tables to join and assigns table aliases. The keywords LEFT JOIN specify the type of join. The order of the tables in the FROM clause is important. Proclib.Payroll is listed first and is considered the “left” table. Proclib.Payroll2 is the “right” table.

```
from proclib.payroll as p left join proclib.payroll2 as p2
```

Specify the join criterion. The ON clause specifies that the join be performed based on the values of the ID numbers from each table.

```
on p.IdNumber=p2.idnum;
```

Output: OUTER JOIN Based on ID Number

As the output shows, all rows from the left table, Proclib.Payroll, are returned. PROC SQL assigns missing values for rows in the left table, Payroll, that have no matching values for IdNum in Payroll2.

Output 7.9 Most Current Jobcode and Salary Information

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Jobcode</th>
<th>Salary</th>
<th>New Jobcode</th>
<th>New Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1009</td>
<td>TA1</td>
<td>28880</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1017</td>
<td>TA3</td>
<td>40858</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1036</td>
<td>TA3</td>
<td>39392</td>
<td>TA3</td>
<td>$42,465</td>
</tr>
<tr>
<td>1037</td>
<td>TA1</td>
<td>28558</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1038</td>
<td>TA1</td>
<td>26533</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1050</td>
<td>ME2</td>
<td>35167</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1065</td>
<td>ME2</td>
<td>35090</td>
<td>ME3</td>
<td>$38,090</td>
</tr>
<tr>
<td>1076</td>
<td>PT1</td>
<td>66558</td>
<td>PT1</td>
<td>$69,742</td>
</tr>
<tr>
<td>1094</td>
<td>FA1</td>
<td>22268</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>BCK</td>
<td>25004</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program Using COALESCE and LEFT JOIN

```
proc sql outobs=10;

title 'Most Current Jobcode and Salary Information';
```
select p.idnumber, coalesce(p2.jobcode,p.jobcode) label='Current Jobcode',
coalesce(p2.salary,p.salary) label='Current Salary'
format=dollar8.
from proclib.payroll p left join proclib.payroll2 p2
on p.IdNumber=p2.idnum;

Program Description

proc sql outobs=10;

Specify the title for the second query.

title 'Most Current Jobcode and Salary Information';

Select the columns and coalesce the Jobcode columns. The SELECT clause lists the
columns to select. COALESCE overlays the like-named columns. For each row,
COALESCE returns the first nonmissing value of either P2.JobCode or P.JobCode.
Because P2.JobCode is the first argument, if there is a nonmissing value for P2.JobCode,
COALESCE returns that value. Thus, the output contains the most recent job code
information for every employee. LABEL= assigns a column label.

select p.idnumber, coalesce(p2.jobcode,p.jobcode) label='Current Jobcode',
coalesce(p2.salary,p.salary) label='Current Salary'
format=dollar8.
from proclib.payroll p left join proclib.payroll2 p2
on p.IdNumber=p2.idnum;

Specify the type of join and the join criterion. The FROM clause lists the tables to join
and assigns table aliases. The keywords LEFT JOIN specify the type of join. The ON
clause specifies that the join is based on the ID numbers from each table.

from proclib.payroll p left join proclib.payroll2 p2
on p.IdNumber=p2.idnum;
Output: COALESCE and LEFT JOIN

Output 7.10 Most Current Jobcode and Salary Information

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Current Jobcode</th>
<th>Current Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1009</td>
<td>TA1</td>
<td>$28,880</td>
</tr>
<tr>
<td>1017</td>
<td>TA3</td>
<td>$40,858</td>
</tr>
<tr>
<td>1036</td>
<td>TA3</td>
<td>$42,465</td>
</tr>
<tr>
<td>1037</td>
<td>TA1</td>
<td>$28,558</td>
</tr>
<tr>
<td>1038</td>
<td>TA1</td>
<td>$26,533</td>
</tr>
<tr>
<td>1050</td>
<td>ME2</td>
<td>$35,167</td>
</tr>
<tr>
<td>1065</td>
<td>ME3</td>
<td>$38,090</td>
</tr>
<tr>
<td>1076</td>
<td>PT1</td>
<td>$69,742</td>
</tr>
<tr>
<td>1094</td>
<td>FA1</td>
<td>$22,268</td>
</tr>
<tr>
<td>1100</td>
<td>BCK</td>
<td>$25,004</td>
</tr>
</tbody>
</table>

Program to Subset the Query

```sql
proc sql;
  title 'Most Current Information for Ticket Agents';
  select p.IdNumber,
     coalesce(p2.jobcode,p.jobcode) label='Current Jobcode',
     coalesce(p2.salary,p.salary) label='Current Salary'
  from proclib.payroll p left join proclib.payroll2 p2
  on p.IdNumber=p2.idnum
  where p2.jobcode contains 'TA';
```

Program Description

Subset the query. The WHERE clause subsets the left join to include only those rows containing the value TA.

```sql
proc sql;
  title 'Most Current Information for Ticket Agents';
  select p.IdNumber,
     coalesce(p2.jobcode,p.jobcode) label='Current Jobcode',
     coalesce(p2.salary,p.salary) label='Current Salary'
  from proclib.payroll p left join proclib.payroll2 p2
  on p.IdNumber=p2.idnum
  where p2.jobcode contains 'TA';
```
Example 9: Creating a View from a Query’s Result

Features:
- CREATE VIEW statement
- GROUP BY clause
- SELECT clause
  - COUNT function
- HAVING clause

Other features:
- AVG summary function
- data set option
- PW=

Table names:
- Proclib.Payroll
- Proclib.Jobs

Details

This example creates the PROC SQL view Proclib.Jobs with four columns resulting from a query expression. One physical column is selected from the source table, and then the COUNT and AVG functions are used to create three derived columns.

Submit this code to display the first 10 rows of the table used in the example in the Program topic.

```plaintext
title1 'Proclib.Payroll';
title2 'First 10 Rows Only';

proc sql outobs=10;
  select * from proclib.payroll
  order by idnumber;
quit;

title;
```
Program

libname proclib 'SAS-library';

proc sql;
  create view proclib.jobs(pw=red) as
    select Jobcode,
       count(jobcode) as number label='Number',
       avg(int((today()-birth)/365.25)) as avgage
           format=2. label='Average Age',
       avg(salary) as avgsal
           format=dollar8. label='Average Salary'
    from payroll
    group by jobcode
    having avgage ge 30;
  title1 'Current Summary Information for Each Job Category';
  title2 'Average Age Greater Than or Equal to 30';
  select * from proclib.jobs(pw=red);
  title;

Program Description

Declare the Proclib library. The Proclib library is used in these examples to store created tables.

libname proclib 'SAS-library';
Create the Proclib.Jobs view. CREATE VIEW creates the PROC SQL view Proclib.Jobs. The PW= data set option assigns password protection to the data that is generated by this view.

```sql
proc sql;
  create view proclib.jobs(pw=red) as
```

**Select the columns.** The SELECT clause specifies four columns for the view: JobCode and three columns, Number, AvgAge, and AvgSal, whose values are the products functions. COUNT returns the number of nonmissing values for each job code because the data is grouped by Jobcode. LABEL= assigns a label to the column.

```sql
  select Jobcode,
        count(jobcode) as number label='Number',
```

**Calculate the AvgAge and AvgSal columns.** The AVG summary function calculates the average age and average salary for each job code.

```sql
    avg(int((today()-birth)/365.25)) as avgage
      format=2. label='Average Age',
    avg(salary) as avgsal
      format=dollar8. label='Average Salary'
```

**Specify the table from which the data is obtained.** The FROM clause specifies Payroll as the table to select from. PROC SQL assumes the libref of Payroll to be Proclib because Proclib is used in the CREATE VIEW statement.

```sql
  from payroll
```

**Organize the data into groups and specify the groups to include in the output.** The GROUP BY clause groups the data by the values of Jobcode. Thus, any summary statistics are calculated for each grouping of rows by value of Jobcode. The HAVING clause subsets the grouped data and returns rows for job codes that contain an average age of greater than or equal to 30.

```sql
  group by jobcode
    having avgage ge 30;
```

**Specify the titles.**

```sql
  title1 'Current Summary Information for Each Job Category';
  title2 'Average Age Greater Than or Equal to 30';
```

**Display the entire Proclib.Jobs view.** The SELECT statement selects all columns from Proclib.Jobs. PW=RED is necessary because the view is password protected. The QUIT statement ends the SQL procedure.

```sql
  select * from proclib.jobs(pw=red);
  title;
```
### Example 10: Joining Three Tables

**Features:**
- FROM clause
- joined-table component
- WHERE clause

**Table names:**
- `Proclib.Staff2`
- `Proclib.Schedule2`
- `Proclib.Superv2`

**Details**
This example joins three tables and produces a report that contains columns from each table.

### Current Summary Information for Each Job Category

**Average Age Greater Than or Equal to 30**

<table>
<thead>
<tr>
<th>Jobcode</th>
<th>Number</th>
<th>Average Age</th>
<th>Average Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK</td>
<td>9</td>
<td>47</td>
<td>$25,794</td>
</tr>
<tr>
<td>FA1</td>
<td>11</td>
<td>44</td>
<td>$23,039</td>
</tr>
<tr>
<td>FA2</td>
<td>16</td>
<td>48</td>
<td>$27,987</td>
</tr>
<tr>
<td>FA3</td>
<td>7</td>
<td>50</td>
<td>$32,934</td>
</tr>
<tr>
<td>ME1</td>
<td>8</td>
<td>45</td>
<td>$28,500</td>
</tr>
<tr>
<td>ME2</td>
<td>14</td>
<td>51</td>
<td>$35,577</td>
</tr>
<tr>
<td>ME3</td>
<td>7</td>
<td>53</td>
<td>$42,411</td>
</tr>
<tr>
<td>NA1</td>
<td>5</td>
<td>41</td>
<td>$42,032</td>
</tr>
<tr>
<td>NA2</td>
<td>3</td>
<td>53</td>
<td>$52,383</td>
</tr>
<tr>
<td>PT1</td>
<td>8</td>
<td>49</td>
<td>$67,908</td>
</tr>
<tr>
<td>PT2</td>
<td>10</td>
<td>54</td>
<td>$87,925</td>
</tr>
<tr>
<td>SCP</td>
<td>7</td>
<td>48</td>
<td>$18,309</td>
</tr>
<tr>
<td>TA1</td>
<td>9</td>
<td>47</td>
<td>$27,721</td>
</tr>
<tr>
<td>TA2</td>
<td>20</td>
<td>48</td>
<td>$33,575</td>
</tr>
<tr>
<td>TA3</td>
<td>12</td>
<td>51</td>
<td>$39,680</td>
</tr>
</tbody>
</table>
Example Code 1  Proclib.Staff2 Table

data proclib.staff2;
input IdNum $4. @7 Lname $12. @20 Fname $8. @30 City $10. @42 State $2. @50 Hphone $12.;
datalines;
1106 MARSHBURN JASPER STAMFORD CT 203/781-1457
1430 DABROWSKI SANDRA BRIDGEPORT CT 203/675-1647
1118 DENNIS ROGER NEW YORK NY 718/383-1122
1126 KIMANI ANNE NEW YORK NY 212/586-1229
1402 BLALOCK RALPH NEW YORK NY 718/384-2849
1882 TUCKER ALAN NEW YORK NY 718/384-0216
1479 BALLETTI MARIE NEW YORK NY 718/384-8816
1420 ROUSE JEREMY PATERSON NJ 201/732-9834
1403 BOWDEN EARL BRIDGEPORT CT 203/675-3434
1616 FUENTAS CARLA NEW YORK NY 718/384-3329;
;
run;

proc sql;
  title 'Proclib.Staff2';
  select * from proclib.staff2;
  title;

Figure 7.11  Proclib.Staff2

Example Code 2  Proclib.Schedule2 Table

data proclib.schedule2;
  input flight $3. +5 date date7. +2 dest $3. +3 idnum $4.;
  format date date7.;
informat date date7.;
data proclib.superv2;
input supid $4. +8 state $2. +5 jobcat  $2.;
label supid='Supervisor Id' jobcat='Job Category';
datalines;
1417        NJ     NA
1352        NY     NA
1106        CT     PT
1442        NJ     PT
1118        NY     PT
;
run;

proc sql;
  title 'Proclib.Schedule2';
  select * from proclib.schedule2;
  title;

Figure 7.12  Proclib.Schedule2

<table>
<thead>
<tr>
<th>flight</th>
<th>date</th>
<th>dest</th>
<th>idnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>01MAR94</td>
<td>BOS</td>
<td>1118</td>
</tr>
<tr>
<td>132</td>
<td>01MAR94</td>
<td>BOS</td>
<td>1402</td>
</tr>
<tr>
<td>219</td>
<td>02MAR94</td>
<td>PAR</td>
<td>1616</td>
</tr>
<tr>
<td>219</td>
<td>02MAR94</td>
<td>PAR</td>
<td>1478</td>
</tr>
<tr>
<td>622</td>
<td>03MAR94</td>
<td>LON</td>
<td>1430</td>
</tr>
<tr>
<td>622</td>
<td>03MAR94</td>
<td>LON</td>
<td>1882</td>
</tr>
<tr>
<td>271</td>
<td>04MAR94</td>
<td>NYC</td>
<td>1430</td>
</tr>
<tr>
<td>271</td>
<td>04MAR94</td>
<td>NYC</td>
<td>1118</td>
</tr>
<tr>
<td>579</td>
<td>05MAR94</td>
<td>RDU</td>
<td>1126</td>
</tr>
<tr>
<td>579</td>
<td>05MAR94</td>
<td>RDU</td>
<td>1106</td>
</tr>
</tbody>
</table>
;

Example Code 3  Proclib.Superv2 Table
Program

libname proclib 'SAS-library';

proc sql;
  title 'All Flights for Each Supervisor';
  select s.IdNum, Lname, City 'Hometown', Jobcat,
       Flight, Date
  from proclib.schedule2 s, proclib.staff2 t, proclib.superv2 v
  where s.idnum=t.idnum and t.idnum=v.supid;

Program Description

Declare the Proclib library. The Proclib library is used in these examples to store created tables.
libname proclib 'SAS-library';

Select the columns. The SELECT clause specifies the columns to select. IdNum is prefixed with a table alias because it appears in two tables.

   proc sql;
      title 'All Flights for Each Supervisor';
      select s.IdNum, Lname, City 'Hometown', Jobcat, Flight, Date
      from proclib.schedule2 s, proclib.staff2 t, proclib.superv2 v
      where s.idnum=t.idnum and t.idnum=v.supid;

Specify the tables to include in the join. The FROM clause lists the three tables for the join and assigns an alias to each table.

Specify the join criteria. The WHERE clause specifies the columns that join the tables. The Staff2 and Schedule2 tables each have an IdNum column, which enables a join on rows where these column values match in both tables. The Staff2 and Superv2 tables have the IdNum and SupId columns, which enable a join on rows where these column values match in both tables. The combination of these two conditions enables the three tables to be joined.

Output: Joining Three Tables

Output 7.13  All Flights for Each Supervisor

<table>
<thead>
<tr>
<th>idnum</th>
<th>Lname</th>
<th>Hometown</th>
<th>Job Category</th>
<th>flight</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1106</td>
<td>MARSHBURN</td>
<td>STAMFORD</td>
<td>PT</td>
<td>579</td>
<td>05MAR94</td>
</tr>
<tr>
<td>1118</td>
<td>DENNIS</td>
<td>NEW YORK</td>
<td>PT</td>
<td>132</td>
<td>01MAR94</td>
</tr>
<tr>
<td>1118</td>
<td>DENNIS</td>
<td>NEW YORK</td>
<td>PT</td>
<td>271</td>
<td>04MAR94</td>
</tr>
<tr>
<td>1126</td>
<td>KIMANI</td>
<td>NEW YORK</td>
<td>TA</td>
<td>579</td>
<td>05MAR94</td>
</tr>
<tr>
<td>1882</td>
<td>TUCKER</td>
<td>NEW YORK</td>
<td>ME</td>
<td>622</td>
<td>03MAR94</td>
</tr>
</tbody>
</table>

Example 11: Querying an In-Line View

Features: FROM clause
in-line view

Table names: Proclib.Staff2
Proclib.Schedule2
Proclib.Superv2

This example shows an alternative way to construct the query that is explained in “Example 10: Joining Three Tables” on page 293 by joining one of the tables with the
results of an in-line view. The example also shows how to rename columns with an in-line view.

**Program**

```sas
libname proclib 'SAS-library';
proc sql;
    title 'All Flights for Each Supervisor';
    select three.*, v.jobcat
    from (select lname, s.idnum, city, flight, date
        from proclib.schedule2 s, proclib.staff2 t
        where s.idnum=t.idnum)
    as three (Surname, Emp_ID, Hometown, FlightNumber, FlightDate),
    proclib.superv2 v
        where three.Emp_ID=v.supid;
```

**Program Description**

**Declare the Proclib library.** The Proclib library is used in these examples to store created tables.

```sas
libname proclib 'SAS-library';
```

**Select the columns.** The SELECT clause selects all columns that are returned by the in-line view (to which the alias Three is assigned), plus one column from the third table (to which the alias V is assigned).

```sas
proc sql;
    title 'All Flights for Each Supervisor';
    select three.*, v.jobcat
```

**Specify the in-line query.** Instead of including the name of a table or view, the FROM clause includes a query that joins two of the three tables. In the in-line query, the SELECT clause lists the columns to select. IdNum is prefixed with a table alias because it appears in both tables. The FROM clause lists the two tables for the join and assigns an alias to each table. The WHERE clause specifies the columns that join the tables. The Staff2 and Schedule2 tables each have an IdNum column, which enables a join on rows where these column values match in both tables.

```sas
from (select lname, s.idnum, city, flight, date
        from proclib.schedule2 s, proclib.staff2 t
        where s.idnum=t.idnum)
    as three (Surname, Emp_ID, Hometown, FlightNumber, FlightDate),
```

**Specify an alias for the query and names for the columns.** The alias Three refers to the results of the in-line view. The names in parentheses become the names for the columns in the view.

```sas
    proclib.superv2 v
        where three.Emp_ID=v.supid;
```

**Join the results of the in-line view with the third table.** The WHERE clause specifies the columns that join the table with the in-line view. Note that the WHERE clause specifies the renamed Emp_ID column from the in-line view.
Example 12: Retrieving Values with the SOUNDS-LIKE Operator

Features:
- ORDER BY clause
- SOUNDS-LIKE operator

Table name: Proclib.Staff

This example returns rows based on the functionality of the SOUNDS-LIKE operator in a WHERE clause. The SOUNDS-LIKE operator is based on the SOUNDEX algorithm for identifying words that sound alike. The SOUNDEX algorithm is English-biased and is less useful for languages other than English. For more information about the “SOUNDEX Function” in SAS Functions and CALL Routines: Reference algorithm, see SAS Functions and CALL Routines: Reference.

Details

```sql
proc sql outobs=10;
    title 'Proclib.Staff';
    title2 'First 10 Rows Only';
    select * from proclib.staff;
    title;
```
### Program to Select Names That Sound like 'Johnson'

```sas
libname proclib 'SAS-library';
proc sql;
  title "Employees Whose Last Name Sounds Like 'Johnson'";
  select idnum, upcase(lname), fname
  from proclib.staff
  where lname="Johnson"
  order by 2;
```

### Program Description

**Declare the Proclib library.** The Proclib library is used in these examples to store created tables.

```sas
libname proclib 'SAS-library';
```

**Select the columns and the table from which the data is obtained.** The SELECT clause selects all columns from the table in the FROM clause, Proclib.Staff.

```sas
proc sql;
  title "Employees Whose Last Name Sounds Like 'Johnson'";
  select idnum, upcase(lname), fname
  from proclib.staff
```

**Subset the query and sort the output.** The WHERE clause uses the SOUNDS-LIKE operator to subset the table by those employees whose last name sounds like Johnson. The ORDER BY clause orders the output by the second column.

<table>
<thead>
<tr>
<th>idnum</th>
<th>iname</th>
<th>fname</th>
<th>city</th>
<th>state</th>
<th>hphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td>ADAMS</td>
<td>GERALD</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-1255</td>
</tr>
<tr>
<td>1653</td>
<td>ALIBRANDI</td>
<td>MARIA</td>
<td>BRIDGEPORT</td>
<td>CT</td>
<td>203/675-7715</td>
</tr>
<tr>
<td>1400</td>
<td>ALBERTANI</td>
<td>ABDULLAH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-0808</td>
</tr>
<tr>
<td>1350</td>
<td>ALVAREZ</td>
<td>MERCEDES</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-1549</td>
</tr>
<tr>
<td>1401</td>
<td>ALVAREZ</td>
<td>CARLOS</td>
<td>PATERSON</td>
<td>NJ</td>
<td>201/732-8787</td>
</tr>
<tr>
<td>1499</td>
<td>BAREFOOT</td>
<td>JOSSEPH</td>
<td>PRINCETON</td>
<td>NJ</td>
<td>201/812-5665</td>
</tr>
<tr>
<td>1101</td>
<td>BAUCOM</td>
<td>WALTER</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-8060</td>
</tr>
<tr>
<td>1333</td>
<td>BANADYG</td>
<td>JUSTIN</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-1777</td>
</tr>
<tr>
<td>1402</td>
<td>BLALOCK</td>
<td>RALPH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-2849</td>
</tr>
<tr>
<td>1479</td>
<td>BALLETTI</td>
<td>MARIE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-8816</td>
</tr>
</tbody>
</table>
where lname="*Johnson"
    order by 2;

**Output:** Names That Sound like 'Johnson'

**Output 7.15**  Johnson Employee Table

#### Employees Whose Last Name Sounds Like 'Johnson'

<table>
<thead>
<tr>
<th>idnum</th>
<th>lname</th>
<th>fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1411</td>
<td>JOHNSN</td>
<td>JACK</td>
</tr>
<tr>
<td>1113</td>
<td>JOHNSN</td>
<td>LESLIE</td>
</tr>
<tr>
<td>1369</td>
<td>JONSN</td>
<td>ANTHONY</td>
</tr>
</tbody>
</table>

**Program to Select Names That Sound like 'Sanders'

SOUNDS-LIKE is useful, but there might be instances where it does not return every row that seems to satisfy the condition. Proclib.Staff has an employee with the last name Sanders and an employee with the last name Sanyers. The algorithm does not find Sanyers, but it does find Sanders and Sanderson.

```sql
proc sql;
    title "Employees Whose Last Name Sounds Like 'Sanders'";
    select *
        from proclib.staff
    where lname="*Sanders"
    order by 2;
```

**Output:** Names That Sound like 'Sanders'

**Output 7.16**  Sanders Employee Table

#### Employees Whose Last Name Sounds Like 'Sanders'

<table>
<thead>
<tr>
<th>idnum</th>
<th>lname</th>
<th>fname</th>
<th>city</th>
<th>state</th>
<th>hphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1561</td>
<td>SANDERS</td>
<td>RAYMOND</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/588-6615</td>
</tr>
<tr>
<td>1414</td>
<td>SANDERSON</td>
<td>NATHAN</td>
<td>BRIDGEPORT</td>
<td>CT</td>
<td>203/675-1715</td>
</tr>
<tr>
<td>1434</td>
<td>SANDERSON</td>
<td>EDITH</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-1333</td>
</tr>
</tbody>
</table>

---

**Example 13: Joining Two Tables and Calculating a New Value**

**Features:**
- GROUP BY clause
- HAVING clause
- SELECT clause
- ABS function
The tables contain X and Y coordinates that represent the location of the stores and houses.
Figure 7.15 Stores and Houses Tables

**Stores Table**
Coordinates of Stores

<table>
<thead>
<tr>
<th>Store</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>store1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>store2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>store3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>store4</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

**Houses Table**
Coordinates of Houses

<table>
<thead>
<tr>
<th>House</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>house1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>house2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>house3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>house4</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

**Program**

```sql
proc sql;
  title 'Each House and the Closest Store';
  select house, store label='Closest Store',
     sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2)) as dist
  label='Distance' format=4.2
  from stores s, houses h
  group by house
  having dist=min(dist);
```

**Program Description**

**Specify the query.** The SELECT clause specifies three columns: HOUSE, STORE, and DIST. The arithmetic expression uses the square root function (SQRT) to create the values of DIST, which contain the distance from HOUSE to STORE for each row. The double asterisk (**) represents exponentiation. LABEL= assigns a label to STORE and to DIST.

```sql
proc sql;
  title 'Each House and the Closest Store';
  select house, store label='Closest Store',
     sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2)) as dist
```
Organize the data into groups and subset the query. The minimum distance from each house to all the stores is calculated because the data are grouped by house. The HAVING clause specifies that each row be evaluated to determine whether its value of DIST is the same as the minimum distance from that house to any store.

```
group by house
    having dist=min(dist);
```

**Output: Joining Two Tables and Calculating a New Value**

Note that two stores are tied for shortest distance from house2.

**Output 7.17  Each House and the Closest Store**

```
<table>
<thead>
<tr>
<th>House</th>
<th>Closest Store</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>house1</td>
<td>store1</td>
<td>4.00</td>
</tr>
<tr>
<td>house2</td>
<td>store2</td>
<td>2.00</td>
</tr>
<tr>
<td>house2</td>
<td>store3</td>
<td>2.00</td>
</tr>
<tr>
<td>house3</td>
<td>store3</td>
<td>2.24</td>
</tr>
<tr>
<td>house4</td>
<td>store4</td>
<td>2.00</td>
</tr>
</tbody>
</table>
```

**Example 14: Producing All the Possible Combinations of the Values in a Column**

**Features:**
- CASE expression
- joined-table component
- Cross join
- SELECT clause
- DISTINCT keyword

**Table names:** Proclib.March
- Flights

**Details**

This example joins a table with itself to get all the possible combinations of the values in a column.

```
proc sql outobs=10;
    title 'Proclib.March';
    title2 'First 10 Rows Only';
    select * from proclib.march;
```
Program to Create the Flights Table

```sql
libname proclib 'SAS-library';
proc sql;
create table flights as
    select distinct dest
    from proclib.march;
title 'Cities Serviced by the Airline';
select * from flights;
```

Program Description

**Declare the Proclib library.** The Proclib library is used in these examples to store created tables.

```sql
libname proclib 'SAS-library';
```

**Create the Flights table.** The CREATE TABLE statement creates the table Flights from the output of the query. The SELECT clause selects the unique values of Dest. DISTINCT specifies that only one row for each value of city be returned by the query and stored in the table Flights. The FROM clause specifies Proclib.March as the table to select from.

```sql
proc sql;
create table flights as
    select distinct dest
    from proclib.march;
```
from proclib.march;

Specify the title.
    title 'Cities Serviced by the Airline';

Display the entire Flights table.
    select * from flights;

Output: Creating the Flights Table

Output 7.18  The Flights Table

<table>
<thead>
<tr>
<th>Cities Serviced by the Airline</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
</tr>
<tr>
<td>FRA</td>
</tr>
<tr>
<td>LAX</td>
</tr>
<tr>
<td>LON</td>
</tr>
<tr>
<td>ORD</td>
</tr>
<tr>
<td>PAR</td>
</tr>
<tr>
<td>WAS</td>
</tr>
<tr>
<td>YYZ</td>
</tr>
</tbody>
</table>

Program Using Conventional Join

    proc sql;
    title 'All Possible Connections';
    select f1.Dest, case
            when f1.dest ne ' ' then 'to and from'
        end,
        f2.Dest
    from flights as f1, flights as f2
    where f1.dest < f2.dest
    order by f1.dest;

Program Description

    proc sql;

Specify the title.
    title 'All Possible Connections';
Select the columns. The SELECT clause specifies three columns for the output. The prefixes on DEST are table aliases to specify which table to take the values of Dest from. The CASE expression creates a column that contains the character string to and from.

```sql
select f1.Dest, case
    when f1.dest ne ' ' then 'to and from'
    end,
    f2.Dest
```

Specify the type of join. The FROM clause joins Flights with itself and creates a table that contains every possible combination of rows (a Cartesian product). The table contains two rows for each possible route (for example, PAR <-> WAS and WAS <-> PAR).

```sql
from flights as f1, flights as f2
```

Specify the join criterion. The WHERE clause subsets the internal table by choosing only those rows where the name in F1.Dest sorts before the name in F2.Dest. Thus, there is only one row for each possible route.

```sql
where f1.dest < f2.dest
```

Sort the output. ORDER BY sorts the result by the values of F1.Dest.

```sql
order by f1.dest;
```
Output: Conventional Join

Output 7.19  All Possible Connections

<table>
<thead>
<tr>
<th>dest</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRA</td>
<td>to and from</td>
</tr>
<tr>
<td>FRA</td>
<td>to and from</td>
</tr>
<tr>
<td>FRA</td>
<td>to and from</td>
</tr>
<tr>
<td>FRA</td>
<td>to and from</td>
</tr>
<tr>
<td>FRA</td>
<td>to and from</td>
</tr>
<tr>
<td>FRA</td>
<td>to and from</td>
</tr>
<tr>
<td>LAX</td>
<td>to and from</td>
</tr>
<tr>
<td>LAX</td>
<td>to and from</td>
</tr>
<tr>
<td>LAX</td>
<td>to and from</td>
</tr>
<tr>
<td>LAX</td>
<td>to and from</td>
</tr>
<tr>
<td>LAX</td>
<td>to and from</td>
</tr>
<tr>
<td>LAX</td>
<td>to and from</td>
</tr>
<tr>
<td>LON</td>
<td>to and from</td>
</tr>
<tr>
<td>LON</td>
<td>to and from</td>
</tr>
<tr>
<td>LON</td>
<td>to and from</td>
</tr>
<tr>
<td>LON</td>
<td>to and from</td>
</tr>
<tr>
<td>LON</td>
<td>to and from</td>
</tr>
<tr>
<td>LON</td>
<td>to and from</td>
</tr>
<tr>
<td>PAR</td>
<td>to and from</td>
</tr>
<tr>
<td>PAR</td>
<td>to and from</td>
</tr>
<tr>
<td>PAR</td>
<td>to and from</td>
</tr>
<tr>
<td>PAR</td>
<td>to and from</td>
</tr>
<tr>
<td>WAS</td>
<td>to and from</td>
</tr>
</tbody>
</table>

Program Using Cross Join

```sql
proc sql;
  title 'All Possible Connections';
  select f1.Dest, case
    when f1.dest ne ' ' then 'to and from'
  end,
  f2.Dest
  from flights as f1 cross join flights as f2
  where f1.dest < f2.dest
```
Program Description

**Specify a cross join.** Because a cross join is functionally the same as a Cartesian product join, the cross join syntax can be substituted for the conventional join syntax.

```sql
proc sql;
  title 'All Possible Connections';
  select f1.Dest, case
    when f1.dest ne ' ' then 'to and from'
  end,
  f2.Dest
  from flights as f1 cross join flights as f2
  where f1.dest < f2.dest
  order by f1.dest;
```
Example 15: Matching Case Rows and Control Rows

Features: joined-table component
Table names: Match_11

This example uses a table that contains data for a case-control study. Each row contains information for a case or a control. To perform statistical analysis, you need a table with
one row for each case-control pair. PROC SQL joins the table with itself in order to match the cases with their appropriate controls. After the rows are matched, differencing can be performed on the appropriate columns.

The input table “Match_11” on page 440 contains one row for each case and one row for each control. Pair contains a number that associates the case with its control. Low is 0 for the controls and 1 for the cases. The remaining columns contain information about the cases and controls.

```sql
proc sql outobs=10;
  title 'Match_11 Table';
  title2 'First 10 Rows Only';
  select * from match_11;
```

**Figure 7.17  Match_11 Table, First 10 Rows**

![Match_11 Table, First 10 Rows](image)

**Program**

```sql
proc sql;
  create table match as
    select
      one.Low,
      one.Pair,
      (one.lwt - two.lwt) as Lwt_d,
      (one.smoke - two.smoke) as Smoke_d,
      (one.ptd - two.ptd) as Ptd_d,
      (one.ht - two.ht) as Ht_d,
      (one.ui - two.ui) as UI_d
    from match_11 one, match_11 two
    where (one.pair=two.pair and one.low>two.low);
  title 'Differences for Cases and Controls';
```
select *
from match(obs=5);

Program Description

Create the Match table. The SELECT clause specifies the columns for the table Match. SQL expressions in the SELECT clause calculate the differences for the appropriate columns and create new columns.

```
proc sql;
create table match as
  select
    one.Low,
    one.Pair,
    (one.lwt - two.lwt) as Lwt_d,
    (one.smoke - two.smoke) as Smoke_d,
    (one.ptd - two.ptd) as Ptd_d,
    (one.ht - two.ht) as Ht_d,
    (one.ui - two.ui) as UI_d
from match_11 one, match_11 two
where (one.pair=two.pair and one.low>two.low);
```

Specify the type of join and the join criterion. The FROM clause lists the table Match_11 twice. Thus, the table is joined with itself. The WHERE clause returns only the rows for each pair that show the difference when the values for control are subtracted from the values for case.

```
from match_11 one, match_11 two
where (one.pair=two.pair and one.low>two.low);
```

Specify the title.

```
title 'Differences for Cases and Controls';
```

Display the first five rows of the Match table. The SELECT clause selects all the columns from Match. The OBS= data set option limits the printing of the output to five rows.

```
select *
from match(obs=5);
```

Output: Matching Case Rows and Control Rows

```
Output 7.21  Differences for Cases and Controls

<table>
<thead>
<tr>
<th>Low</th>
<th>Pair</th>
<th>Lwt_d</th>
<th>Smoke_d</th>
<th>Ptd_d</th>
<th>Ht_d</th>
<th>UI_d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-34</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>-12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Example 16: Counting Missing Values with a SAS Macro

Features:  COUNT function

Table name:  Survey

This example uses a SAS macro to create columns. The SAS macro is not explained here. See SAS Macro Language: Reference for information about SAS macros.

“Survey” on page 452 contains data from a questionnaire about diet and exercise habits. SAS enables you to use a special notation for missing values. In the EDUC column, the .x notation indicates that the respondent gave an answer that is not valid, and .n indicates that the respondent did not answer the question. A period as a missing value indicates a data entry error.

Program

```
%macro countm(col);
  count(&col) "Valid Responses for &col",
  nmiss(&col) "Missing or NOT VALID Responses for &col",
  count(case
    when &col=.n then "count me"
  end) "Coded as NO ANSWER for &col",
  count(case
    when &col=.x then "count me"
  end) "Coded as NOT VALID answers for &col",
  count(case
    when &col=. then "count me"
  end) "Data Entry Errors for &col"
%mend;

proc sql;
  title 'Counts for Each Type of Missing Response';
  select count(*)  "Total No. of Rows",
         %countm(educ)
  from survey;
```

Program Description

**Count the nonmissing responses.** The COUNTM macro uses the COUNT function to perform various counts for a column. Each COUNT function uses a CASE expression to select the rows to be counted. The first COUNT function uses only the column as an argument to return the number of nonmissing rows.

```
%macro countm(col);
  count(&col) "Valid Responses for &col",
%mend;
```

**Count missing or invalid responses.** The NMISS function returns the number of rows for which the column has any type of missing value: .n, .x, or a period.

```
nmiss(&col) "Missing or NOT VALID Responses for &col",
```
Count the occurrences of various sources of missing or invalid responses. The last three COUNT functions use CASE expressions to count the occurrences of the three notations for missing values. The “count me” character string gives the COUNT function a nonmissing value to count.

```
count(case
    when &col=.n then "count me"
end) "Coded as NO ANSWER for &col",

count(case
    when &col=.x then "count me"
end) "Coded as NOT VALID answers for &col",

count(case
    when &col=. then "count me"
end) "Data Entry Errors for &col"%mend;
```

Use the COUNTM macro to create the columns. The SELECT clause specifies the columns that are in the output. COUNT(*) returns the total number of rows in the table. The COUNTM macro uses the values of the EDUC column to create the columns that are defined in the macro.

```
proc sql;
  title 'Counts for Each Type of Missing Response';
  select count(*)  "Total No. of Rows",
       %countm(educ)
  from survey;
%mend;
```

Output: Counting Missing Values with a SAS Macro

**Output 7.22  Counts for Each Type of Missing Response**

<table>
<thead>
<tr>
<th>Total No. of Rows</th>
<th>Valid Responses for educ</th>
<th>Missing or NOT VALID Responses for educ</th>
<th>Coded as NO ANSWER for educ</th>
<th>Coded as NOT VALID answers for educ</th>
<th>Data Entry Errors for educ</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Overview

This section describes the components that are used in SQL procedure statements. Components are the items in PROC SQL syntax that appear in roman type.

Most components are contained in clauses within the statements. For example, the basic SELECT statement includes the SELECT and FROM clauses, where each clause contains one or more components. Components can also contain other components.

For easy reference, components appear in alphabetical order, and some terms are referred to before they are defined. Use the index or the “See Also” references to refer to other statement or component descriptions that might be helpful.
Dictionary

**BETWEEN Condition**

Selects rows where column values are within a range of values.

### Syntax

```
sql-expression <NOT> BETWEEN sql-expression AND sql-expression
```

### Required Argument

`sql-expression`

See “sql-expression” on page 351.

### Details

- The SQL expressions must be of compatible data types. They must be either all numeric or all character types.
- Because a BETWEEN condition evaluates the boundary values as a range, it is not necessary to specify the smaller quantity first.
- You can use the NOT logical operator to exclude a range of numbers. For example, you can eliminate customer numbers between 1 and 15 (inclusive) so that you can retrieve data on more recently acquired customers.
- PROC SQL supports the same comparison operators that the DATA step supports. For example:
  - `x between 1 and 3`
  - `x between 3 and 1`
  - `1<=x<=3`
  - `x>=1 and x<=3`

---

**BTRIM Function**

Removes blanks or specified characters from the beginning, the end, or both the beginning and end of a character string.

### Syntax

```
BTRIM(<btrim-specification> <btrim-character> FROM> sql-expression)
```

### Required Argument

`sql-expression`

must resolve to a character string or character variable.
**Optional Arguments**

*btrim-specification*

is one of the following:

**LEADING**
removes the blanks or specified characters from the beginning of the character string.

**TRAILING**
removes the blanks or specified characters from the end of the character string.

**BOTH**
removes the blanks or specified characters from both the beginning and the end of the character string.

Default: **BOTH**

*btrim-character*

is a single character that is to be removed from the character string. The default character is a blank.

**Details**

The BTRIM function operates on character strings. BTRIM removes one or more instances of a single character (the value of *btrim-character*) from the beginning, the end, or both the beginning and end of a string, depending whether **LEADING**, **TRAILING**, or **BOTH** is specified. If *btrim-specification* is not specified, then **BOTH** is used. If *btrim-character* is omitted, then blanks are removed.

**Note:** SAS adds trailing blanks to character values that are shorter than the length of the variable. Suppose you have a character variable Z, with length 10, and a value `xxabcxx`. SAS stores the value with three blanks after the last x (for a total length of 10). If you attempt to remove all the x characters with

```
btrim(both 'x' from z)
```

then the result is `abcxx` because PROC SQL sees the trailing characters as blanks, not the x character. In order to remove all the x characters, use this statement.

```
btrim(both 'x' from btrim(z))
```

The inner BTRIM function removes the trailing blanks before passing the value to the outer BTRIM function.

---

**CALCULATED**

Refers to columns already calculated in the SELECT clause.

**Syntax**

```
CALCULATED column-alias
```
**Required Argument**

column-alias

is the name that is assigned to the column in the SELECT clause.

**Details**

CALCULATED enables you to use the results of an expression in the same SELECT clause or in the WHERE clause. It is valid only when used to refer to columns that are calculated in the immediate query expression.

---

**CASE Expression**

Selects result values that satisfy specified conditions.

**Examples:**

“Example 3: Updating Data in a PROC SQL Table” on page 273

“Example 14: Producing All the Possible Combinations of the Values in a Column” on page 304

---

**Syntax**

```sql
CASE <case-operand>
  WHEN when-condition THEN result-expression
  <WHEN when-condition THEN result-expression …>
  <ELSE result-expression>
END
```

**Required Arguments**

when-condition

- When case-operand is specified, when-condition is a shortened SQL expression that assumes case-operand as one of its operands and that resolves to true or false.
- When case-operand is not specified, when-condition is an SQL expression that resolves to true or false.

result-expression

is an SQL expression that resolves to a value.

See “sql-expression” on page 351.

---

**Optional Argument**

case-operand

is a valid SQL expression that resolves to a table column whose values are compared to all the when-conditions.

See “sql-expression” on page 351.
Details

The CASE expression selects values if certain conditions are met. A CASE expression returns a single value that is conditionally evaluated for each row of a table (or view). Use the WHEN-THEN clauses when you want to execute a CASE expression for some but not all of the rows in the table that is being queried or created. An optional ELSE expression gives an alternative action if no THEN expression is executed.

When you omit case-operand and when-condition is evaluated as a Boolean (true or false) value. If when-condition returns a nonzero, nonmissing result, then the WHEN clause is true. If case-operand is specified, then it is compared with when-condition for equality. If case-operand equals when-condition, then the WHEN clause is true.

If the when-condition is true for the row that is being executed, then the result expression that follows THEN is executed. If when-condition is false, then PROC SQL evaluates the next when-condition until they are all evaluated. If every when-condition is false, then PROC SQL executes the ELSE expression, and its result becomes the CASE expression's result. If no ELSE expression is present and every when-condition is false, then the result of the CASE expression is a missing value.

You can use a CASE expression as an item in the SELECT clause and as either operand in an SQL expression.

Example

The following two PROC SQL steps show two equivalent CASE expressions that create a character column with the strings in the THEN clause. The CASE expression in the second PROC SQL step is a shorthand method that is useful when all the comparisons are with the same column.

```sql
proc sql;
   select Name, case
      when Continent = 'North America' then 'Continental U.S.'
      when Continent = 'Oceania' then 'Pacific Islands'
      else 'None'
   end as Region
   from states;
proc sql;
   select Name, case Continent
      when 'North America' then 'Continental U.S.'
      when 'Oceania' then 'Pacific Islands'
      else 'None'
   end as Region
   from states;
```

Note: When you use the shorthand method, the conditions must all be equality tests. That is, they cannot use comparison operators or other types of operators.

COALESCE Function

Returns the first nonmissing value from a list of columns.

Example: “Example 8: Performing an Outer Join” on page 284
Syntax

COALESCE (column-name-1 <, column-name-2, …>)

Required Argument

column-name
See “column-name” on page 323.

Details

COALESCE accepts one or more column names of the same data type. The 
COALESCE function checks the value of each column in the order in which they are 
listed and returns the first nonmissing value. If only one column is listed, the 
COALESCE function returns the value of that column. If all the values of all arguments 
are missing, the COALESCE function returns a missing value.

In some SQL DBMSs, the COALESCE function is called the IFNULL function. For 
more information, see “PROC SQL and the ANSI Standard” on page 387.

Note: If your query contains a large number of COALESCE function calls, it might be 
more efficient to use a natural join instead. See “Natural Joins” on page 336.

column-definition

Defines PROC SQL’s data types and dates

See: “column-modifier” on page 321

Example: “Example 1: Creating a Table and Inserting Data into It” on page 269

Syntax

column data-type <column-modifier(s)>

Required Arguments

column
is a column name.

data-type
is one of the following data types:

CHARACTER | VARCHAR <(width)>
indicates a character column with a column width of width. The default column 
width is eight characters.

INTEGER | SMALLINT
indicates an integer column.

DECIMAL | NUMERIC | FLOAT <(width, ndec)>
indicates a floating-point column with a column width of width and ndec decimal 
places.

REAL | DOUBLE PRECISION
indicates a floating-point column.
DATE
    indicates a date column.

Optional Argument
column-modifier
    See “column-modifier” on page 321.

Details
- SAS supports many but not all of the data types that SQL-based databases support.
- For all the numeric data types (INTEGER, SMALLINT, DECIMAL, NUMERIC, FLOAT, REAL, DOUBLE PRECISION, and DATE), the SQL procedure defaults to the SAS data type NUMERIC. The width and ndec arguments are ignored; PROC SQL creates all numeric columns with the maximum precision allowed by SAS. If you want to create numeric columns that use less storage space, then use the LENGTH statement in the DATA step. The various numeric data type names, along with the width and ndec arguments, are included for compatibility with other SQL software.
- For the character data types (CHARACTER and VARCHAR), the SQL procedure defaults to the SAS data type CHARACTER. The width argument is honored.
- The CHARACTER, INTEGER, and DECIMAL data types can be abbreviated to CHAR, INT, and DEC, respectively.
- A column that is declared with DATE is a SAS numeric variable with a date informat or format. You can use any of the column-modifiers to set the appropriate attributes for the column that is being defined. For more information about dates, see SAS Formats and Informats: Reference.
- When using the VARCHAR2 data type for the Oracle database, or the VARCHAR data type for Greenplum and Aster databases, do not use trailing blanks in column values. Trailing blanks in the VARCHAR2 and VARCHAR data types are considered significant for some databases. Therefore, the results might not be correct, and the generated query is less efficient.

column-modifier
Sets column attributes.

See:
- “column-definition” on page 320
- “SELECT Clause” on page 253

Examples:
- “Example 1: Creating a Table and Inserting Data into It” on page 269
- “Example 2: Creating a Table from a Query’s Result” on page 271

Syntax
column-modifier

Required Argument
column-modifier
    is one of the following:
INFORMAT=\texttt{informatw.d}

specifies a SAS informat to be used when SAS accesses data from a table or view. You can change one permanent informat to another by using the ALTER statement. PROC SQL stores informats in its table definitions so that other SAS procedures and the DATA step can use this information when they reference tables created by PROC SQL.

For more information about informats, see \textit{SAS Formats and Informats: Reference}.

FORMAT=\texttt{formatw.d}

specifies a SAS format for determining how character and numeric values in a column are displayed by the query expression. If the FORMAT= modifier is used in the ALTER, CREATE TABLE, or CREATE VIEW statements, then it specifies the permanent format to be used when SAS displays data from that table or view. You can change one permanent format to another by using the ALTER statement.

For more information about formats, see \textit{SAS Formats and Informats: Reference}.

LABEL='\texttt{label}''

specifies a column label. If the LABEL= modifier is used in the ALTER, CREATE TABLE, or CREATE VIEW statements, then it specifies the permanent label to be used when displaying that column. You can change one permanent label to another by using the ALTER statement.

A label can begin with the following characters: a through z, A through Z, 0 through 9, an underscore (_), or a blank space. If you begin a label with any other character, such as number sign (#), then that character is used as a split character and it splits the label onto the next line wherever it appears. For example:

\begin{verbatim}
select dropout label='Percentage of Students Who Dropped Out' from educ(obs=5);
\end{verbatim}

If a special character must appear as the first character in the output, then precede it with a space or a forward slash (/).

You can omit the LABEL= part of the column-modifier and still specify a label. Be sure to enclose the label in quotation marks, as in this example: \begin{verbatim}
select empname "Names of Employees" from sql.employees;
\end{verbatim}

If an apostrophe must appear in the label, then enter it twice so that SAS reads the apostrophe as a literal. Alternatively, you can use single and double quotation marks alternately (for example, “Date Rec’d”).

LENGTH=\texttt{length}

specifies the length of the column. This column modifier is valid only in the context of a SELECT statement.

TRANSCODE=YES | NO

for character columns, specifies whether values can be transcoded. Use TRANSCODE=NO to suppress transcoding. Note that when you create a table by using the CREATE TABLE AS statement, the transcoding attribute for a given character column in the created table is the same as it is in the source table unless you change it with the TRANSCODE= column modifier. For more information about transcoding, see \textit{SAS National Language Support (NLS): Reference Guide}.

Default YES

Restrictions The TRANSCODE=NO argument is not supported by some SAS Workspace Server clients. In SAS 9.2, if the argument is not
supported, column values with TRANSCODE=NO are replaced (masked) with asterisks (*). Prior to SAS 9.2, column values with TRANSCODE=NO were transcoded.

Suppression of transcoding is not supported for the V6TAPE engine.

Interaction
If the TRANSCODE= attribute is set to NO for any character variable in a table, then PROC CONTENTS prints a transcode column that contains the TRANSCODE= value for each variable in the data set. If all variables in the table are set to the default TRANSCODE= value (YES), then no transcode column is printed.

Details
If you refer to a labeled column in the ORDER BY or GROUP BY clause, then you must use either the column name (not its label), the column's alias, or its ordering integer (for example, ORDER BY 2). For more information about labels, see the section on SAS statements in SAS DATA Step Statements: Reference.

column-name

Specifies the column to select.

See:  "column-modifier" on page 321
 "SELECT Clause" on page 253

Syntax

column-name

Required Argument

column-name

is one of the following:

column

is the name of a column.

table-name.column

is the name of a column in the table table-name.

table-alias.column

is the name of a column in the table that is referenced by table-alias.

view-name.column

is the name of a column in the view view-name.

view-alias.column

is the name of a column in the view that is referenced by view-alias.

Details

A column can be referred to by its name alone if it is the only column by that name in all the tables or views listed in the current query expression. If the same column name exists
in more than one table or view in the query expression, then you must qualify each use of the column name by prefixing a reference to the table that contains it. Consider the following examples:

```sql
SALARY       /* name of the column */
EMP.SALARY   /* EMP is the table or view name */
E.SALARY     /* E is an alias for the table or view that contains the SALARY column */
```

Note: A column name is not the same as a column label. The only valid use of a column label in a query is as a column heading in the .LST output.

### CONNECTION TO
Retrieves and uses DBMS data in a PROC SQL query or view.

**Tip:** You can use CONNECTION TO in the SELECT statement's FROM clause as part of the from-list.

**See:** "Connecting to a DBMS By Using the SQL Procedure Pass-Through Facility" on page 171
SAS/ACCESS documentation

### Syntax

CONNECTION TO `dbms-name (dbms-query)`
CONNECTION TO `alias (dbms-query)`

**Required Arguments**

- `dbms-name` identifies the DBMS that you are using.
- `dbms-query` specifies the query to send to a DBMS. The query uses the DBMS's dynamic SQL. You can use any SQL syntax that the DBMS understands, even if that syntax is not valid for PROC SQL. For example, your DBMS query can contain a semicolon.

The DBMS determines the number of tables that you can join with `dbms-query`. Each CONNECTION TO component counts as one table toward the 256-table PROC SQL limit for joins.

For more information about DBMS queries, see SAS/ACCESS for Relational Databases: Reference.

- `alias` specifies an alias, if one was defined in the CONNECT statement.

### CONTAINS Condition
Tests whether a string is part of a column's value.

**Alias:** ?

**Restriction:** The CONTAINS condition is used only with character operands.
Example:  “Example 8: Performing an Outer Join” on page 284

Syntax

sql-expression <NOT> CONTAINS sql-expression

Required Argument

sql-expression
See “sql-expression” on page 351.

EXISTS Condition

Tests if a subquery returns one or more rows.

See:  “Query Expressions (Subqueries)” on page 354

Syntax

<NOT> EXISTS (query-expression)

Required Argument

query-expression
See “query expression” on page 343.

Details

The EXISTS condition is an operator whose right operand is a subquery. The result of an EXISTS condition is true if the subquery resolves to at least one row. The result of a NOT EXISTS condition is true if the subquery evaluates to zero rows. For example, the following query subsets Proclib.Payroll based on the criteria in the subquery. See “Example 2: Creating a Table from a Query’s Result” on page 271. If the value for Staff.Idnum is on the same row as the value CT in Proclib.Staff, then the matching Idnum in Proclib.Payroll is included in the output. See “Example 4: Joining Two Tables” on page 276. Thus, the query returns all the employees from Proclib.Payroll who live in CT.

```
proc sql;
  select *
  from proclib.payroll p
  where exists (select *
               from proclib.staff s
               where p.idnumber=s.idnum
               and state='CT')
;```

IN Condition

Tests set membership.

Example:  “Example 4: Joining Two Tables” on page 276
Syntax

\[
sql-expression \ <\text{NOT}> \ \text{IN} (query-expression \ | \ constant-1 \ < , \ constant-2 , \ldots )
\]

Required Arguments

sql-expression

See “sql-expression” on page 351.

query-expression

See “query expression” on page 343.

constant

is a number or a quoted character string (or other special notation) that indicates a fixed value. Constants are also called \textit{literals}.

Details

An IN condition tests if the column value that is returned by the SQL expression on the left is a member of the set (of constants or values returned by the query expression) on the right. The IN condition is true if the value of the left-hand operand is in the set of values that are defined by the right-hand operand.

IS Condition

Tests for a missing value.

Example:

“Example 6: Combining Two Tables” on page 280

Syntax

\[
sql-expression \ \text{IS} \ <\text{NOT}> \ \text{NULL} \ | \ \text{MISSING}
\]

Required Argument

sql-expression

See “sql-expression” on page 351.

Details

IS NULL and IS MISSING are predicates that test for a missing value. IS NULL and IS MISSING are used in the WHERE, ON, and HAVING expressions. Each predicate resolves to true if the SQL expression's result is missing and false if it is not missing.

SAS stores a numeric missing value as a period (.) and a character missing value as a blank space. Unlike missing values in some versions of SQL, missing values in SAS always appear first in the collating sequence. Therefore, in Boolean and comparison operations, the following expressions resolve to true in a predicate:

\[
3 > \text{null} \\
-3 > \text{null} \\
0 > \text{null}
\]

The SAS method for evaluating missing values differs from the method of the ANSI standard for SQL. According to the standard, these expressions are NULL. For more information about predicates and operators, see “sql-expression” on page 351.
information about the ANSI standard, see Appendix 2, “PROC SQL and the ANSI Standard,” on page 387.

joined-table

Joins a table with itself or with other tables or views.

Restriction: Joins are limited to 256 tables.

See: “FROM Clause” on page 261

“query expression” on page 343

Examples: “Example 4: Joining Two Tables” on page 276

“Example 8: Performing an Outer Join” on page 284

“Example 10: Joining Three Tables” on page 293

“Example 14: Producing All the Possible Combinations of the Values in a Column” on page 304

“Example 15: Matching Case Rows and Control Rows” on page 310

Syntax

table-name-1 <AS> alias-1>, table-name-2 <AS> alias-2>
<, table-name-3 <AS> alias-3, …>>
table-name-1 <AS> alias-1> <INNER> JOIN table-name-2 <AS> alias-2> ON sql-expression
table-name-1 <AS> alias-1> LEFT JOIN | RIGHT JOIN | FULL JOIN
table-name-2 <AS> alias-2> ON sql-expression
table-name-1 <AS> alias-1> CROSS JOIN table-name-2 <AS> alias-2>
table-name-1 <AS> alias-1> UNION JOIN table-name-2 <AS> alias-2>
table-name-1 <AS> alias-1> NATURAL
<INNER | FULL <OUTER> | LEFT <OUTER> | RIGHT <OUTER>> JOIN table-name-2
<AS> alias-2>

Required Arguments

table-name

can be one of the following:

• the name of a PROC SQL table.
• the name of a SAS view or PROC SQL view.
• a query expression. A query expression in the FROM clause is usually referred to as an inline view. For more information about in-line views, see “FROM Clause” on page 261.
• a connection to a DBMS in the form of the CONNECTION TO component. For more information, see “CONNECTION TO” on page 324.

table-name can be a one-level name, a two-level libref.table name, or a physical pathname that is enclosed in single quotation marks.

Note: If you include parentheses, then be sure to include them in pairs. Parentheses are not valid around comma joins (type).
Optional Argument

alias

specifies an alias for table-name. The AS keyword is optional. For more information, see “Using the CALCULATED Keyword with Column Aliases” on page 148.

Details

Types of Joins

• Inner join. See “Inner Joins” on page 329.
• Outer join. See “Outer Joins” on page 332.
• Cross join. See “Cross Joins” on page 334.
• Union join. See “Union Joins” on page 335.
• Natural join. See “Natural Joins” on page 336.

Joining Tables

When multiple tables, views, or query expressions are listed in the FROM clause, they are processed to form one table. The resulting table contains data from each contributing table. These queries are referred to as joins.

Conceptually, when two tables are specified, each row of table A is matched with all the rows of table B to produce an internal or intermediate table. The number of rows in the intermediate table (Cartesian product) is equal to the product of the number of rows in each of the source tables. The intermediate table becomes the input to the rest of the query in which some of its rows can be eliminated by the WHERE clause or summarized by a summary function.

A common type of join is an equijoin, in which the values from a column in the first table must equal the values of a column in the second table.

Table Limit

PROC SQL can process a maximum of 256 tables for a join. If you are using views in a join, then the number of tables on which the views are based count toward the 256-table limit. Each CONNECTION TO component in the pass-through facility counts as one table.

Specifying the Rows to Be Returned

The WHERE clause or ON clause contains the conditions (SQL expression) under which the rows in the Cartesian product are kept or eliminated in the result table. WHERE is used to select rows from inner joins. ON is used to select rows from inner or outer joins.

The expression is evaluated for each row from each table in the intermediate table described earlier in “Joining Tables” on page 328. The row is considered to be matching if the result of the expression is true (a nonzero, nonmissing value) for that row.

Note: You can follow the ON clause with a WHERE clause to further subset the query result. See “Example 8: Performing an Outer Join” on page 284 for an example.
Table Aliases
Table aliases are used in joins to distinguish the columns of one table from the columns in the other table or tables. A table name or alias must be prefixed to a column name when you are joining tables that have matching column names. For more information about table aliases, see “FROM Clause” on page 261.

Joining a Table with Itself
A single table can be joined with itself to produce more information. These joins are sometimes called reflexive joins. In these joins, the same table is listed twice in the FROM clause. Each instance of the table must have a table alias or you will not be able to distinguish between references to columns in either instance of the table. For examples, see “Example 14: Producing All the Possible Combinations of the Values in a Column” on page 304 and “Example 15: Matching Case Rows and Control Rows” on page 310.

Inner Joins
An inner join returns a result table for all the rows in a table that have one or more matching rows in the other tables, as specified by the SQL expression. Inner joins can be performed on up to 256 tables in the same query expression.

You can perform an inner join by using a list of table-names separated by commas or by using the INNER, JOIN, and ON keywords.

The Lefttab and Righttab tables are used to illustrate this type of join:

```sql
data lefttab;
  input Continent $ Export $ Country $;
  datalines;
  NA   wheat Canada
  EUR  corn  France
  EUR  rice  Italy
  AFR  oil   Egypt
;  
data righttab;
  input Continent $ Export $ Country $;
  datalines;
  NA   sugar USA
  EUR  corn  Spain
  EUR  beets Belgium
  ASIA rice  Vietnam
;  
proc sql;
  title 'Left Table - Lefttab';
  select * from lefttab;
  
  title 'Right Table - Righttab';
  select * from righttab;
```
The following example joins the Lefttab and Righttab tables to get the Cartesian product of the two tables. The Cartesian product is the result of combining every row from one table with every row from another table. You get the Cartesian product when you join two tables and do not subset them with a WHERE clause or ON clause.

```
proc sql;
  title 'The Cartesian Product of';
  title2 'Lefttab and Righttab';
  select *
    from lefttab, righttab;
```

---

**Left Table - Lefttab**

<table>
<thead>
<tr>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
</tr>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
</tr>
</tbody>
</table>

---

**Right Table - Righttab**

<table>
<thead>
<tr>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
</tbody>
</table>
The Cartesian Product of Lefttab and Righttab

<table>
<thead>
<tr>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
</tbody>
</table>

The Lefttab and Righttab tables can be joined by listing the table names in the FROM clause. The following query represents an equijoin because the values of Continent from each table are matched. The column names are prefixed with the table aliases so that the correct columns can be selected.

```
proc sql;
  title 'Inner Join';
  select *
    from lefttab as l, righttab as r
  where l.continent=r.continent;
```
Inner Join

The following PROC SQL step is equivalent to the previous one and shows how to write an equijoin using the INNER JOIN and ON keywords.

```sql
proc sql;
  title 'Inner Join';
  select *
    from lefttab as l inner join
    righttab as r
    on l.continent=r.continent;
```

### See Also

#### Examples

- “Example 4: Joining Two Tables” on page 276
- “Example 14: Producing All the Possible Combinations of the Values in a Column” on page 304
- “Example 15: Matching Case Rows and Control Rows” on page 310

#### Outer Joins

Outer joins are inner joins that have been augmented with rows that did not match with any row from the other table in the join. The three types of outer joins are left, right, and full.

A left outer join, specified with the keywords LEFT JOIN and ON, has all the rows from the Cartesian product of the two tables for which the SQL expression is true, plus rows from the first (Lefttab) table that do not match any row in the second (Righttab) table.

```sql
proc sql;
  title 'Left Outer Join';
  select *
    from lefttab as l left join
    righttab as r
    on l.continent=r.continent;
```
Output 8.4 Left Outer Join

**Left Outer Join**

<table>
<thead>
<tr>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>com</td>
<td>France</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>EUR</td>
<td>com</td>
<td>France</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
</tbody>
</table>

A right outer join, specified with the keywords RIGHT JOIN and ON, has all the rows from the Cartesian product of the two tables for which the SQL expression is true, plus rows from the second (Righttab) table that do not match any row in the first (Lefttab) table.

```sql
proc sql;
    title 'Right Outer Join';
    select *
        from lefttab as l right join
            righttab as r
        on l.continent=r.continent;
```

Output 8.5 Right Outer Join

**Right Outer Join**

<table>
<thead>
<tr>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>EUR</td>
<td>com</td>
<td>France</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>com</td>
<td>France</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
</tbody>
</table>

A full outer join, specified with the keywords FULL JOIN and ON, has all the rows from the Cartesian product of the two tables for which the SQL expression is true, plus rows from each table that do not match any row in the other table.

```sql
proc sql;
    title 'Full Outer Join';
    select *
        from lefttab as l full join
            righttab as r
```
on l.continent=r.continent;

**Output 8.6  Full Outer Join**

### Full Outer Join

<table>
<thead>
<tr>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
<th>Continent</th>
<th>Export</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFR</td>
<td>oil</td>
<td>Egypt</td>
<td>ASIA</td>
<td>rice</td>
<td>Vietnam</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>rice</td>
<td>Italy</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
<td>EUR</td>
<td>beets</td>
<td>Belgium</td>
</tr>
<tr>
<td>EUR</td>
<td>corn</td>
<td>France</td>
<td>EUR</td>
<td>corn</td>
<td>Spain</td>
</tr>
<tr>
<td>NA</td>
<td>wheat</td>
<td>Canada</td>
<td>NA</td>
<td>sugar</td>
<td>USA</td>
</tr>
</tbody>
</table>

**See Also**

“Example 8: Performing an Outer Join” on page 284

### Cross Joins

A cross join returns as its result table the product of the two tables.

Using the Lefttab and Righttab example tables, the following program demonstrates the cross join:

```sql
proc sql;
  title 'Cross Join';
  select *
     from lefttab as l cross join
     righttab as r;
```
The cross join is not functionally different from a Cartesian product join. You would get the same result by submitting the following program:

```sql
proc sql;
    select *
    from lefttab, righttab;
```

Do not use an ON clause with a cross join. An ON clause will cause a cross join to fail. However, you can use a WHERE clause to subset the output.

### Union Joins
A union join returns a union of the columns of both tables. The union join places in the results all rows with their respective column values from each input table. Columns that do not exist in one table will have null (missing) values for those rows in the result table. The following example demonstrates a union join.

```sql
proc sql;
    title 'Union Join';
    select *
    from lefttab union join righttab;
```
Union Join

Using a union join is similar to concatenating tables with the OUTER UNION set operator. For more information, see “query expression” on page 343.

Do not use an ON clause with a union join. An ON clause will cause a union join to fail.

Natural Joins

A natural join selects rows from two tables that have equal values in columns that share the same name and the same type. An error results if two columns have the same name but different types. If join-specification is omitted when specifying a natural join, then INNER is implied. If no like columns are found, then a cross join is performed.

The following examples use these two tables:

```sql
data table1;
  input x y z;
  datalines;
  1 2 3
  2 1 8
  6 5 4
  2 5 6
;?

data table2;
  input x b z;
  datalines;
  1 5 3
  3 5 4
  2 7 8
  6 0 4
;?

proc sql;
  title 'Table1';
  select * from table1;
```
The following program demonstrates a natural inner join.

```sql
proc sql;
  title 'Natural Inner Join';
  select *
  from table1 natural join table2;
quit;
```

**Output 8.10  Natural Inner Join**

The following program demonstrates a natural left outer join.

```sql
proc sql;
  title 'Natural Left Outer Join';
  select *
  from table1 natural join table2;
quit;
```

**Output 8.11  Natural Left Outer Join**
NATURAL LEFT OUTER JOIN

<table>
<thead>
<tr>
<th>x</th>
<th>z</th>
<th>b</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>.</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Do not use an ON clause with a natural join. An ON clause will cause a natural join to fail. When using a natural join, an ON clause is implied, matching all like columns.

JOINING MORE THAN TWO TABLES

Inner joins are usually performed on two or three tables, but they can be performed on up to 256 tables in PROC SQL. You can combine several joins of the same or different types as shown in the following code lines:

```
a natural join b natural join c
```

```
a natural join b cross join c
```

You can also use parentheses to group joins together and control what joins happen in what order as shown in the following examples:

```
(a, b) left join c on a.X=c.Y
```

```
a left join (b full join c on b.Z=c.Z) on a.Y=b.Y
```

Note: Commutative behavior varies depending on the type of join that is performed.

A join on three tables is described here to explain how and why the relationships work among the tables.

In a three-way join, the SQL expression consists of two conditions: one condition relates the first table to the second table; and the other condition relates the second table to the third table. It is possible to break this example into stages. You could perform a two-way join to create a temporary table and then you could join the temporary table with the third one. However, PROC SQL can do it all in one step as shown in the next example. The final table would be the same in both cases.

The example shows the joining of three tables: Comm, Price, and Amount. To calculate the total revenue from exports for each country, you need to multiply the amount exported (Amount table) by the price of each unit (Price table), and you must know the commodity that each country exports (Comm table).

```
data comm;
   input Continent $ Export $ Country $ ;
datalines;
NA wheat Canada
EUR corn France
EUR rice Italy
```
data price;
   input Export $ Price;
   datalines;
rice 3.56
corn 3.45
oil  18
wheat 2.98
;

data amount;
   input Country $ Quantity;
   datalines;
Canada 16000
France 2400
Italy  500
Egypt 10000
;

proc sql;
   title 'Comm Table';
   select * from comm;
   title 'Price Table';
   select * from price;
   title 'Amount Table';
   select * from amount;
**Source for Joining More Than Two Tables**

```sql
proc sql;
  title 'Total Export Revenue';
    as Total
  from comm as c JOIN price as p
    on (c.export=p.export)
    JOIN amount as a
    on (c.country=a.country);
quit;
```
Comparison of Joins and Subqueries

You can often use a subquery or a join to get the same result. However, it is often more efficient to use a join if the outer query and the subquery do not return duplicate rows. For example, the following queries produce the same result. The second query is more efficient:

```
proc sql;
  select IDNumber, Birth
  from proclib.payroll
  where IDNumber in (select idnum
                    from proclib.staff
                    where lname like 'B%');
```

```
proc sql;
  select  p.IDNumber, p.Birth
  from proclib.payroll p, proclib.staff s
  where p.idnumber=s.idnum
    and s.lname like 'B%';
```

*Note:* Proclib.Payroll is shown in “Example 2: Creating a Table from a Query’s Result” on page 271.

LIKE Condition

Tests for a matching pattern.

**Syntax**

```
sql-expression <NOT> LIKE sql-expression <ESCAPE character-expression>
```

**Required Arguments**

`sql-expression`

See “`sql-expression`” on page 351.
character-expression

is an SQL expression that evaluates to a single character. The operands of character-expression must be character or string literals.

Note: If you use an ESCAPE clause, then the pattern-matching specification must be a quoted string or quoted concatenated string; it cannot contain column names.

Details

The LIKE condition selects rows by comparing character strings with a pattern-matching specification. It resolves to true and displays the matched strings if the left operand matches the pattern specified by the right operand. The ESCAPE clause is used to search for literal instances of the percent (%) and underscore (_) characters, which are usually used for pattern matching.

Patterns for Searching

Patterns consist of three classes of characters:

- underscore (_)
  - matches any single character.
- percent sign (%)
  - matches any sequence of zero or more characters.
- any other character
  - matches that character.

These patterns can appear before, after, or on both sides of characters that you want to match. The LIKE condition is case-sensitive.

The following list uses these values: Smith, Smooth, Smothers, Smart, and Smuggle.

'Sm%'
- matches Smith, Smooth, Smothers, Smart, Smuggle.

'%th'
- matches Smith, Smooth.

'S__gg%'
- matches Smuggle.

'S_o'
- matches a three-letter word, so it has no matches here.

'S_o%
- matches Smooth, Smothers.

'S%th'
- matches Smith, Smooth.

'Z'
- matches the single, uppercase character Z only, so it has no matches here.

Searching for Literal % and _

Because the % and _ characters have special meaning in the context of the LIKE condition, you must use the ESCAPE clause to search for these character literals in the input character string.

These examples use the values app, a_, a__, bbaa1, and ba_1.
• The condition like 'a_%' matches app, a_%, and a__, because the underscore (_ in the search pattern matches any single character (including the underscore), and the percent (%) in the search pattern matches zero or more characters, including '%', and '_'.

• The condition like 'a_^%' escape '^' matches only a_%, because the escape character (^) specifies that the pattern search for a literal '%'.

• The condition like 'a_% escape '_' matches none of the values, because the escape character (_) specifies that the pattern search for an 'a' followed by a literal '%', which does not apply to any of these values.

**Searching for Mixed-Case Strings**

To search for mixed-case strings, use the UPCASE function to make all the names uppercase before entering the LIKE condition:

```sql
upcase(name) like 'SM%';
```

*Note:* When you are using the % character, be aware of the effect of trailing blanks. You might have to use the TRIM function to remove trailing blanks in order to match values.

---

**LOWER Function**

Converts the case of a character string to lowercase.

See: "UPPER Function" on page 369

**Syntax**

```sql
LOWER (sql-expression)
```

**Required Argument**

`sql-expression`

See "sql-expression" on page 351.

Requirement `sql-expression` must resolve to a character string.

**Details**

The LOWER function operates on character strings. LOWER changes the case of its argument to all lowercase.

*Note:* The LOWER function is provided for compatibility with the ANSI SQL standard. You can also use the SAS function LOWCASE.

---

**query expression**

Retrieves data from tables.

See: "In-Line Views" on page 262

"Query Expressions (Subqueries)" on page 354
Syntax

table-expression-1 <set-operator table-expression-2> <set-operator table-expression-3> ...

Required Argument

table-expression

See “table-expression” on page 368.

Optional Argument

set-operator

is one of the following:

INTERSECT <CORRESPONDING> <ALL>
OUTER UNION <CORRESPONDING>
UNION <CORRESPONDING> <ALL>
EXCEPT <CORRESPONDING> <ALL>

Details

Query Expressions and Table Expressions

A query expression is one or more table expressions. Multiple table expressions are linked by set operators. The following figure illustrates the relationship between table expressions and query expressions.

Set Operators

PROC SQL provides these set operators:

OUTER UNION
concatenates the query results.

UNION
produces all unique rows from both queries.

EXCEPT
produces rows that are part of the first query only.

INTERSECT
produces rows that are common to both query results.
A query expression with set operators is evaluated as follows.

- Each table expression is evaluated to produce an (internal) intermediate result table.
- Each intermediate result table then becomes an operand linked with a set operator to form an expression (for example, A UNION B).
- If the query expression involves more than two table expressions, then the result from the first two becomes an operand for the next set operator and operand, such as (A UNION B) EXCEPT C, ((A UNION B) EXCEPT C) INTERSECT D, and so on.
- Evaluating a query expression produces a single output table.

Set operators follow this order of precedence unless they are overridden by parentheses in the expressions: INTERSECT is evaluated first. OUTER UNION, UNION, and EXCEPT have the same level of precedence.

PROC SQL performs set operations even if the tables or views that are referred to in the table expressions do not have the same number of columns. The reason for this behavior is that the ANSI standard for SQL requires that tables or views that are involved in a set operation have the same number of columns and that the columns have matching data types. If a set operation is performed on a table or view that has fewer columns than the one or ones with which it is being linked, then PROC SQL extends the table or view with fewer columns by creating columns with missing values of the appropriate data type. This temporary alteration enables the set operation to be performed correctly.

**CORRESPONDING (CORR) Keyword**

The CORRESPONDING keyword is used only when a set operator is specified. CORR causes PROC SQL to match the columns in table expressions by name and not by ordinal position. Columns that do not match by name are excluded from the result table, except for the OUTER UNION operator. See “OUTER UNION” on page 345.

For example, when performing a set operation on two table expressions, PROC SQL matches the first specified column-name (listed in the SELECT clause) from one table expression with the first specified column-name from the other. If CORR is omitted, then PROC SQL matches the columns by ordinal position.

**ALL Keyword**

The set operators automatically eliminate duplicate rows from their output tables. The optional ALL keyword preserves the duplicate rows, reduces the execution by one step, and thereby improves the query expression's performance. You use it when you want to display all the rows resulting from the table expressions, rather than just the unique rows. The ALL keyword is used only when a set operator is also specified.

**OUTER UNION**

Performing an OUTER UNION is very similar to performing the SAS DATA step with a SET statement. The OUTER UNION concatenates the intermediate results from the table expressions. Thus, the result table for the query expression contains all the rows produced by the first table expression followed by all the rows produced by the second table expression. Columns with the same name are in separate columns in the result table.

For example, the following query expression concatenates the ME1 and ME2 tables but does not overlay like-named columns. Output 8.15 on page 347 shows the result.

```sas
data me1;
  input IDnum $ Jobcode $ Salary Bonus;
datalines;
```
1400   ME1         29769  587
1403   ME1         28072  342
1120   ME1         28619  986
1120   ME1         28619  986

;

data me2;
  input IDnum $ Jobcode $ Salary;
datalines;
1653   ME2         35108
1782   ME2         35345
1244   ME2         36925
;

proc sql ;
  title 'ME1';
  select * from me1;
  title 'ME2';
  select * from me2;

Output 8.14  ME1 and ME2 Tables

<table>
<thead>
<tr>
<th>IDnum</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>ME1</td>
<td>29769</td>
<td>587</td>
</tr>
<tr>
<td>1403</td>
<td>ME1</td>
<td>28072</td>
<td>342</td>
</tr>
<tr>
<td>1120</td>
<td>ME1</td>
<td>28619</td>
<td>986</td>
</tr>
<tr>
<td>1120</td>
<td>ME1</td>
<td>28619</td>
<td>986</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDnum</th>
<th>Jobcode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1653</td>
<td>ME2</td>
<td>35108</td>
</tr>
<tr>
<td>1782</td>
<td>ME2</td>
<td>35345</td>
</tr>
<tr>
<td>1244</td>
<td>ME2</td>
<td>36925</td>
</tr>
</tbody>
</table>

proc sql;
  title 'ME1 and ME2: OUTER UNION';
  select *
    from me1
  outer union
  select *
    from me2;
Concatenating tables with the OUTER UNION set operator is similar to performing a union join. For more information, see “Union Joins” on page 335.

To overlay columns with the same name, use the CORRESPONDING keyword.

```
proc sql;
  title 'ME1 and ME2: OUTER UNION CORRESPONDING';
  select *
    from me1
  outer union corr
    select *
    from me2;
```

In the resulting concatenated table, notice the following:

- OUTER UNION CORRESPONDING retains all nonmatching columns.
• For columns with the same name, if a value is missing from the result of the first table expression, then the value in that column from the second table expression is inserted.

• The ALL keyword is not used with OUTER UNION because this operator’s default action is to include all rows in a result table. Thus, both rows from the table ME1 where IDnum is 1120 appear in the output.

**UNION**

The UNION operator produces a table that contains all the unique rows that result from both table expressions. That is, the output table contains rows produced by the first table expression, the second table expression, or both.

Columns are appended by position in the tables, regardless of the column names. However, the data type of the corresponding columns must match or the union will not occur. PROC SQL issues a warning message and stops executing.

The names of the columns in the output table are the names of the columns from the first table expression unless a column (such as an expression) has no name in the first table expression. In such a case, the name of that column in the output table is the name of the respective column in the second table expression.

In the following example, PROC SQL combines the two tables:

```sql
proc sql;
  title 'ME1 and ME2: UNION';
  select *
    from me1
  union
  select *
    from me2;
```

**Output 8.17  Union of ME1 and ME2 Tables**

**ME1 and ME2: UNION**

<table>
<thead>
<tr>
<th>IDnum</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1120</td>
<td>ME1</td>
<td>28619</td>
<td>986</td>
</tr>
<tr>
<td>1244</td>
<td>ME2</td>
<td>36925</td>
<td>.</td>
</tr>
<tr>
<td>1400</td>
<td>ME1</td>
<td>29769</td>
<td>887</td>
</tr>
<tr>
<td>1403</td>
<td>ME1</td>
<td>28072</td>
<td>342</td>
</tr>
<tr>
<td>1653</td>
<td>ME2</td>
<td>35108</td>
<td>.</td>
</tr>
<tr>
<td>1782</td>
<td>ME2</td>
<td>35345</td>
<td>.</td>
</tr>
</tbody>
</table>

In the following example, ALL includes the duplicate row from ME1. In addition, ALL changes the sorting by specifying that PROC SQL make one pass only. Thus, the values from ME2 are simply appended to the values from ME1.

```sql
proc sql;
  title 'ME1 and ME2: UNION ALL';
  select *
    from me1
  union all
```
select *
    from me2;

Output 8.18  Union All

ME1 and ME2: UNION ALL

<table>
<thead>
<tr>
<th>IDnum</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>ME1</td>
<td>29769</td>
<td>587</td>
</tr>
<tr>
<td>1403</td>
<td>ME1</td>
<td>28072</td>
<td>342</td>
</tr>
<tr>
<td>1120</td>
<td>ME1</td>
<td>28619</td>
<td>986</td>
</tr>
<tr>
<td>1120</td>
<td>ME1</td>
<td>28619</td>
<td>986</td>
</tr>
<tr>
<td>1653</td>
<td>ME2</td>
<td>35108</td>
<td>.</td>
</tr>
<tr>
<td>1782</td>
<td>ME2</td>
<td>35345</td>
<td>.</td>
</tr>
<tr>
<td>1244</td>
<td>ME2</td>
<td>36925</td>
<td>.</td>
</tr>
</tbody>
</table>

See “Example 6: Combining Two Tables” on page 280 for another example.

EXCEPT

The EXCEPT operator produces (from the first table expression) an output table that has unique rows that are not in the second table expression. If the intermediate result from the first table expression has at least one occurrence of a row that is not in the intermediate result of the second table expression, then that row (from the first table expression) is included in the result table.

In the following example, the In_USA table contains flights to cities within and outside the USA. The Out_USA table contains flights only to cities outside the USA.

data in_usa;
    input Flight $ Dest $;
    datalines;
145 ORD
156 WAS
188 LAX
193 FRA
207 LON
;
data out_USA;
    input Flight $ Dest $;
    datalines;
193 FRA
207 LON
311 SJA
;
proc sql;
    title 'In_USA';
    select * from in_usa;
    title 'Out_USA';
    select * from out_usa;
This example returns only the rows from In_USA that are not also in Out_USA:

```sql
proc sql;
  title 'Flights from In_USA Only';
  select * from in_usa
  except
  select * from out_usa;
```

**Output 8.20 Flights from In_USA Only**

### Flights from In_USA Only

<table>
<thead>
<tr>
<th>Flight</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>ORD</td>
</tr>
<tr>
<td>156</td>
<td>WAS</td>
</tr>
<tr>
<td>188</td>
<td>LAX</td>
</tr>
<tr>
<td>193</td>
<td>FRA</td>
</tr>
<tr>
<td>207</td>
<td>LON</td>
</tr>
</tbody>
</table>

---

**INTERSECT**

The INTERSECT operator produces an output table that has rows that are common to both tables. For example, using the In_USA and Out_USA tables shown above, the following example returns rows that are in both tables:

```sql
proc sql;
```

**Source Tables for Except Examples**

### In_USA

<table>
<thead>
<tr>
<th>Flight</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>ORD</td>
</tr>
<tr>
<td>156</td>
<td>WAS</td>
</tr>
<tr>
<td>188</td>
<td>LAX</td>
</tr>
<tr>
<td>193</td>
<td>FRA</td>
</tr>
<tr>
<td>207</td>
<td>LON</td>
</tr>
</tbody>
</table>

### Out_USA

<table>
<thead>
<tr>
<th>Flight</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>FRA</td>
</tr>
<tr>
<td>207</td>
<td>LON</td>
</tr>
<tr>
<td>311</td>
<td>SJA</td>
</tr>
</tbody>
</table>
title ‘Flights from Both In_USA and Out_USA’;
select * from in_usa
intersect
select * from out_usa;

Output 8.21  Flights from Both In_USA and Out_USA

<table>
<thead>
<tr>
<th>Flight</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>FRA</td>
</tr>
<tr>
<td>207</td>
<td>LON</td>
</tr>
</tbody>
</table>

sql-expression

Produces a value from a sequence of operands and operators.

Syntax

operand operator operand

Required Arguments

operand

is one of the following:

• a constant, which is a number or a quoted character string (or other special notation) that indicates a fixed value. Constants are also called literals. Constants are described in SAS Functions and CALL Routines: Reference.

• a column-name, which is described in “column-name” on page 323.

• a CASE expression, which is described in “CASE Expression” on page 318.

• any supported SAS function. PROC SQL supports many of the functions available to the SAS DATA step. Some of the functions that are not supported are the variable information functions, functions that work with arrays of data, and functions that operate on rows other than the current row. Other SQL databases support their own sets of functions. Functions are described in the SAS Functions and CALL Routines: Reference.

• any functions, except those with array elements, that are created with PROC FCMP.

• the ANSI SQL functions COALESCE, BTRIM, LOWER, UPPER, and SUBSTRING.

• a summary-function, which is described in “summary-function” on page 360.

• a query expression, which is described in “query expression” on page 343.

• the USER literal, which references the user ID of the person who submitted the program. The user ID that is returned is operating environment-dependent, but
PROC SQL uses the same value that the &SYSJOBID macro variable has on the operating environment.

**operator**

See “Operators and the Order of Evaluation” on page 352.

**Note:** SAS functions, including summary functions, can stand alone as SQL expressions. For example:

```
select min(x) from table;
```

```
select scan(y,4) from table;
```

**Details**

**SAS Functions**

PROC SQL supports many of the functions available to the SAS DATA step. Some of the functions that are not supported are the variable information functions and functions that work with arrays of data. Other SQL databases support their own sets of functions. For example, the SCAN function is used in the following query:

```
select style, scan(street,1) format=$15.
from houses;
```

**Note:** SAS DATA step functions that are used in PROC SQL must follow the ANSI SQL standard guidelines and the ISO SQL standard guidelines for arguments. SAS DATA step functions cannot have empty arguments. Some SAS function arguments might not be supported when used in PROC SQL. For example, the DATA step INPUT function might support “?” and “??”. The INPUT function in PROC SQL supports only “?”. If “??” is used in PROC SQL, an error occurs.

PROC SQL supports any user-written functions except functions with array elements that are created using “FCMP Procedure” in Base SAS Procedures Guide. For complete documentation of SAS functions, see the SAS Functions and CALL Routines: Reference. Summary functions are also SAS functions. For more information, see “summary-function” on page 360.

**USER Literal**

USER can be specified in a view definition. For example, you can create a view that restricts access to the views in the user's department. Note that the USER literal value is stored in uppercase, so it is advisable to use the UPCASE function when comparing to this value:

```
create view myemp as
    select * from dept12.employees
    where upcase(manager)=user;
```

This view produces a different set of employee information for each manager who references it.

**Operators and the Order of Evaluation**

The order in which operations are evaluated is the same as in the DATA step with this one exception: NOT is grouped with the logical operators AND and OR in PROC SQL; in the DATA step, NOT is grouped with the unary plus and minus signs.

Unlike missing values in some versions of SQL, missing values in SAS always appear first in the collating sequence. Therefore, in Boolean and comparison operations, the following expressions resolve to true in a predicate:
You can use parentheses to group values or to nest mathematical expressions. Parentheses make expressions easier to read and can also be used to change the order of evaluation of the operators. Evaluating expressions with parentheses begins at the deepest level of parentheses and moves outward. For example, SAS evaluates A+B*C as A+(B*C), although you can add parentheses to make it evaluate as (A+B)*C for a different result.

Higher priority operations are performed first: that is, group 0 operators are evaluated before group 5 operators. The following table shows the operators and their order of evaluation, including their priority groups.

### Table 8.1 Operators and Order of Evaluation

<table>
<thead>
<tr>
<th>Group</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( )</td>
<td>forces the expression enclosed to be evaluated first</td>
</tr>
<tr>
<td>1</td>
<td>case-expression</td>
<td>selects result values that satisfy specified conditions</td>
</tr>
<tr>
<td>2</td>
<td>**</td>
<td>raises to a power</td>
</tr>
<tr>
<td></td>
<td>unary +, unary -</td>
<td>indicates a positive or negative number</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>multiplies</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divides</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>adds</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>subtracts</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;NOT&gt; BETWEEN condition</td>
<td>See “BETWEEN Condition” on page 316.</td>
</tr>
<tr>
<td></td>
<td>&lt;NOT&gt; CONTAINS condition</td>
<td>See “CONTAINS Condition” on page 324.</td>
</tr>
<tr>
<td></td>
<td>&lt;NOT&gt; EXISTS condition</td>
<td>See “EXISTS Condition” on page 325.</td>
</tr>
<tr>
<td></td>
<td>&lt;NOT&gt; IN condition</td>
<td>See “IN Condition” on page 325.</td>
</tr>
<tr>
<td></td>
<td>IS &lt;NOT&gt; condition</td>
<td>See “IS Condition” on page 326.</td>
</tr>
<tr>
<td></td>
<td>&lt;NOT&gt; LIKE condition</td>
<td>See “LIKE Condition” on page 341.</td>
</tr>
<tr>
<td>7</td>
<td>=, eq</td>
<td>equals</td>
</tr>
<tr>
<td></td>
<td>¬=, ^=, &lt; &gt;, ne</td>
<td>does not equal</td>
</tr>
<tr>
<td></td>
<td>&gt;, gt</td>
<td>is greater than</td>
</tr>
</tbody>
</table>

sql-expression 353
<table>
<thead>
<tr>
<th>Group</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;, lt</td>
<td>is less than</td>
</tr>
<tr>
<td></td>
<td>&gt;=, ge</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td></td>
<td>&lt;=, le</td>
<td>is less than or equal to</td>
</tr>
<tr>
<td></td>
<td>=*, *</td>
<td>sounds like (use with character operands only). See “Example 12: Retrieving Values with the SOUNDS-LIKE Operator” on page 299.</td>
</tr>
<tr>
<td></td>
<td>eqt</td>
<td>equal to truncated strings (use with character operands only). See “Truncated String Comparison Operators” on page 354.</td>
</tr>
<tr>
<td></td>
<td>gt</td>
<td>greater than truncated strings</td>
</tr>
<tr>
<td></td>
<td>lt</td>
<td>less than truncated strings</td>
</tr>
<tr>
<td></td>
<td>get</td>
<td>greater than or equal to truncated strings</td>
</tr>
<tr>
<td></td>
<td>let</td>
<td>less than or equal to truncated strings</td>
</tr>
<tr>
<td></td>
<td>net</td>
<td>not equal to truncated strings</td>
</tr>
<tr>
<td>8</td>
<td>¬, ^, NOT</td>
<td>indicates logical NOT</td>
</tr>
<tr>
<td>9</td>
<td>&amp;, AND</td>
<td>indicates logical AND</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>indicates logical OR</td>
</tr>
</tbody>
</table>

Symbols for operators might vary, depending on your operating environment. For more information, see “SAS Operators in Expressions” in *SAS Language Reference: Concepts*.

**Truncated String Comparison Operators**
PROC SQL supports truncated string comparison operators. (See Group 7 in Table 8.1 on page 353.) In a truncated string comparison, the comparison is performed after making the strings the same length by truncating the longer string to be the same length as the shorter string. For example, the expression `'TWOSTORY' eqt 'TWO'` is true because the string 'TWOSTORY' is reduced to 'TWO' before the comparison is performed. Note that the truncation is performed internally; neither operand is permanently changed.

Note: Unlike the DATA step, PROC SQL does not support the colon operators (such as =:, >:, and <=:) for truncated string comparisons. Use the alphabetic operators (such as EQT, GTT, and LET).

**Query Expressions (Subqueries)**
A query expression is called a subquery when it is used in a WHERE or HAVING clause. A subquery is a query expression that is nested as part of another query expression. A subquery selects one or more rows from a table based on values in another table.
Depending on the clause that contains it, a subquery can return a single value or multiple values. If more than one subquery is used in a query expression, then the innermost query is evaluated first, then the next innermost query, and so on, moving outward.

PROC SQL allows a subquery (contained in parentheses) at any point in an expression where a simple column value or constant can be used. In this case, a subquery must return a single value, that is, one row with only one column.

Here is an example of a subquery that returns one value. This PROC SQL step subsets the ProcLib.Payroll table based on information in the ProcLib.Staff table. (ProcLib.Payroll is shown in “Example 2: Creating a Table from a Query’s Result” on page 271, and ProcLib.Staff is shown in “Example 4: Joining Two Tables” on page 276.) ProcLib.Payroll contains employee identification numbers (IdNumber) and their salaries (Salary) but does not contain their names. If you want to return only the row from ProcLib.Payroll for one employee, then you can use a subquery that queries the ProcLib.Staff table, which contains the employees’ identification numbers and their names (Lname and Fname).

```sql
proc sql;
  title 'Information for Earl Bowden';
  select *
    from procLib.payroll
  where idnumber=
    (select idnum
     from procLib.staff
     where upcase(Lname)='BOWDEN');
```

**Output 8.22  Query Output – One Value**

**Information for Earl Bowden**

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1403</td>
<td>M</td>
<td>ME1</td>
<td>26072</td>
<td>28JAN69</td>
<td>21DEC91</td>
</tr>
</tbody>
</table>

Subqueries can return multiple values. The following example uses the tables ProcLib.Delay and ProcLib.March. These tables contain information about the same flights and have the Flight column in common. The following subquery returns all the values for Flight in ProcLib.Delay for international flights. The values from the subquery complete the WHERE clause in the outer query. Thus, when the outer query is executed, only the international flights from ProcLib.March are in the output.

```sql
proc sql outobs=5;
  title 'International Flights from';
  title2 'ProcLib.March';
  select Flight, Date, Dest, Boarded
    from procLib.march
  where Flight in
    (select flight
     from procLib.delay
     where destype='International');
```
Sometimes it is helpful to compare a value with a set of values returned by a subquery. The keywords ANY or ALL can be specified before a subquery when the subquery is the right-hand operand of a comparison. If ALL is specified, then the comparison is true only if it is true for all values that are returned by the subquery. If a subquery returns no rows, then the result of an ALL comparison is true for each row of the outer query.

If ANY is specified, then the comparison is true if it is true for any one of the values that are returned by the subquery. If a subquery returns no rows, then the result of an ANY comparison is false for each row of the outer query.

The following example selects all of the employees in Proclib.Payroll who earn more than the highest paid ME3:

```sql
proc sql;
title "Employees who Earn More than";
title2 "All MEs";
select *
from proclib.payroll
where salary > all (select salary
                      from proclib.payroll
                      where jobcode='ME3');
```
Query Output Using ALL Comparison

**Employees who Earn More than All MEs**

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Gender</th>
<th>Jobcode</th>
<th>Salary</th>
<th>Birth</th>
<th>Hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>88606</td>
<td>30MAR61</td>
<td>10FEB81</td>
</tr>
<tr>
<td>1739</td>
<td>M</td>
<td>PT1</td>
<td>66517</td>
<td>25DEC64</td>
<td>27JAN91</td>
</tr>
<tr>
<td>1428</td>
<td>F</td>
<td>PT1</td>
<td>68767</td>
<td>04APR60</td>
<td>16NOV91</td>
</tr>
<tr>
<td>1404</td>
<td>M</td>
<td>PT2</td>
<td>91376</td>
<td>24FEB53</td>
<td>01JAN80</td>
</tr>
<tr>
<td>1935</td>
<td>F</td>
<td>NA2</td>
<td>51081</td>
<td>28MAR54</td>
<td>16OCT81</td>
</tr>
<tr>
<td>1905</td>
<td>M</td>
<td>PT1</td>
<td>65111</td>
<td>16APR72</td>
<td>29MAY92</td>
</tr>
<tr>
<td>1407</td>
<td>M</td>
<td>PT1</td>
<td>68096</td>
<td>23MAR69</td>
<td>18MAR90</td>
</tr>
<tr>
<td>1410</td>
<td>M</td>
<td>PT2</td>
<td>84685</td>
<td>03MAY67</td>
<td>07NOV86</td>
</tr>
<tr>
<td>1439</td>
<td>F</td>
<td>PT1</td>
<td>70736</td>
<td>06MAR64</td>
<td>10SEP90</td>
</tr>
<tr>
<td>1545</td>
<td>M</td>
<td>PT1</td>
<td>66130</td>
<td>12AUG59</td>
<td>29MAY90</td>
</tr>
<tr>
<td>1106</td>
<td>M</td>
<td>PT2</td>
<td>89632</td>
<td>06NOV57</td>
<td>16AUG84</td>
</tr>
<tr>
<td>1442</td>
<td>F</td>
<td>PT2</td>
<td>84536</td>
<td>05SEP66</td>
<td>12APR88</td>
</tr>
<tr>
<td>1417</td>
<td>M</td>
<td>NA2</td>
<td>52270</td>
<td>27JUN64</td>
<td>07MAR89</td>
</tr>
<tr>
<td>1478</td>
<td>M</td>
<td>PT2</td>
<td>94203</td>
<td>09AUG59</td>
<td>24OCT90</td>
</tr>
<tr>
<td>1556</td>
<td>M</td>
<td>PT1</td>
<td>71349</td>
<td>22JUN64</td>
<td>11DEC91</td>
</tr>
<tr>
<td>1352</td>
<td>M</td>
<td>NA2</td>
<td>53798</td>
<td>02DEC60</td>
<td>16OCT86</td>
</tr>
<tr>
<td>1890</td>
<td>M</td>
<td>PT2</td>
<td>91908</td>
<td>20JUL51</td>
<td>25NOV79</td>
</tr>
<tr>
<td>1107</td>
<td>M</td>
<td>PT2</td>
<td>89977</td>
<td>09JUN54</td>
<td>10FEB79</td>
</tr>
<tr>
<td>1830</td>
<td>F</td>
<td>PT2</td>
<td>84471</td>
<td>27MAY57</td>
<td>29JAN83</td>
</tr>
<tr>
<td>1928</td>
<td>M</td>
<td>PT2</td>
<td>89858</td>
<td>16SEP54</td>
<td>13JUL90</td>
</tr>
<tr>
<td>1076</td>
<td>M</td>
<td>PT1</td>
<td>66558</td>
<td>14OCT55</td>
<td>03OCT91</td>
</tr>
</tbody>
</table>

**Note:** See the first item in “Subqueries and Efficiency” on page 358 for a note about efficiency when using ALL.

In order to visually separate a subquery from the rest of the query, you can enclose the subquery in any number of pairs of parentheses.

**Correlated Subqueries**

In a correlated subquery, the WHERE expression in a subquery refers to values in a table in the outer query. The correlated subquery is evaluated for each row in the outer query. With correlated subqueries, PROC SQL executes the subquery and the outer query together.
The following example uses the Proclib.Delay and Proclib.March tables. A DATA step creates Proclib.Delay. See “Proclib.Delay” on page 442. Proclib.March is shown in “Example 14: Producing All the Possible Combinations of the Values in a Column” on page 304. Proclib.Delay has the Flight, Date, Orig, and Dest columns in common with Proclib.March:

```sql
proc sql outobs=5;
  title 'International Flights';
  select *
  from proclib.march
  where 'International' in
    (select destype
     from proclib.delay
     where march.Flight=delay.Flight);
```

The subquery resolves by substituting every value for March.Flight into the subquery's WHERE clause, one row at a time. For example, when March.Flight=\(219\), the subquery resolves as follows:

1. PROC SQL retrieves all the rows from DELAY where Flight=\(219\) and passes their DESTYPE values to the WHERE clause.
   
2. PROC SQL uses the DESTYPE values to complete the WHERE clause:
   ```sql
   where 'International' in
     ('International', 'International', ...)
   ```

3. The WHERE clause checks to determine whether \textbf{International} is in the list. Because it is, all rows from March that have a value of \(219\) for Flight become part of the output.

The following output contains the rows from March for international flights only.

\textit{Output 8.25 Correlated Subquery Output}

**International Flights**

<table>
<thead>
<tr>
<th>flight</th>
<th>date</th>
<th>depart</th>
<th>orig</th>
<th>dest</th>
<th>miles</th>
<th>boarded</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>219</td>
<td>01MAR08</td>
<td>9:31</td>
<td>LGA</td>
<td>LON</td>
<td>3442</td>
<td>198</td>
<td>250</td>
</tr>
<tr>
<td>622</td>
<td>01MAR08</td>
<td>12:19</td>
<td>LGA</td>
<td>FRA</td>
<td>3857</td>
<td>207</td>
<td>250</td>
</tr>
<tr>
<td>132</td>
<td>01MAR08</td>
<td>15:35</td>
<td>LGA</td>
<td>YYZ</td>
<td>366</td>
<td>115</td>
<td>178</td>
</tr>
<tr>
<td>271</td>
<td>01MAR08</td>
<td>13:17</td>
<td>LGA</td>
<td>PAR</td>
<td>3635</td>
<td>138</td>
<td>250</td>
</tr>
<tr>
<td>219</td>
<td>02MAR08</td>
<td>9:31</td>
<td>LGA</td>
<td>LON</td>
<td>3442</td>
<td>147</td>
<td>250</td>
</tr>
</tbody>
</table>

\textit{Subqueries and Efficiency}

- Use the MAX function in a subquery instead of the ALL keyword before the subquery. For example, the following queries produce the same result, but the second query is more efficient:

```sql
proc sql;
  select * from proclib.payroll
  where salary> all(select salary
                     from proclib.payroll
                     group by salary)
```
where jobcode='ME3');

proc sql;
  select * from proclib.payroll
  where salary> (select max(salary)
    from proclib.payroll
    where jobcode='ME3');

• With subqueries, use IN instead of EXISTS when possible. For example, the
  following queries produce the same result, but the second query is usually more
  efficient:
  proc sql;
    select *
    from proclib.payroll p
    where exists (select *
      from staff s
      where p.idnum=s.idnum
      and state='CT');

  proc sql;
    select *
    from proclib.payroll
    where idnum in (select idnum
      from staff
      where state='CT');

---

**SUBSTRING Function**

Returns a part of a character expression.

**Syntax**

SUBSTRING (sql-expression FROM start <FOR length>)

**Required Arguments**

*sql-expression*

See “sql-expression” on page 351.

**Requirement**

*sql-expression* must be a character string.

*start*

is a number (not a variable or column name) that specifies the position, counting
from the left end of the character string, at which to begin extracting the substring.

**Optional Argument**

*length*

is a number (not a variable or column name) that specifies the length of the substring
that is to be extracted.
Details

The SUBSTRING function operates on character strings. SUBSTRING returns a specified part of the input character string, beginning at the position that is specified by start. If length is omitted, then the SUBSTRING function returns all characters from start to the end of the input character string. The values of start and length must be numbers (not variables) and can be positive, negative, or zero.

If start is greater than the length of the input character string, then the SUBSTRING function returns a zero-length string.

If start is less than 1, then the SUBSTRING function begins extraction at the beginning of the input character string.

If length is specified, then the sum of start and length cannot be less than start or an error is returned. If the sum of start and length is greater than the length of the input character string, then the SUBSTRING function returns all characters from start to the end of the input character string. If the sum of start and length is less than 1, then the SUBSTRING function returns a zero-length string.

Note: The SUBSTRING function is provided for compatibility with the ANSI SQL standard. You can also use the SAS function SUBSTR.

summary-function

Performs statistical summary calculations.

Restriction: A summary function cannot appear in an ON clause or a WHERE clause.

See: “GROUP BY Clause” on page 263
     “HAVING Clause” on page 264
     “SELECT Clause” on page 253
     “table-expression” on page 368

Examples: “Example 9: Creating a View from a Query’s Result” on page 290
          “Example 13: Joining Two Tables and Calculating a New Value” on page 301
          “Example 16: Counting Missing Values with a SAS Macro” on page 313

Syntax

summary-function (<DISTINCT | ALL> sql-expression)

Required Arguments

summary-function

is one of the following:

AVG | MEAN
    arithmetic mean or average of values
COUNT | FREQ | N
    number of nonmissing values
CSS
    corrected sum of squares
CV
    coefficient of variation (percent)
MAX
largest value

MEDIAN
the 50th percentile; a value that exceeds half of the sample data values and is exceeded by half of the sample data values

MIN
smallest value

NMISS
number of missing values

PRT
is the two-tailed $p$-value for Student's $t$ statistic, $T$ with $n - 1$ degrees of freedom.

RANGE
range of values

STD
standard deviation

STDERR
standard error of the mean

SUM
sum of values

SUMWGT
sum of the WEIGHT variable values

T
Student's $t$ value for testing the hypothesis that the population mean is zero

USS
uncorrected sum of squares

VAR
variance

For a description and the formulas used for these statistics, see “SAS Elementary Statistics Procedures” in Base SAS Procedures Guide.

DISTINCT
specifies that only the unique values of an SQL expression be used in the calculation.

ALL
specifies that all values of an SQL expression be used in the calculation. If neither DISTINCT nor ALL is specified, then ALL is used.

sql-expression
See “sql-expression” on page 351.

Details

**Summarizing Data**
Summary functions produce a statistical summary of the entire table or view that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group. These functions reduce all the values in each row or column in a table to

---

1 Currently, there is no way to designate a WEIGHT variable for a table in PROC SQL. Thus, each row (or observation) has a weight of 1.
one summarizing or aggregate value. For this reason, these functions are often called aggregate functions. For example, the sum (one value) of a column results from the addition of all the values in the column.

**Counting Rows**

The COUNT function counts rows. COUNT(*) returns the total number of rows in a group or in a table. If you use a column name as an argument to COUNT, then the result is the total number of rows in a group or in a table that have a nonmissing value for that column. If you want to count the unique values in a column, then specify COUNT(DISTINCT column).

If the SELECT clause of a table expression contains one or more summary functions and that table expression resolves to no rows, then the summary function results are missing values. The following are exceptions that return zeros:

- COUNT(*)
- COUNT(<DISTINCT> sql-expression)
- NMISS(<DISTINCT> sql-expression)

For examples, see “Example 9: Creating a View from a Query’s Result” on page 290 and “Example 16: Counting Missing Values with a SAS Macro” on page 313.

**Calculating Statistics Based on the Number of Arguments**

The number of arguments that is specified in a summary function affects how the calculation is performed. If you specify a single argument, then the values in the column are calculated. If you specify multiple arguments, then the arguments or columns that are listed are calculated for each row.

*Note:* When more than one argument is used within an SQL aggregate function, the function is no longer considered to be an SQL aggregate or summary function. If there is a like-named Base SAS function, then PROC SQL executes the Base SAS function, and the results that are returned are based on the values for the current row. If no like-named Base SAS function exists, then an error will occur. For example, if you use multiple arguments for the AVG function, an error will occur because there is no AVG function for Base SAS.

For example, consider calculations on the following table.

```sql
data summary;
  input X Y Z;
  datalines;
  1 3 4
  2 4 5
  8 9 4
  4 5 4
;
proc sql;
  title 'Summary Table';
  select * from summary;
```
If you use one argument in the function, then the calculation is performed on that column only. If you use more than one argument, then the calculation is performed on each row of the specified columns. In the following PROC SQL step, the MIN and MAX functions return the minimum and maximum of the columns that they are used with. The SUM function returns the sum of each row of the columns specified as arguments:

```sql
proc sql;
    select min(x) as Colmin_x,
           min(y) as Colmin_y,
           max(z) as Colmax_z,
           sum(x,y,z) as Rowsum
    from summary;
```

Output 8.26  Summary Functions

### Summary Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

If you use one argument in the function, then the calculation is performed on that column only. If you use more than one argument, then the calculation is performed on each row of the specified columns. In the following PROC SQL step, the MIN and MAX functions return the minimum and maximum of the columns that they are used with. The SUM function returns the sum of each row of the columns specified as arguments:

```sql
proc sql;
    select min(x) as Colmin_x,
           min(y) as Colmin_y,
           max(z) as Colmax_z,
           sum(x,y,z) as Rowsum
    from summary;
```

Output 8.26  Summary Functions

### Summary Table

<table>
<thead>
<tr>
<th>Colmin_x</th>
<th>Colmin_y</th>
<th>Colmax_z</th>
<th>Rowsum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

### Remerging Data

When you use a summary function in a SELECT clause or a HAVING clause, you might see the following message in the SAS log:

```
NOTE: The query requires remerging summary
      statistics back with the original
      data.
```

The process of remerging involves two passes through the data. On the first pass, PROC SQL

- calculates and returns the value of summary functions. It then uses the result to calculate the arithmetic expressions in which the summary function participates.

- groups data according to the GROUP BY clause.

On the second pass, PROC SQL retrieves any additional columns and rows that it needs to show in the output.
To specify that PROC SQL not process queries that use remerging of data, use either the PROC SQL NOREMERGE option or the NOSQLREMERGE system option. If remerging is attempted when the NOMERGE option or the NOSQLREMERGE system option is set, an error is written to the SAS log. For more information, see “REMERGE | NOREMERGE” on page 229 and “SQLREMERGE System Option” on page 383.

The following examples use the Proclib.Payroll table to show when remerging of data is and is not necessary. See “Example 2: Creating a Table from a Query’s Result” on page 271.

The first query requires remerging. The first pass through the data groups the data by Jobcode and resolves the AVG function for each group. However, PROC SQL must make a second pass in order to retrieve the values of IdNumber and Salary.

```
proc sql outobs=10;
  title 'Salary Information';
  title2 '(First 10 Rows Only)';
  select  IdNumber, Jobcode, Salary,
          avg(salary) as AvgSalary
    from proclib.payroll
    group by jobcode;
```

**Output 8.27  Salary Information That Required Remerging**

<table>
<thead>
<tr>
<th>IdNumber</th>
<th>Jobcode</th>
<th>Salary</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1677</td>
<td>BCK</td>
<td>26007</td>
<td>25794.22</td>
</tr>
<tr>
<td>1383</td>
<td>BCK</td>
<td>25823</td>
<td>25794.22</td>
</tr>
<tr>
<td>1389</td>
<td>BCK</td>
<td>25028</td>
<td>25794.22</td>
</tr>
<tr>
<td>1673</td>
<td>BCK</td>
<td>25477</td>
<td>25794.22</td>
</tr>
<tr>
<td>1100</td>
<td>BCK</td>
<td>25004</td>
<td>25794.22</td>
</tr>
<tr>
<td>1834</td>
<td>BCK</td>
<td>26896</td>
<td>25794.22</td>
</tr>
<tr>
<td>1845</td>
<td>BCK</td>
<td>25996</td>
<td>25794.22</td>
</tr>
<tr>
<td>1663</td>
<td>BCK</td>
<td>26452</td>
<td>25794.22</td>
</tr>
<tr>
<td>1704</td>
<td>BCK</td>
<td>25465</td>
<td>25794.22</td>
</tr>
<tr>
<td>1970</td>
<td>FA1</td>
<td>22615</td>
<td>23039.36</td>
</tr>
</tbody>
</table>

You can change the previous query to return only the average salary for each job code. The following query does not require remerging because the first pass of the data does the summarizing and the grouping. A second pass is not necessary.

```
proc sql outobs=10;
  title 'Average Salary for Each Jobcode';
  select Jobcode, avg(salary) as AvgSalary
    from proclib.payroll
    group by jobcode;
```
Output 8.28  Salary Information That Did Not Require Remerging

Average Salary for Each Jobcode

<table>
<thead>
<tr>
<th>Jobcode</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK</td>
<td>25794.22</td>
</tr>
<tr>
<td>FA1</td>
<td>23039.36</td>
</tr>
<tr>
<td>FA2</td>
<td>27986.88</td>
</tr>
<tr>
<td>FA3</td>
<td>32933.86</td>
</tr>
<tr>
<td>ME1</td>
<td>28500.25</td>
</tr>
<tr>
<td>ME2</td>
<td>35576.86</td>
</tr>
<tr>
<td>ME3</td>
<td>42410.71</td>
</tr>
<tr>
<td>NA1</td>
<td>42032.2</td>
</tr>
<tr>
<td>NA2</td>
<td>52383</td>
</tr>
<tr>
<td>PT1</td>
<td>67908</td>
</tr>
</tbody>
</table>

When you use the HAVING clause, PROC SQL might have to remerge data to resolve the HAVING expression.

First, consider a query that uses HAVING but that does not require remerging. The query groups the data by values of Jobcode, and the result contains one row for each value of Jobcode and summary information for people in each Jobcode. On the first pass, the summary functions provide values for the **Number**, **Average Age**, and **Average Salary** columns. The first pass provides everything that PROC SQL needs to resolve the HAVING clause, so no remerging is necessary.

```
proc sql outobs=10;
title 'Summary Information for Each Jobcode';
title2 '(First 10 Rows Only)';
select Jobcode,
    count(jobcode) as number
        label='Number',
    avg(int((today()-birth)/365.25))
        as avgage format=2.
        label='Average Age',
    avg(salary) as avgsal format=dollar8.
        label='Average Salary'
from proclib.payroll
group by jobcode
having avgage ge 30;
```
In the following query, PROC SQL remerges the data because the HAVING clause uses the Salary column in the comparison and Salary is not in the GROUP BY clause.

```
proc sql outobs=10;
title 'Employees who Earn More than the';
title2 'Average for Their Jobcode';
title3 '(First 10 Rows Only)';
select Jobcode, Salary,
  avg(salary) as AvgSalary
from proclib.payroll
group by jobcode
having salary > AvgSalary;
```
Employees who Earn More than the Average for Their Jobcode
(First 10 Rows Only)

<table>
<thead>
<tr>
<th>Jobcode</th>
<th>Salary</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK</td>
<td>26007</td>
<td>25794.22</td>
</tr>
<tr>
<td>BCK</td>
<td>25823</td>
<td>25794.22</td>
</tr>
<tr>
<td>BCK</td>
<td>26896</td>
<td>25794.22</td>
</tr>
<tr>
<td>BCK</td>
<td>25996</td>
<td>25794.22</td>
</tr>
<tr>
<td>BCK</td>
<td>26452</td>
<td>25794.22</td>
</tr>
<tr>
<td>FA1</td>
<td>23916</td>
<td>23039.36</td>
</tr>
<tr>
<td>FA1</td>
<td>23177</td>
<td>23039.36</td>
</tr>
<tr>
<td>FA1</td>
<td>23644</td>
<td>23039.36</td>
</tr>
<tr>
<td>FA1</td>
<td>23738</td>
<td>23039.36</td>
</tr>
<tr>
<td>FA1</td>
<td>23979</td>
<td>23039.36</td>
</tr>
</tbody>
</table>

Keep in mind that PROC SQL remerges data under these conditions:

- the values returned by a summary function are used in a calculation. For example, the following query returns the values of X and the percentage of the total for each row. On the first pass, PROC SQL computes the sum of X, and on the second pass PROC SQL computes the percentage of the total for each value of X:

```plaintext
data summary;
  input x;
datalines;
32
86
49
49
;
proc sql;
  title 'Percentage of the Total';
  select X, (100*x/sum(X)) as Pct_Total
    from summary;
```
the values returned by a summary function are compared to values of a column that is not specified in the GROUP BY clause. For example, the following query uses the Proclib.Payroll table. PROC SQL remerges data because the column Salary is not specified in the GROUP BY clause:

```sql
proc sql;
  select jobcode, salary,
       avg(salary) as avsal
  from proclib.payroll
  group by jobcode
  having salary > avsal;
```

a column from the input table is specified in the SELECT clause and is not specified in the GROUP BY clause. This rule does not refer to columns used as arguments to summary functions in the SELECT clause.

For example, in the following query, the presence of IdNumber in the SELECT clause causes PROC SQL to remerge the data because IdNumber is not involved in grouping or summarizing during the first pass. In order for PROC SQL to retrieve the values for IdNumber, it must make a second pass through the data.

```sql
proc sql;
  select IdNumber, jobcode,
       avg(salary) as avsal
  from proclib.payroll
  group by jobcode;
```

### Syntax

**SELECT** <DISTINCT> object-item-1 < object-item-2, ...>

<INTO :macro-variable-specification-1 <, :macro-variable-specification-2, ...>>

**FROM** from-list

<WHERE sql-expression>

---

**Figure 8.1** Percentage of the Total

<table>
<thead>
<tr>
<th>x</th>
<th>Pct_Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>14.81481</td>
</tr>
<tr>
<td>86</td>
<td>39.81481</td>
</tr>
<tr>
<td>49</td>
<td>22.68519</td>
</tr>
<tr>
<td>49</td>
<td>22.68519</td>
</tr>
</tbody>
</table>
<GROUP BY group-by-item-1 <, group-by-item-2, …>>
<HAVING sql-expression>

Details
A table expression is a SELECT statement. It is the fundamental building block of most SQL procedure statements. You can combine the results of multiple table expressions with set operators, which creates a query expression. Use one ORDER BY clause for an entire query expression. Place a semicolon only at the end of the entire query expression. A query expression is often only one SELECT statement or table expression.

UPPER Function
Converts the case of a character string to uppercase.

See: “LOWER Function” on page 343

Syntax
UPPER (sql-expression)

Required Argument
sql-expression
See “sql-expression” on page 351.

Requirement sql-expression must be a character string.

Details
The UPPER function operates on character strings. UPPER converts the case of its argument to all uppercase.
Part 3

Appendixes

Appendix 1
SQL Macro Variables and System Options

Appendix 2
PROC SQL and the ANSI Standard

Appendix 3
Example Code Shown in Using the SQL Procedure

Appendix 4
Data Sets for Examples in SQL Procedure Reference
Appendix 1

SQL Macro Variables and System Options

Dictionary

**SQLCONSTDATETIME System Option**

Specifies whether the SQL procedure replaces references to the DATE, TIME, DATETIME, and TODAY functions in a query with their equivalent constant values before the query executes.

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Categories:** Files: SAS files
System administration: SQL

**PROC OPTIONS GROUP=** SASFILES
SQL

**Note:** This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.

**Syntax**

`SQLCONSTDATETIME | NOSQLCONSTDATETIME`
Syntax Description

SQLCONSTDATETIME
specifies that the SQL procedure is to replace references to the DATE, TIME, DATETIME, and TODAY functions with their equivalent numeric constant values.

NOSQLCONSTDATETIME
specifies that the SQL procedure is not to replace references to the DATE, TIME, DATETIME, and TODAY functions with their equivalent numeric constant values.

Details

When the SQLCONSTDATETIME system option is set, the SQL procedure evaluates the DATE, TIME, DATETIME, and TODAY functions in a query once and uses those values throughout the query. Computing these values once ensures consistency of results when the functions are used multiple times in a query or when the query executes the functions close to a date or time boundary.

When the NOSQLCONSTDATETIME system option is set, the SQL procedure evaluates these functions in a query each time it processes an observation.

If both the SQLREDUCEPUT system option and the SQLCONSTDATETIME system option are specified, the SQL procedure replaces the DATE, TIME, DATETIME, and TODAY functions with their respective values to determine the PUT function value before the query executes:

```sql
select x from &lib..c where (put(bday, date9.) = put(today(), date9.));
```

Note: The value that is specified in the SQLCONSTDATETIME system option is in effect for all SQL procedure statements, unless the CONSTDATETIME option in the PROC SQL statement is set. The value of the CONSTDATETIME option takes precedence over the SQLCONSTDATETIME system option. However, changing the value of the CONSTDATETIME option does not change the value of the SQLCONSTDATETIME system option.

See Also

- “Improving Query Performance” on page 142

Procedure Statement Options:

- CONSTDATETIME option on page 222

System Options:

- “SQLREDUCEPUT= System Option” on page 379

---

**SQLGENERATION= System Option**

Specifies whether and when SAS procedures generate SQL for in-database processing of source data.

**Valid in:**
- SAS 9.4: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
- SAS Viya: configuration file, SAS invocation, OPTIONS statement, SSV9__OPTIONS environment variable

**Categories:**
- Data Access
- System Administration: Performance
Defaults: SAS 9.4: (NONE DBMS="TERADATA DB2 ORACLE NETEZZA ASTER GREENPLM HADOOP SAPHANA IMPALA HAWQ POSTGRES REDSHIFT SQLSVR VERTICA")
SAS Viya: NONE

Restrictions: Parentheses are required when this option value contains multiple keywords.
The maximum length of the option value is 4096 characters.
For DBMS= and EXCLUDEDB= values, the maximum length of an engine name is eight characters. For the EXCLUDEPROC= value, the maximum length of a procedure name is 16 characters. An engine can appear only once, and a procedure can appear only once for a given engine.
Not all procedures support SQL generation for in-database processing for every engine type. If you specify a setting that is not supported, an error message indicates the level of SQL generation that is not supported. The procedure can then reset to the default so that source table records can be read and processed within SAS. If this is not possible, the procedure ends and sets SYSERR= as needed.
If you are using the Metadata LIBNAME Engine, the only valid SQLGENERATION= modifiers are NONE and DBMS. The engine ignores the DBMS=, EXCLUDEDB=, and EXCLUDEPROC= modifiers.

Requirement: You must specify NONE or DBMS as the primary state.

Interactions: Use this option with such procedures as PROC FREQ to indicate that SQL is generated for in-database processing of DBMS tables through supported SAS/ACCESS engines.
You can specify different SQLGENERATION= values for the DATA= and OUT= data sets by using different LIBNAME statements for each of these data sets.

Data source: Amazon Redshift, Aster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, Hadoop, HAWQ, Impala, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, SAP HANA, Teradata, Vertica

Note: Support for Impala and HAWQ was added in SAS 9.4M3.

Tip: After you set a required value (primary state), you can specify optional values (modifiers).

See: SQLGENERATION= LIBNAME option (includes examples)
"Running In-Database Procedures" in SAS In-Database Products: User’s Guide

Syntax

```
SQLGENERATION=<None | DBMS
    <DBMS='engine1 engine2...enginen'>
    <EXCLUDEDB=engine | 'engine1...enginen'>
    <EXCLUDEPROC='"engine=procl...procn' enginen=\"\"\"\"\"procl...procn'">
```

Required Arguments

NONE

prevents those SAS procedures that are enabled for in-database processing from generating SQL for in-database processing. This is a primary state.
DBMS
allows SAS procedures that are enabled for in-database processing to generate SQL for in-database processing of DBMS tables through supported SAS/ACCESS engines. This is a primary state.

*Note:* As a best practice, run as many calculations in-database as possible. Processing that is run in-database generally results in better performance.

" " resets the value to the default that was shipped.

**Optional Arguments**

**DBMS=**"engine1…engineN"
specifies one or more SAS/ACCESS engines. It modifies the primary state.

**EXCLUDEDB=**"engine1…engineN"
prevents SAS procedures from generating SQL for in-database processing for one or more specified SAS/ACCESS engines.

**EXCLUDEPROC=**"""engine1…procN" engineN="proc1…procN" ""prevents one or more specified SAS procedures from generating SQL for in-database processing for one or more specified SAS/ACCESS engines.

**Details**
Here is how SAS/ACCESS handles precedence between the LIBNAME and system option.

**Table A1.1**  *Precedence of Values for SQLGENERATION= LIBNAME and System Options*

<table>
<thead>
<tr>
<th>LIBNAME Option</th>
<th>PROC EXCLUDE on System Option?</th>
<th>Engine Specified on System Option</th>
<th>Resulting Value</th>
<th>From (option)</th>
</tr>
</thead>
<tbody>
<tr>
<td>not set</td>
<td>yes</td>
<td>NONE</td>
<td>NONE</td>
<td>system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBMS</td>
<td>EXCLUDEPROC</td>
<td></td>
</tr>
<tr>
<td>NONE</td>
<td></td>
<td>NONE</td>
<td>NONE</td>
<td>LIBNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBMS</td>
<td></td>
<td>NONE</td>
<td>EXCLUDEPROC</td>
<td>system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not set</td>
<td>no</td>
<td>NONE</td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBMS</td>
<td>DBMS</td>
<td></td>
</tr>
<tr>
<td>NONE</td>
<td></td>
<td>NONE</td>
<td>NONE</td>
<td>LIBNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBMS</td>
<td></td>
<td>NONE</td>
<td>DBMS</td>
<td></td>
</tr>
</tbody>
</table>
Example

Here is the default that is shipped with the product.

```sas
options sqlgeneration='';
proc options option=sqlgeneration
run;
```

SAS procedures generate SQL for in-database processing for all databases except DB2 in this example.

```sas
options sqlgeneration='';
options sqlgeneration=({DBMS EXCLUDEDB='DB2'});
proc options option=sqlgeneration;
run;
```

In this example, in-database processing occurs only for Teradata. SAS procedures that are run on other databases do not generate SQL for in-database processing.

```sas
options sqlgeneration='';
options SQLGENERATION=(NONE DBMS='Teradata');
proc options option=sqlgeneration;
run;
```

For this example, SAS procedures generate SQL for Teradata and Oracle in-database processing. However, no SQL is generated for PROC1 and PROC2 in Oracle.

```sas
options sqlgeneration='';
options SQLGENERATION=(NONE DBMS='Teradata Oracle'
  EXCLUDEPROC='oracle='proc1 proc2');
proc options option=sqlgeneration;
run;
```

---

**SQLIPONEATTEMPT System Option**

Specifies whether PROC SQL allows an SQL query to continue processing when an implicit pass-through request fails.

<table>
<thead>
<tr>
<th>Valid in:</th>
<th>configuration file, SAS invocation, OPTIONS statement, SAS System Options window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category:</td>
<td>System administration: SQL</td>
</tr>
<tr>
<td>PROC OPTIONS GROUP=</td>
<td>SQL</td>
</tr>
<tr>
<td>Note:</td>
<td>This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.</td>
</tr>
</tbody>
</table>

**Syntax**

```
SQLIPONEATTEMPT | NOSQLIPONEATTEMPT
```

**Syntax Description**

`SQLIPONEATTEMPT`

instructs PROC SQL to fail an implicit pass-through (IP) query if the DBMS that the query or part of the query is being sent to rejects it.
NOSQLIPONEATTEMPT
    has no limitations on how many times PROC SQL can reject a query or part of a
    query that is being sent to a DBMS. This is the default value.

Details

If any part of the query that is being sent to the DBMS cannot be processed successfully
by the DBMS, then the PROC SQL statement fails. If a rejection occurs, then an error is
written to the SAS log.

SQLMAPPUTTO= System Option

Specifies whether the PUT function is mapped to the SAS_PUT function for a database. Optionally, use
this system option to specify an alternative database in which the SAS_PUT function is published.

Valid in: configuration file, SAS invocation, OPTIONS statement
Category: Files: SAS Files
Default: SAS_PUT
Data source: Aster, DB2 under UNIX and PC Hosts, Greenplum, Netezza, Teradata
See: SQL_FUNCTIONS= LIBNAME option, SAS In-Database Products: User’s Guide

Syntax

SQLMAPPUTTO=NONE | SAS_PUT | (database.SAS_PUT)

Syntax Description

NONE
    specifies to PROC SQL that no PUT function mapping is to occur.

SAS_PUT
    specifies that the PUT function be mapped to the SAS_PUT function in the database.

database.SAS_PUT
    specifies the database name where the SAS_PUT function and format definitions
    reside.

    TIP It is not necessary that the format definitions and the SAS_PUT function
    reside in the same database as the one that contains the data that you want to
    format. Use the database.SAS_PUT argument to specify the database where the
    format definitions and the SAS_PUT function are published.

    TIP The database name can be a multilevel name and it can include blanks.

Requirement If you specify a database name, you must enclose the entire argument
in parentheses.

Details

SAS format publishing macros publish the PUT function implementation to the database
as a new function named SAS_PUT. The format publishing macros also publish both
user-defined formats that you create using PROC FORMAT and formats that SAS
supplies. The SAS_PUT function supports the use of SAS formats. You can use it in
SQL queries that SAS submits to the database so that the entire SQL query can be
processed inside the database. You can also use it in conjunction with in-database procedures.

You can use this option with the SQLREDUCEPUT=, SQLREDUCEPUTOBS, and SQLREDUCEPUTVALUES= system options. For more information about these options, see SAS SQL Procedure User’s Guide.

SQLREDUCEPUT= System Option

For the SQL procedure, specifies the engine type to use to optimize a PUT function in a query. The PUT function is replaced with a logically equivalent expression.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Categories:
- Files: SAS files
- System administration: SQL
- System administration: Performance

PROC OPTIONS GROUP= SASFILES
- SQL
- PERFORMANCE

Note: This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.

Syntax

SQLREDUCEPUT= ALL | NONE | DBMS | BASE

Syntax Description

ALL
- specifies to consider the optimization of all PUT functions, regardless of the engine that is used by the query to access the data.

NONE
- specifies to not optimize any PUT function.

DBMS
- specifies to consider the optimization of all PUT functions in a query performed by a SAS/ACCESS engine. This is the default.

Requirement: The first argument to the PUT function must be a variable that is obtained by a table. The table must be accessed using a SAS/ACCESS engine.

BASE
- specifies to consider the optimization of all PUT functions in a query performed by a SAS/ACCESS engine or a Base SAS engine.

Details

If you specify the SQLREDUCEPUT= system option, SAS optimizes the PUT function before the query is executed. If the query also contains a WHERE clause, the evaluation of the WHERE clause is simplified. The following SELECT statements are examples of
queries that are optimized if the SQLREDUCEPUT= option is set to any value other than none:

```sql
select x, y from &lib..b where (PUT(x, abc.) in ('yes', 'no'));
select x from &lib..a where (PUT(x, udfmt.) = trim(left('small')));
```

If both the SQLREDUCEPUT= system option and the SQLCONSTDATETIME system option are specified, PROC SQL replaces the DATE, TIME, DATETIME, and TODAY functions with their respective values to determine the PUT function value before the query executes.

The following two SELECT clauses show the original query and optimized query:

```sql
select x from &lib..c where (put(bday, date9.) = put(today(), date9.));
```

Here, the SELECT clause is optimized.

```sql
select x from &lib..c where (x = '17MAR2011'D);
```

If a query does not contain the PUT function, it is not optimized.

Note: The value that is specified in the SQLREDUCEPUT= system option is in effect for all SQL procedure statements, unless the PROC SQL REDUCEPUT= option is set. The value of the REDUCEPUT= option takes precedence over the SQLREDUCEPUT= system option. However, changing the value of the REDUCEPUT= option does not change the value of the SQLREDUCEPUT= system option.

See Also

- “Improving Query Performance” on page 142

Procedure Statement Options:

- REDUCEPUT= option on page 227

System Options:

- “SQLCONSTDATETIME System Option” on page 373
- “SQLREDUCEPUTOBS= System Option” on page 380

---

**SQLREDUCEPUTOBS= System Option**

For the SQL procedure, when the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, specifies the minimum number of observations that must be in a table for PROC SQL to optimize the PUT function in a query.

**Valid in:**
- configuration file
- SAS invocation
- OPTIONS statement
- SAS System Options window

**Categories:**
- Files: SAS files
- System administration: SQL
- System administration: Performance

**PROC OPTIONS GROUP=**
- SASFILES
- SQL
- PERFORMANCE
Interactions: If the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, conditions for both the SQLREDUCEPUTOBS= and SQLREDUCEPUTVALUES= system options must be met for PROC SQL to optimize the PUT function.

The SQLREDUCEPUTOBS= system option works only for DBMSs that record the number of observations in a table. If your DBMS does not record the number of observations, but you create row counts on your table, the SQLREDUCEPUTOBS= option will work.

Note: This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.

Syntax

SQLREDUCEPUTOBS= n

Syntax Description

n specifies the minimum number of observations that must be in a table for PROC SQL to optimize the PUT function in a query.

Default 0, which indicates that there is no minimum number of observations in a table for PROC SQL to optimize the PUT function.

Range 0–263–1, or approximately 9.2 quintillion

Requirement n must be an integer

Details

For databases that allow implicit pass-through when the row count for a table is not known, PROC SQL allows the PUT function to be optimized in the query, and the query is executed by the database. When the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, PROC SQL considers the values of both the SQLREDUCEPUTVALUES= and SQLREDUCEPUTOBS= system options, and determines whether to optimize the PUT function.

For databases that do not allow implicit pass-through, PROC SQL does not optimize the PUT function, and more of the query is executed by SAS.

See Also

• “Improving Query Performance” on page 142

System Options:

• “SQLREDUCEPUT= System Option” on page 379
• “SQLREDUCEPUTVALUES= System Option” on page 381

SQLREDUCEPUTVALUES= System Option

For the SQL procedure, when the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, specifies the maximum number of SAS format values that can exist in a PUT function expression for PROC SQL to optimize the PUT function in a query.
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Categories: Files: SAS files
System administration: SQL
System administration: Performance

PROC OPTIONS
GROUP=

SASFILES
SQL
PERFORMANCE

Interaction: If the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, conditions for both the SQLREDUCEPUTVALUES= and SQLREDUCEPUTOBS= system options must be met for PROC SQL to optimize the PUT function.

Note: This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.

Syntax

SQLREDUCEPUTVALUES= n

Syntax Description

n specifies the maximum number of SAS format values that can exist in a PUT function expression for PROC SQL to optimize the PUT function in a query.

Default 100

Range 100 – 3,000

Requirement n must be an integer

Interaction If the number of SAS format values in a PUT function expression is greater than this value, PROC SQL does not optimize the PUT function.

Details

Some formats, especially user-defined formats, can contain many format values. Depending on the number of matches for a PUT function expression, the resulting expression can list many format values. If the number of format values becomes too large, query performance can degrade. When the SQLREDUCEPUT= system option is set to DBMS, BASE, or ALL, PROC SQL considers the values of both the SQLREDUCEPUTVALUES= and SQLREDUCEPUTOBS= system options and determines whether to optimize the PUT function.

TIP The value for SQLREDUCEPUTVALUES= is used for each individual optimization. For example, if you have a PUT function in a WHERE clause and another PUT function in a GROUP BY clause, the value of SQLREDUCEPUTVALUES= is applied separately to each clause.

See Also

• “Improving Query Performance” on page 142
SQLREMERGE System Option

Specifies whether PROC SQL can process queries that use remerged data.

**Valid in:**
- configuration file
- SAS invocation
- OPTIONS statement
- SAS System Options window

**Categories:**
- Files: SAS files
- System administration: SQL

**PROC OPTIONS**
- GROUP= SASFILES
- GROUP= SQL

**Note:** This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.

---

**Syntax**

`SQLREMERGE | NOSQLREMERGE`

**Syntax Description**

- `SQLREMERGE` specifies that PROC SQL can process queries that use remerged data.
- `NOSQLREMERGE` specifies that PROC SQL cannot process queries that use remerged data.

**Details**

The remerge feature of PROC SQL makes two passes through a table. Data that is created in the first pass is used in the second pass to complete a query. When the `NOSQLREMERGE` system option is specified, PROC SQL cannot process the remerged data. If remerging is attempted when the `NOSQLREMERGE` system option is specified, an error is written to the SAS log.

**See Also**

- “Improving Query Performance” on page 142

**Procedure Statement Options:**

- REMERGE option on page 229
- “summary-function” on page 360
SQLUNDOPOLICY= System Option

Specifies how PROC SQL handles updated data if errors occur while you are updating data. You can use UNDO_POLICY= to control whether your changes are permanent.

**Valid in:** configuration file, SAS invocation, Options statement

**Categories:** Files: SAS files
System administration: SQL

**PROC OPTIONS**

GROUP= SASFILES
SQL

**Note:** This option can be restricted by a site administrator. For more information, see “Restricted Options” in SAS System Options: Reference.

**Syntax**

SQLUNDOPOLICY=NONE | OPTIONAL | REQUIRED

**Syntax Description**

**NONE**

keeps any updates or inserts.

**OPTIONAL**

reverses any updates or inserts that it can reverse reliably.

**REQUIRED**

reverses all updates or inserts that have been made up to the point of the error. This is the default.

**CAUTION Some UNDO operations cannot be done reliably.** In some cases, the UNDO operation cannot be done reliably. When a change cannot be reversed, PROC SQL issues an error message and does not execute the statement. For example, when a program uses a SAS/ACCESS view or when a SAS data set is accessed through a SAS/SHARE server and is opened with the data set option CNTLLEV=RECORD, you cannot reliably reverse your changes.

**CAUTION Some UNDO operations might not reverse changes.** When multiple transactions are made to the same record, PROC SQL might not reverse a change. PROC SQL issues an error message instead. For example, if one transaction inserts a record and an error occurs, and in another transaction the same record is deleted, an UNDO operation does not reverse the deletion of the record and an error message is issued.

**Details**

The value that is specified in the SQLUNDOPOLICY= system option is in effect for all SQL procedure statements unless the PROC SQL UNDO_POLICY= option is set. The value of the UNDO_POLICY= option takes precedence over the SQLUNDOPOLICY= system option. The RESET statement can be used to set or reset the UNDO_POLICY= option. However, changing the value of the UNDO_POLICY= option does not change
the value of the SQLUNDOPOLICY= system option. After the SQL procedure completes, it reverts to the value of the SQLUNDOPOLICY= system option.

If you are updating a data set using the SAS Scalable Performance Data Engine, you can significantly improve processing performance by setting SQLUNDOPOLICY=NONE. However, ensure that NONE is an appropriate setting for your application.

See Also

Procedure Statement

• UNDO_POLICY on page 230

SYS_SQLSETLIMIT Macro Variable

For the SQL procedure, specifies the maximum number of values that is used to optimize a hash join during DBMS processing.

Syntax

SYS_SQLSETLIMIT= n;

Required Argument

n

specifies the maximum number of values in the IN condition that is passed to the DBMS for processing.

Default 1024

Restriction

The SYS_SQLSETLIMIT macro variable affects only certain hash joins.

Example

%let SYS_SQLSETLIMIT=250;
%let SYS_SQLSETLIMIT=1200;

Details

Hash Join

To optimize performance, the SQL procedure might use a hash join when an index join is eliminated as a possibility. With a hash join, the smaller table is reconfigured in memory as a hash table. PROC SQL sequentially scans the larger table and performs a row-by-row hash lookup against the smaller table to form the result set. A memory-sizing formula determines whether a hash join is used. The formula is based on the PROC SQL UBUFSIZE option, whose default value is 64 KB. On a memory-rich system, you should consider increasing UBUFSIZE to increase the likelihood that a hash join is used.
Appendix 2

PROC SQL and the ANSI Standard

Compliance

PROC SQL follows most of the guidelines set by the American National Standards Institute (ANSI) in its implementation of SQL. However, it is not fully compliant with the current ANSI standard for SQL.2

The SQL research project at SAS has focused primarily on the expressive power of SQL as a query language. Consequently, some of the database features of SQL have not yet been implemented in PROC SQL.

For information about how SQL differs from ANSI SQL, see “Differences in PROC SQL and ANSI SQL” on page 390.

SQL Procedure Enhancements

Reserved Words

PROC SQL reserves very few keywords. And then, only in certain contexts. The ANSI standard reserves all SQL keywords in all contexts. For example, according to the standard, you cannot name a column GROUP because of the keywords GROUP BY.

The following words are reserved in PROC SQL:

- The keyword CASE is always reserved. Its use in the CASE expression (an SQL2 feature) precludes its use as a column name.

  If you have a column named CASE in a table and you want to specify it in a PROC SQL step, then you can use the SAS data set option RENAME= to rename that column for the duration of the query. You can enclose CASE in double quotation marks (“CASE”) and set the PROC SQL option DQUOTE=ANSI.

- The keywords AS, ON, FULL, JOIN, LEFT, FROM, WHEN, WHERE, ORDER, GROUP, RIGHT, INNER, OUTER, UNION, EXCEPT, HAVING, and INTERSECT cannot be used for table aliases. These keywords introduce clauses that appear after a table name. Because the table alias is optional, PROC SQL handles this ambiguity by assuming that any one of these words introduces the corresponding clause and is not the table alias. If you want to use one of these keywords as a table alias, then enclose the keyword in double quotation marks and set the PROC SQL option DQUOTE=ANSI.

- The keyword USER is reserved for the current user ID. If you specify USER in a SELECT statement in conjunction with a CREATE TABLE statement, then the column is created in the table with a temporary column name that is similar to _TEMA001. If you specify USER in a SELECT statement without a CREATE

TABLE statement, then the column is written to the output without a column heading. In either case, the value for the column varies by operating environment, but is typically the user ID of the user who is submitting the program or the value of the &SYSJOBID automatic macro variable.

If you have a column named USER in a table and you want to specify it in a PROC SQL step, then you can use the SAS data set option RENAME= to rename that column for the duration of the query. You can enclose USER in double quotation marks ("USER") and set the PROC SQL option DQUOTE=ANSI.

**Column Modifiers**

PROC SQL supports the SAS INFORMAT=, FORMAT=, and LABEL= modifiers for expressions in the SELECT statement. These modifiers control the format in which output data is displayed and labeled.

**Alternate Collating Sequences**

PROC SQL enables you to specify an alternate collating (sorting) sequence to be used when you specify the ORDER BY clause. For more information about the SORTSEQ= option, see “PROC SQL Statement” on page 221.

**ORDER BY Clause in a View Definition**

PROC SQL permits you to specify an ORDER BY clause in a CREATE VIEW statement. When the view is queried, its data is sorted based on the specified order, unless a query against that view includes a different ORDER BY clause. For more information, see “CREATE VIEW Statement” on page 243.

**CONTAINS Condition**

PROC SQL enables you to test whether a string is part of the value of a column when you specify the CONTAINS condition. For more information, see “CONTAINS Condition” on page 324.

**Inline Views**

The ability to code nested query expressions in the FROM clause is a requirement of the ANSI standard. PROC SQL supports nested coding.

**Outer Joins**

The ability to include columns that both match and do not match in a join expression is a requirement of the ANSI standard. PROC SQL supports this ability.

**Arithmetic Operators**

PROC SQL supports the SAS exponentiation (**) operator. PROC SQL uses the notation <> to mean not equal.

**Orthogonal Expressions**

PROC SQL enables the combination of comparison, Boolean, and algebraic expressions. For example, \((X=3)^7\) yields a value of 7 if \(X=3\) is true because true is defined to be 1. If \(X=3\) is false, then it resolves to 0, and the entire expression yields a value of 0.

PROC SQL permits a subquery in any expression. This feature is required by the ANSI standard. Therefore, you can have a subquery on the left side of a comparison operator in the WHERE expression.
PROC SQL permits you to order and group data by any type of mathematical expression (except a mathematical expression including a summary function) using ORDER BY and GROUP BY clauses. You can group by an expression that appears in the SELECT statement by using the integer that represents the expression’s ordinal position in the SELECT statement. You are not required to select the expression by which you are grouping or ordering. For more information, see “ORDER BY Clause” on page 265 and “GROUP BY Clause” on page 263.

Set Operators
The set operators UNION, INTERSECT, and EXCEPT are required by the ANSI standard. PROC SQL provides these operators and the OUTER UNION operator.

The ANSI standard requires that the tables being operated on have the same number of columns with matching data types. The SQL procedure works on tables that have the same number of columns, and it works on tables that have a different number of columns by creating virtual columns so that a query can evaluate correctly. For more information, see “query expression” on page 343.

Statistical Functions
PROC SQL supports many more summary functions than required by the ANSI standard for SQL.

PROC SQL supports remerging summary function results into the original data of a table. For example, computing the percentage of total is achieved with $100 \times \frac{x}{\text{SUM}(x)}$ in PROC SQL. For more information about summary functions and remerging data, see “summary-function” on page 360.

SAS DATA Step Functions
PROC SQL supports many of the functions available in the SAS DATA step. Some of the functions that are not supported are the variable information functions and functions that work with arrays of data. Other SQL databases support their own sets of functions.

CAUTION:
SAS DATA step functions that are used in PROC SQL must follow the ANSI SQL standard guidelines and the ISO SQL standard guidelines for arguments. SAS DATA step functions cannot have empty arguments.

PROC FCMP Functions
PROC SQL supports any user-written functions except functions with array elements that are created using “FCMP Procedure” in Base SAS Procedures Guide.

The CALCULATED Keyword
The CALCULATED keyword enables you to use the results of an expression in the same SELECT clause or in the WHERE clause. For more information, see “CALCULATED” on page 317 and “Using the CALCULATED Keyword with Column Aliases” on page 148.

SQL Procedure Omissions

COMMIT Statement
The COMMIT statement is not supported.
**ROLLBACK Statement**

The ROLLBACK statement is not supported. The PROC SQL UNDO_POLICY= option or the SQLUNDOPOLICY system option addresses rollback. See the description of the UNDO_POLICY= option in “PROC SQL Statement” on page 221 or in the “SQLUNDOPOLICY= System Option” on page 384.

**Identifiers and Naming Conventions**

In SAS, table names, column names, and aliases are limited to 32 characters and can contain mixed case. For more information about SAS naming conventions, see Base SAS Utilities: Reference. The ANSI standard for SQL allows longer names.

**Granting User Privileges**

The GRANT statement, PRIVILEGES keyword, and authorization-identifier features of SQL are not supported. You might want to use operating environment-specific means of security instead.

**Three-Valued Logic**

ANSI-compatible SQL has three-valued logic. That is, it has special cases for handling comparisons involving NULL values. Any value compared with a NULL value evaluates to NULL.

PROC SQL follows the SAS convention for handling missing values. When numeric NULL values are compared with non-NULL numbers, the NULL values are less than or smaller than all the non-NULL values. When character NULL values are compared with non-NULL characters, the character NULL values are treated as a string of blanks.

**Embedded SQL**

Currently, there is no provision for embedding PROC SQL statements in other SAS programming environments, such as the DATA step or SAS/IML software.

**Column Aliases**

PROC SQL supports uses of aliases that are not supported by the ANSI standard. For more information, see “Using Column Aliases” on page 146.

**Differences in PROC SQL and ANSI SQL**

**Scoping Rules for Names of Tables and Views**

The scoping rules for SAS SQL were developed before the ANSI SQL scoping rules. As a result, there are differences in how PROC SQL handles table and view aliases and correlation names. Here are two examples of how the differences in the scoping rules affect output from PROC SQL.

**Example 1: Matching Correlation Names with Data Sets**

When PROC SQL tries to match a correlation name with a data set, it first attempts to match the correlation name only with an alias. If PROC SQL does not find a match, it attempts to match the correlation name with a data set name that does not have an alias. If this second attempt fails, PROC SQL ignores aliases and tries to match a correlation name with a data set that has a name and an alias associated with it.

Here is a code example that contains a WHERE statement in a NOT EXISTS condition:

```sql
create table work.test as
select DWPCTLG.CUSTOMER_NUMBER_IDENTIFIER,
```
from work.DWPCTLG as DWPCTLG,
work.NSPACCT as NSPACCT
where DWPCTLG.CUSTOMER_NUMBER_IDENTIFIER =
    NSPACCT.BUSINESS_UNIT_ACCOUNT_ID
and DWPCTLG.CATEGORY_CODE = 'G'
and NOT EXISTS ( select *
    from work.DWPCTLG as DWPCTLG_66503
    where DWPCTLG_66503.CUSTOMER_NUMBER_IDENTIFIER =
        DWPCTLG.CUSTOMER_NUMBER_IDENTIFIER
    and STRIP(DWPCTLG_66503.CATEGORY_CODE) <> 'G'
) ;

In this example, the PROC SQL correlation-name-matching rules, except for one
  correlation name, result in the same name matchings as the rules in the ISO/ANSI
  SQL:2012 Standards for correlation name matchings. The exception is the DWPCTLG
  correlation name that appears in the body of the NOT EXISTS condition. In this
  condition, PROC SQL first attempts to match DWPCTLG with a data set alias. The
  work.DWPCTLG data set is the only data set that is part of this condition. It has an alias
  of DWPCTLG_65503. The DWPCTLG correlation name does not match
  DWPCTLG_65503, so PROC SQL attempts to match DWPCTLG with the name of a
  data set in the same scope that does not have an alias. This scope does not contain a data
  set that does not have an alias, so there is no match. Finally, PROC SQL attempts to
  match the correlation name DWPCTLG with the name of a data set in this scope that has
  an alias. The DWPCTLG data set, with an alias of DWPCTLG_65503, is the only data
  set in this NOT EXISTS scope that qualifies. In this case, PROC SQL matches the
  correlation name DWPCTLG with the data set name DWPCTLG. With this match, the
  WHERE clause resolves to this code:

  where DWPCTLG_66503.CUSTOMER_NUMBER_IDENTIFIER =
    DWPCTLG_66503.CUSTOMER_NUMBER_IDENTIFIER
  and STRIP(DWPCTLG_66503.CATEGORY_CODE) <> 'G'

Note the difference in the highlighted string in the second line of code in the above
example and the original example.

The original WHERE statement in the NOT EXISTS condition is optimized to this
statement

  STRIP(DWPCTLG_66503.CATEGORY_CODE) <> 'G'

Note: If PROC SQL used the ISO/ANSI SQL:2012 Standards for correlation name
  matchings, then the reference to CUSTOMER_NUMBER_IDENTIFIER on the right
  side of the equality comparison in the NOT EXISTS condition is matched with the
  DWPCTLG alias in the FROM clause of the top-level SELECT statement in the
  original query.

A query can be modified to ensure that PROC SQL makes the same correlation
matchings as the ISO/ANSI SQL:2012 Standards. For a correlated query such as the
previous one, make sure that the data sets in an outer scope that are associated with
  correlated variables in an inner scope have names or aliases that are different from any
  that appear in the inner scope. For example, the alias for the work.DWPCTLG data set in
  the outer scope of the original query has been changed from DWPCTLG to
  DWPCTLG_X, and the associated reference in the inner scope has also been changed
  from DWPCTLG to DWPCTLG_X. This new alias is different from any data set name
  or alias that appears in the NOT EXISTS condition of this query.

Here is the modified query that produces the same results using PROC SQL rules or
ISO/ANSI SQL:2012 Standards rules for correlation name matchings:
create table work.test as
select DWPCTLG.CUSTOMER_NUMBER_IDENTIFIER,
1 as LMG length = 8
from work.DWPCTLG as DWPCTLG_X,
work.NSPACCT as NSPACCT
where DWPCTLG.CUSTOMER_NUMBER_IDENTIFIER =
NSPACCT.BUSINESS_UNIT_ACCOUNT_ID
and DWPCTLG.CATEGORY_CODE = 'G'
and NOT EXISTS ( select *
from work.DWPCTLG as DWPCTLG_66503
where DWPCTLG_66503.CUSTOMER_NUMBER_IDENTIFIER =
DWPCTLG_X.CUSTOMER_NUMBER_IDENTIFIER
and STRIP(DWPCTLG_66503.CATEGORY_CODE) <> 'G'
);

Example 2: Ambiguous References
This code example has an ambiguous reference because there are two a references in the same scope. A DBMS that is in full compliance with the ANSI SQL standard detects that the a.x reference is ambiguous. PROC SQL does not detect the ambiguous reference, and associates a.x with c.x. It creates a table with one row, when it should not create a table at all.

create table work.test as
select a.ax
from a, c a
where a.ax > 0;
Appendix 3
Example Code Shown in Using the SQL Procedure

This appendix shows the example code for the whole section of “Using the SQL Procedure.” If you copy and paste the code to a SAS editor, the code spacing is preserved if you copy the code from an HTML page.

Introduction to the SQL Procedure

Some output tables restrict the number of observations. To list the entire table, remove the PROC SQL OUTOBS= option.

```sas
/**************************************************************************
 Output 1.1 Sample SQL Output
**************************************************************************/
proc sql;
  title 'Population of Large Countries Grouped by Continent';
  select Continent, sum(Population) as TotPop format=comma15.
    from sql.countries
    where Population gt 1000000
    group by Continent
    order by TotPop;
quit;

/**************************************************************************
 Output 1.2 Sample DATA Step Output
**************************************************************************/
proc sql;
  title 'Large Countries Grouped by Continent';
proc summary data=sql.countries;
  where Population > 1000000;
  class Continent;
  var Population;
  output out=sumPop sum=TotPop;
run;

proc sort data=SumPop;
  by totPop;
run;

proc print data=SumPop noobs;
  var Continent TotPop;
  format TotPop comma15.;
  where _type_=1;
run;
```
Output 1.3 Countries (Partial Output)

options nodate nonumber linesize=84 pagesize=60;
proc sql outobs=15;
title 'Countries';
select Name format=$19., Capital format=$15.,
   Population, Area, Continent format=$15., UNDate format=year4.
   from sql.countries;

Output 1.4 WorldCityCoords (Partial Output)

proc sql outobs=15;
title 'WorldCityCoords';
select City format=$15., Country format=$12., Latitude, Longitude
   from sql.worldcitycoords;

Output 1.5 USCityCoords (Partial Output)

proc sql outobs=15;
title 'USCityCoords';
select City format=$15., State format=$2., Latitude, Longitude
   from sql.uscitycoords;

Output 1.6 UnitedStates (Partial Output)

proc sql outobs=15;
title 'UnitedStates';
select Name format=$17., Capital format=$15.,
   Population, Area, Continent format=$13., Statehood format=date9.
   from sql.unitedstates;

Output 1.7 PostalCodes (Partial Output)

proc sql outobs=15;
title 'PostalCodes';
select Name, Code
   from sql.postalcodes;

Output 1.8 WorldTemps (Partial Output)

proc sql outobs=15;
title 'WorldTemps';
select City, Country, avghigh, avglow
Retrieving Data from a Single Table

Some output tables restrict the number of observations. To list the entire table, remove the PROC SQL OUTOBS= option.

Output 2.1 Selecting All Columns in a Table

proc sql outobs=12;
   title 'U.S. Cities with Their States and Coordinates';
   select *
       from sql.uscitycoords;
*/

Output 2.2 Selecting One Column
--- Appendix 3 • Example Code Shown in Using the SQL Procedure ---

```sql
proc sql outobs=12;
  title 'Names of U.S. Cities';
  select City
    from sql.uscitycoords;
/*-------------------------- */
  Output 2.3 Selecting Multiple Columns

proc sql outobs=12;
  title 'U.S. Cities and Their States';
  select City, State
    from sql.uscitycoords;
/*-------------------------- */
  Output 2.4 Selecting a Column with Duplicate Values

proc sql outobs=12;
  title 'Continents of the United States';
  select Continent
    from sql.unitedstates;
/*-------------------------- */
  Output 2.5 Eliminating Duplicate Values

proc sql;
  title 'Continents of the United States';
  select distinct Continent
    from sql.unitedstates;
/*-------------------------- */
  Log 2.1 Portion of Log to Determine the Structure of a Table

proc sql;
  describe table sql.unitedstates;
/*-------------------------- */
  Output 2.6 Adding Text to Output

proc sql outobs=12;
  title 'U.S. Postal Codes';
  select 'Postal code for', Name, 'is', Code
    from sql.postalcodes;
/*-------------------------- */
  Output 2.7 Suppressing Column Headings in Output

proc sql outobs=12;
  title 'U.S. Postal Codes';
```
select 'Postal code for', Name label='#', 'is', Code label='#'
from sql.postalcodes;

/*---------------------------------------------*/
Output 2.8 Calculating Values

-------------------------------------------------------------------*/
proc sql outobs=12;
  title 'Low Temperatures in Celsius';
  select City, (AvgLow - 32) * 5/9 format=4.1
  from sql.worldtemps;

/*---------------------------------------------*/
Output 2.9 Assigning a Column Alias to a Calculated Column

-------------------------------------------------------------------*/
proc sql outobs=12;
  title 'Low Temperatures in Celsius';
  select City, (AvgLow - 32) * 5/9 as LowCelsius format=4.1
  from sql.worldtemps;

/*---------------------------------------------*/
Output 2.10 Referring to a Calculated Column by Alias

-------------------------------------------------------------------*/
proc sql outobs=12;
  title 'Range of High and Low Temperatures in Celsius';
  select City, (AvgHigh - 32) * 5/9 as HighC format=5.1,
       (AvgLow - 32) * 5/9 as LowC format=5.1,
       (calculated HighC - calculated LowC)
       as Range format=4.1
  from sql.worldtemps;

/*---------------------------------------------*/
Output 2.11 Using a Simple CASE Expression

-------------------------------------------------------------------*/
proc sql outobs=12;
  title 'Climate Zones of World Cities';
  select City, Country, Latitude,
       case
          when Latitude gt 67 then 'North Frigid'
          when 67 ge Latitude ge 23 then 'North Temperate'
          when 23 gt Latitude gt -23 then 'Torrid'
          when -23 ge Latitude ge -67 then 'South Temperate'
          else 'South Frigid'
          end as ClimateZone
  from sql.worldcitycoords
  order by City;

/*---------------------------------------------*/
Output 2.12 Using a CASE Expression in the CASE-OPERAND Form

-------------------------------------------------------------------*/
proc sql outobs=12;
  title 'Assigning Regions to Continents';
select Name, Continent,
case Continent
  when 'North America' then 'Continental U.S.'
  when 'Oceania' then 'Pacific Islands'
  else 'None'
end as Region
from sql.unitedstates;

/*-------------------------------------------------------------------
Output 2.13 Using the COALESCE Function to Replace Missing Values

Note: Either of the following two SQL statements will create Output 2.13.
-------------------------------------------------------------------*/
proc sql;
title 'Continental Low Points';
select Name, coalesce(LowPoint, 'Not Available') as LowPoint
from sql.continents;
proc sql;
title 'Continental Low Points';
select Name, case
  when LowPoint is missing then 'Not Available'
  else Lowpoint
end as LowPoint
from sql.continents;

/*-------------------------------------------------------------------
Output 2.14 Specifying Column Attributes
-------------------------------------------------------------------*/
proc sql outobs=12;
title 'Areas of U.S. States in Square Miles';
select Name label='State', Area format=comma10.
from sql.unitedstates;

/*-------------------------------------------------------------------
Output 2.15 Sorting by Column
-------------------------------------------------------------------*/
proc sql outobs=12;
title 'Country Populations';
select Name, Population format=comma10.
from sql.countries
order by Population;

/*-------------------------------------------------------------------
Output 2.16 Sorting by Multiple Columns
-------------------------------------------------------------------*/
proc sql outobs=12;
title 'Countries, Sorted by Continent and Name';
select Name, Continent
from sql.countries
order by Continent, Name;
Output 2.17 Specifying a Sort Order

```sql
proc sql outobs=12;
  title 'World Topographical Features';
  select Name, Type
  from sql.features
  order by Type desc, Name;
```

Output 2.18 Sorting by Calculated Column

```sql
proc sql outobs=12;
  title 'World Population Densities per Square Mile';
  select Name, Population format=comma12., Area format=comma8.,
      Population/Area as Density format=comma10.
  from sql.countries
  order by Density desc;
```

Output 2.19 Sorting by Column Position

```sql
proc sql outobs=12;
  title 'World Population Densities per Square Mile';
  select Name, Population format=comma12., Area format=comma8.,
      Population/Area format=comma10. label='Density'
  from sql.countries
  order by 4 desc;
```

Output 2.20 Sorting by Columns That Are Not Selected

```sql
proc sql outobs=12;
  title 'Countries, Sorted by Population';
  select Name, Continent
  from sql.countries
  order by Population;
```

Output 2.21 Sorting Columns That Contain Missing Values

```sql
proc sql;
  title 'Continents, Sorted by Low Point';
  select Name, LowPoint
  from sql.continents
  order by LowPoint;
```

Output 2.22 Using a Simple WHERE Clause

```sql
proc sql outobs=12;
```
title 'Countries in Europe';
select Name, Population format=comma10.
  from sql.countries
  where Continent = 'Europe';

PROC SQL;
title 'States with Populations over 5,000,000';
select Name, Population format=comma10.
  from sql.unitedstates
  where Population > 5000000
  order by Population desc;
PROC SQL;
title 'Countries in Africa with Populations over 20,000,000';
select Name, Population format=comma10.
  from sql.countries
  where Continent = 'Africa' and Population > 20000000
  order by Population desc;
PROC SQL outobs=12;
title 'World Mountains and Waterfalls';
select Name, Type, Height format=comma10.
  from sql.features
  where Type in ('Mountain', 'Waterfall')
  order by Height;
PROC SQL;
title 'Countries with Missing Continents';
select Name, Continent
  from sql.countries
  where Continent is missing;
PROC SQL outobs=12;
title 'Equatorial Cities of the World';
select City, Country, Latitude
  from sql.worldcitycoords
  where Latitude between -2.5 and 0.5;
where Latitude between -5 and 5;

/*-----------------------------------------------*/
\textbf{Output 2.28 Using the LIKE Operator}

proc sql;
    title1 'Country Names that Begin with the Letter "Z"';
    title2 'or Are 5 Characters Long and End with the Letter "a"';
    select Name
        from sql.countries
        where Name like 'Z%' or Name like '____a';

/*-----------------------------------------------*/
\textbf{Output 2.29 Using a Truncated String Comparison Operator}

proc sql;
    title '"New" U.S. States';
    select Name
        from sql.unitedstates
        where Name eqt 'New ';

/*-----------------------------------------------*/
\textbf{Output 2.30 Using a WHERE Clause with Missing Values (Incorrect Output)}

/* incorrect output */
proc sql outobs=12;
    title 'World Features with a Depth of Less than 500 Feet';
    select Name, Depth
        from sql.features
        where Depth lt 500
        order by Depth;

/*-----------------------------------------------*/
\textbf{Output 2.31 Using a WHERE Clause with Missing Values (Corrected Output)}

/* corrected output */
proc sql outobs=12;
    title 'World Features with a Depth of Less than 500 Feet';
    select Name, Depth
        from sql.features
        where Depth lt 500 and Depth is not missing
        order by Depth;

/*-----------------------------------------------*/
\textbf{Output 2.32 Using the MEAN Function with a WHERE Clause}

proc sql outobs=12;
    title 'Mean Temperatures for World Cities';
    select City, Country, mean(AvgHigh, AvgLow)
as MeanTemp
from sql.worldtemps
where calculated MeanTemp gt 75
order by MeanTemp desc;

Output 2.33 Displaying Sums

-------------------------------------------------------------------
proc sql;
title 'World Oil Reserves';
select sum(Barrels) format=comma18. as TotalBarrels
from sql.oilrsrvs;

Output 2.34 Using Aggregate Functions

-------------------------------------------------------------------
proc sql outobs=12;
title 'Largest Country Populations';
select Name, Population format=comma20.,
       max(Population) as MaxPopulation format=comma20.
from sql.countries
order by Population desc;

Output 2.35 Remerging Summary Statistics

-------------------------------------------------------------------
proc sql outobs=12;
title 'Percentage of World Population in Countries';
select Name, Population format=comma14.,
       (Population / sum(Population) * 100) as Percentage format=comma8.2
from sql.countries
order by Percentage desc;

Output 2.36 Using DISTINCT with the COUNT Function

-------------------------------------------------------------------
proc sql;
title 'Number of Continents in the COUNTRIES Table';
select count(distinct Continent) as Count
from sql.countries;

Output 2.37 Effect of Not Using DISTINCT with the COUNT Function

-------------------------------------------------------------------
proc sql;
title 'Countries for Which a Continent is Listed';
select count(Continent) as Count
from sql.countries;
Output 2.38 Using the COUNT FUNCTION to Count All Rows in a Table

---

```sql
proc sql;
  title 'Number of Countries in the SQL.COUNTRIES Table';
  select count(*) as Number
    from sql.countries;
```

---

Output 2.39 Finding Errors Caused by Missing Values (Unexpected Output)

---

```sql
/* unexpected output */
proc sql;
  title 'Average Length of Angel Falls, Amazon and Nile Rivers';
  select Name, Length, avg(Length) as AvgLength
    from sql.features
    where Name in ('Angel Falls', 'Amazon', 'Nile');
```

---

Output 2.40 Finding Errors Caused by Missing Values (Modified Output)

---

```sql
/* modified output */
proc sql;
  title 'Average Length of Angel Falls, Amazon and Nile Rivers';
  select Name, Length, coalesce(Length, 0) as NewLength,
    avg(calculated NewLength) as AvgLength
    from sql.features
    where Name in ('Angel Falls', 'Amazon', 'Nile');
```

---

Output 2.41 Grouping by One Column

---

```sql
proc sql;
  title 'Total Populations of World Continents';
  select Continent, sum(Population) format=comma14. as TotalPopulation
    from sql.countries
    where Continent is not missing
    group by Continent;
```

---

Output 2.42 Grouping without Aggregate Functions

Log 2.2 Grouping without Aggregate Functions (Partial Log)

---

```sql
proc sql outobs=12;
  title 'High and Low Temperatures';
  select City, Country, AvgHigh, AvgLow
    from sql.worldtemps
    group by Country;
```

---
Output 2.43 Grouping by Multiple Columns

```sql
proc sql;
  title 'Total Square Miles of Deserts and Lakes';
  select Location, Type, sum(Area) as TotalArea format=comma16.
    from sql.features
    where type in ('Desert', 'Lake')
    group by Location, Type;
```

Output 2.44 Grouping with an ORDER BY Clause

```sql
options nodate nonumber linesize=90 pagesize=60;
proc sql;
  title 'Total Square Miles of Deserts and Lakes';
  select Location, Type, sum(Area) as TotalArea format=comma16.
    from sql.features
    where type in ('Desert', 'Lake')
    group by Location, Type
    order by Location desc;
```

Output 2.45 Finding Grouping Errors Caused by Missing Values (Unexpected Output)

```sql
options nodate nonumber linesize=90 pagesize=60;
/* unexpected output */
proc sql outobs=12;
  title 'Areas of World Continents';
  select Name format=$25.,
      Continent,
      sum(Area) format=comma12. as TotalArea
    from sql.countries
    group by Continent
    order by Continent, Name;
```

Output 2.46 Adjusting the Query to Avoid Errors Due to Missing Values (Modified Output)

```sql
options nodate nonumber linesize=90 pagesize=60;
/* modified output */
proc sql outobs=12;
  title 'Areas of World Continents';
  select Name format=$25.,
      Continent,
      sum(Area) format=comma12. as TotalArea
    from sql.countries
    where Continent is not missing
    group by Continent
    order by Continent;
order by Continent, Name;

```sql
/*-------------------------------
Output 2.47 Using a Simple HAVING Clause
-------------------------------*/
proc sql;
  title 'Numbers of Islands, Oceans, and Seas';
  select Type, count(*) as Number
    from sql.features
    group by Type
    having Type in ('Island', 'Ocean', 'Sea')
    order by Type;
```

```sql
/*-------------------------------
Output 2.48 Using HAVING with the COUNT Function
-------------------------------*/
proc sql;
  title 'Total Populations of Continents with More than 15 Countries';
  select Continent,
    sum(Population) as TotalPopulation format=comma16.,
    count(*) as Count
    from sql.countries
    group by Continent
    having count(*) gt 15
    order by Continent;
```

```sql
/* Log 2.3 Validating a Query (Partial Log)
-------------------------------*/
proc sql;
  validate
    select Name, Statehood
    from sql.unitedstates
    where Statehood lt '01Jan1800'd;
```

```sql
/* Log 2.4 Validating an Invalid Query (Partial Log)
-------------------------------*/
proc sql;
  validate
    select Name, Statehood
    from sql.unitedstates
    where lt '01Jan1800'd;
```

**Retrieving Data from Multiple Tables**

Some output tables restrict the number of observations. To list the entire table, remove the PROC SQL OUTOBS= option.

```sql
/*-------------------------------
Output 3.1 Table One, Table Two
Output 3.2 Cartesian Product of Table One and Table Two
-------------------------------*/
proc sql;
```
data one;
  input X Y $;
  datalines;
  1 2
  2 3
 ;

data two;
  input X Z $;
  datalines;
  2 5
  3 6
  4 9
 ;
run;

proc sql;
  title 'Table One';
  select * from one;
  title 'Table Two';
  select * from two;
/

proc sql;
  title 'Table One and Table Two';
  select * from one, two
  where one.x=two.x;
/

proc sql outobs=6;
title 'Oil Production/Reserves of Countries';
select * from sql.oilprod as p, sql.oilrsrvs as r
where p.country = r.country;

/******************
Output 3.5 Ordering the Output of Joined Tables

Note: Both of the following PROC SQL steps produce Output 3.6.
---------------------------------------------*/
proc sql outobs=6;
  title 'Oil Production/Reserves of Countries';
  select p.country, barrelsperday 'Production', barrels 'Reserves'
  from sql.oilprod p, sql.oilrsrvs r
  where p.country = r.country
  order by barrelsperday desc;

proc sql outobs=6;
  title 'Oil Production/Reserves of Countries';
  select p.country, barrelsperday 'Production', barrels 'Reserves'
  from sql.oilprod p inner join sql.oilrsrvs r
    on p.country = r.country
  order by barrelsperday desc;

/******************
Output 3.6 Using Comparison Operators to Join Tables
Log 3.2 Comparison Query Log Message

---------------------------------------------*/
proc sql;
  title 'US Cities South of Cairo, Egypt';
  select us.City, us.State, us.Latitude, world.city, world.latitude
  from sql.worldcitycoords world, sql.uscitycoords us
  where world.city = 'Cairo' and
    us.latitude < world.latitude;

/******************
Output 3.7 Joining Tables That Contain Null Values

---------------------------------------------*/
data one;
  input a $ b;
datalines;
a 1
b 2
c
;

data two;
  input a $ b;
cards;
a 1
b 2
c
D 4
;
proc sql;
  title 'One and Two Joined';
  select one.a 'One', one.b, two.a 'Two', two.b
  from one, two
  where one.b=two.b;
run;

proc sql;
  title 'One and Two Joined';
  select one.a 'One', one.b, two.a 'Two', two.b
  from one, two
  where one.b=two.b and
  one.b is not missing;
run;

proc sql;
  title 'Coordinates of Capital Cities';
  select Capital format=$12., Name format=$12.,
    City format=$12., Country format=$12.,
    Latitude, Longitude
  from sql.countries, sql.worldcitycoords
  where Capital like 'L%' and
  Capital = City;
run;

proc sql;
  title 'Coordinates of Capital Cities';
  select Capital format=$12., Name format=$12.,
    City format=$12., Country format=$12.,
    latitude, longitude
  from sql.countries, sql.worldcitycoords
  where Capital like 'L%' and
  Capital = City and
  Name = Country;
run;

proc sql outobs=10;
  title 'Coordinates of State Capitals';
  select Capital format=$12., Name format=$12.,
    City format=$12., Country format=$12.,
    latitude, longitude
  from sql.countries, sql.worldcitycoords
  where Capital like 'L%' and
  Capital = City and
  Name = Country;
run;

Appendix 3 • Example Code Shown in Using the SQL Procedure
select us.Capital format=$15., us.Name 'State' format=$15.,
        pc.Code, c.Latitude, c.Longitude
from sql.unitedstates us, sql.postalcodes pc,
    sql.uscitycoords c
where us.Capital = c.City and
    us.Name = pc.Name and
    pc.Code = c.State;

/*-----------------------------------------------
Output 3.12 Joining a Table to Itself (Self-Join)
-----------------------------------------------*/
proc sql;
title "Cities' High Temps = Cities' Low Temps";
select High.City format $12., High.Country format $12.,
        High.AvgHigh, ' | ',
        Low.City format $12., Low.Country format $12.,
        Low.AvgLow
from sql.worldtemps High, sql.worldtemps Low
where High.AvgHigh = Low.AvgLow and
    High.city ne Low.city and
    High.country ne Low.country;

/*-----------------------------------------------
Output 3.13 Left Join of Countries and WordCityCoords
-----------------------------------------------*/
proc sql outobs=10;
title 'Coordinates of Capital Cities';
select Capital format=$20., Name 'Country' format=$20.,
        Latitude, Longitude
from sql.countries a left join sql.worldcitycoords b
    on a.Capital = b.City and
        a.Name = b.Country
order by Capital;

/*-----------------------------------------------
Output 3.14 Right Join of Countries and WorldCityCoords
-----------------------------------------------*/
proc sql outobs=10;
title 'Populations of Capitals Only';
select City format=$20., Country 'Country' format=$20.,
        Population
from sql.countries right join sql.worldcitycoords
    on Capital = City and
        Name = Country
order by City;

/*-----------------------------------------------
Output 3.15 Full Outer Join of Countries and WorldCityCoords
-----------------------------------------------*/
proc sql outobs=10;
title 'Populations and/or Coordinates of World Cities';
select City '#City#(WorldCityCoords)' format=$20.,
Capital 'Capital\(\text{Countries}\)' format=$20., Population, Latitude, Longitude
from sql.countries full join sql.worldcitycoords
on Capital = City and Name = Country;

Output 3.16 Tables One and Two

data one;
  input X Y $;
datalines;
  1 2
  2 3
;

data two;
  input W Z $;
datalines;
  2 5
  3 6
  4 9
;
run;
proc sql;
title 'Table One';
select * from one;
title 'Table Two';
select * from two;
title;
quit;

Output 3.17 Cross Join

Note: This example uses the same data sets as in Output 3.16.

proc sql;
title 'Table One and Table Two';
select *
  from one cross join two;

Output 3.18 Union Join

Note: This example uses the same data sets as in Output 3.16.

proc sql;
select *
  from one union join two;
libname sql 'SAS-library';

proc sql outobs=6;
  title 'Oil Production/Reserves of Countries';
  select country, barrelsperday 'Production', barrels 'Reserve'
    from sql.oilprod natural join sql.oilrsrvs
    order by barrelsperday desc;

proc sql outobs=10;
  title 'Populations and/or Coordinates of World Cities';
  select coalesce(Capital, City,Name) format=$20. 'City',
      coalesce(Name, Country) format=$20. 'Country',
      Population, Latitude, Longitude
    from sql.countries full join sql.worldcitycoords
    on Capital = City and
    Name = Country;

data fltsuper;
  input Flight Supervisor $;
datalines;
  145 Kang
  150 Miller
  155 Evanko
;

data fltdest;
  input Flight Destination $;
datalines;
  145 Brussels
  150 Paris
  155 Honolulu
;
run;

data merged;
  merge FltSuper FltDest;
  by Flight;
run;

proc print data=merged noobs;
  title 'Table MERGED';

run;

proc sql;
   title 'Table Merged';
   select s.flight, Supervisor, Destination
       from fltsuper s, fltdest d
       where s.Flight=d.Flight;

/*-------------------------------------------------------------------
Output 3.22 Merged Tables When Some of the Values Match

Note: The two DATA steps produce the datasets used by both
the merge and the join. The merge and the join produce
identical output.
-------------------------------------------------------------------*/
data fltsuper;
   input Flight Supervisor $;
datalines;
  145 Kang
  150 Miller
  155 Evanko
  157 Lei
;
data fltdest;
   input Flight Destination $;
datalines;
  145 Brussels
  150 Paris
  165 Seattle
;
run;

data merged;
   merge fltsuper fltdest;
   by flight;
run;
proc print data=merged noobs;
   title 'Table Merged';
run;

proc sql;
   select coalesce(s.Flight,d.Flight) as Flight, Supervisor, Destination
       from fltsuper s full join fltdest d
       on s.Flight=d.Flight;

/*-------------------------------------------------------------------
Output 3.23 Match-Merge of the FltSuper and FltDest Tables
-------------------------------------------------------------------*/
data fltsuper;
   input Flight Supervisor $;
datalines;
  145 Kang
  145 Rameriz
  150 Miller
  150 Picard
;
data fltdest;
input Flight Destination $;
datalines;
145   Brussels
145   Edmonton
150   Paris
150   Madrid
165   Seattle
;
run;

data merged;
merge fltsuper fltdest;
by flight;
run;

proc print data=merged noobs;
title 'Table Merged';
run;

/*---------------------------------------------------
Output 3.24 PROC SQL Join of the FltSuper and FltDest Tables
---------------------------------------------------*/
data fltsuper;
input Flight Supervisor $;
datalines;
145   Kang
145   Rameriz
150   Miller
150   Picard
155   Evanko
157   Lei
;

data fltdest;
input Flight Destination $;
datalines;
145   Brussels
145   Edmonton
150   Paris
150   Madrid
165   Seattle
;
run;

proc sql;
title 'Table Joined';
select *
from fltsuper s, fltdest d
where s.Flight=d.Flight;
Output 3.25 Single-Value Subquery

```sql
proc sql;
  title 'U.S. States with Population Greater than Belgium';
  select Name 'State' , population format=comma10.
  from sql.unitedstates
  where population gt
    (select population from sql.countries
     where name = "Belgium");
```

Output 3.26 Multiple-Value Subquery Using IN

```sql
proc sql outobs=5;
  title 'Populations of Major Oil Producing Countries';
  select name 'Country', Population format=comma15.
  from sql.countries
  where Name in
    (select Country from sql.oilprod);
```

Output 3.27 Multiple-Value Subquery Using NOT IN

```sql
proc sql outobs=5;
  title 'Populations of NonMajor Oil Producing Countries';
  select name 'Country', Population format=comma15.
  from sql.countries
  where Name not in
    (select Country from sql.oilprod);
```

Output 3.28 Correlated Subquery

```sql
proc sql;
  title 'Oil Reserves of Countries in Africa';
  select * from sql.oilrsrvs o
  where 'Africa' =
    (select Continent from sql.countries c
     where c.Name = o.Country);
```

Output 3.29 Testing for the Existence of a Group of Values

```sql
proc sql;
  title 'Oil Reserves of Countries in Africa';
  select * from sql.oilrsrvs o
  where exists
    (select Continent from sql.countries c
     where o.Country = c.Name and
     Continent = 'Africa');
```
/*---------------------------------------------*/
Output 3.30 Multiple Levels of Subquery Nesting

proc sql;
  title 'Coordinates of African Cities with Major Oil Reserves';
  select * from sql.worldcitycoords
    where country in
      (select Country from sql.oilrsrvs o
        where o.Country in
          (select Name from sql.countries c
            where c.Continent = 'Africa'));

/*---------------------------------------------*/
Output 3.31 Combining a Join with a Subquery

proc sql outobs=10;
  title 'Neighboring Cities';
  select a.City format=$10., a.State,
    a.Latitude 'Lat', a.Longitude 'Long',
    b.City format=$10., b.State,
    b.Latitude 'Lat', b.Longitude 'Long',
    sqrt(((b.latitude-a.latitude)**2) +
      ((b.longitude-a.longitude)**2)) as dist format=6.1
  from sql.uscitycoords a, sql.uscitycoords b
  where a.city ne b.city and
    calculated dist =
      (select min(sqrt(((d.latitude-c.latitude)**2) +
        ((d.longitude-c.longitude)**2)))
        from sql.uscitycoords c, sql.uscitycoords d
        where c.city = a.city and
          c.state = a.state and
          d.city ne c.city)
  order by a.city;

/*---------------------------------------------*/
Output 3.32 Tables Used in Set Operation Examples

proc sql;
  create table sql.A
    (x int, y varchar(12));
  create table sql.B
    (x int, z varchar(12));
  insert into sql.A
    values (1,'one')
    values (2,'two')
    values (2,'two')
    values (3,'three')
;
proc sql;
  insert into sql.B
    values (1,'one')
    values (2,'two')
    values (4,'four');
quit;

proc sql;
  title 'Table A';
  select * from sql.a;
  title 'Table B';
  select * from sql.b;

/*-------------------------------------------------------------------
 Output 3.33 Producing Unique Rows from Both Queries (UNION)
 Note: This example uses the data sets that were created in Output 3.32.
-------------------------------------------------------------------*/
proc sql;
  title 'A UNION B';
  select * from sql.a
  union
  select * from sql.b;

/*-------------------------------------------------------------------
 Output 3.34 Producing Rows from Both Queries (UNION ALL)
 Note: This example uses the data sets that were created in Output 3.32.
-------------------------------------------------------------------*/
proc sql;
  title 'A UNION ALL B';
  select * from sql.a
  union all
  select * from sql.b;

/*-------------------------------------------------------------------
 Output 3.35 Producing Rows That Are in Only the First Query Result (EXCEPT)
 Note: This example uses the data sets that were created in Output 3.32.
-------------------------------------------------------------------*/
proc sql;
  title 'A EXCEPT B';
  select * from sql.a
  except
  select * from sql.b;

/*-------------------------------------------------------------------
 Output 3.36 Producing Rows That Are in Only the First Query Result
 (EXCEPT ALL)
 Note: This example uses the data sets that were created in Output 3.32.
-------------------------------------------------------------------*/
proc sql;
title 'A EXCEPT ALL B';
select * from sql.a
except all
select * from sql.b;

/*
Output 3.37 Producing Rows That Belong to Both Query Results (INTERSECT)

Note: This example uses the data sets that were created in Ouptut 3.32.*/
proc sql;
title 'A INTERSECT B';
select * from sql.a
intersect
select * from sql.b;

/*
Output 3.38 Concatenating the Query Results (OUTER UNION)

Note: This example uses the data sets that were created in Ouptut 3.32.*/
proc sql;
title 'A OUTER UNION B';
select * from sql.a
outer union
select * from sql.b;

/*
Output 3.39 Concatenating the Query Results (OUTER UNION CORR)

Note: This example uses the data sets that were created in Ouptut 3.32.*/
proc sql;
title 'A OUTER UNION CORR B';
select * from sql.a
outer union corr
select * from sql.b;

/*
Output 3.40 Producing Rows from the First Query or the Second Query

Note: This example uses the data sets that were created in Ouptut 3.32.*/
proc sql;
title 'A EXCLUSIVE UNION B';
(select * from sql.a
except
select * from sql.b)
union
(select * from sql.b
except
select * from sql.a);
Creating and Updating Tables and Views

Some output tables restrict the number of observations. To list the entire table, remove the PROC SQL OUTOBS= option.

```sql
/*----------------------------------------------------------*/
/* Log 4.1 Table Created from Column Definitions */
/*----------------------------------------------------------*/
proc sql;
  create table sql.newstates
    (state char(2), /* 2–character column for state abbreviation */
     date num /* column for date of entry into the US */
      informat=date9. /* with an informat */
      format=date9., /* and format of DATE9. */
     population num); /* column for population */
  describe table sql.newstates;
/*----------------------------------------------------------*/
/* Output 4.1 Table Created from a Query Result */
/*----------------------------------------------------------*/
proc sql outobs=10;
title 'Densities of Countries';
create table sql.densities as
  select Name 'Country' format $15.,
         Population format=comma10.0,
         Area as SquareMiles,
         Population/Area format=6.2 as Density
  from sql.countries;
select * from sql.densities;
/*----------------------------------------------------------*/
/* Log 4.2 SAS Log for DESCRIBE TABLE Statement for Densities */
/*----------------------------------------------------------*/
proc sql;
describe table sql.densities;
/*----------------------------------------------------------*/
/* Log 4.3 SAS Log for DESCRIBE TABLE Statement for NewCountries */
/*----------------------------------------------------------*/
/* Create the newcountries table. */
proc sql;
create table sql.newcountries
  like sql.countries;
```
describe table sql.newcountries;

/*----------------------------------------------------------*/
/* Insert 4.2 Rows Inserted with the SET Clause */
/*----------------------------------------------------------*/
proc sql;
create table sql.newcountries
like sql.countries;
/* Insert all of the rows from countries into newcountries based */
/* on a population of 130000000.                             */
proc sql;
insert into sql.newcountries
select * from sql.countries
where population ge 130000000;
/* Insert 2 new rows in the newcountries table. */
/* Print the table. */
proc sql;
insert into sql.newcountries
title "World's Largest Countries"
set name='Bangladesh',
capital='Dhaka',
population=126391060
set name='Japan',
capital='Tokyo',
population=126352003;
title "World's Largest Countries";
select name format=$20.,
capital format=$15.,
population format=comma15.0
from sql.newcountries;

/*----------------------------------------------------------*/
/* Insert 4.3 Rows Inserted with the VALUES Clause */
/*----------------------------------------------------------*/
proc sql;
insert into sql.newcountries
values ('Pakistan', 'Islamabad', 123060000, ., ' ', .)
values ('Nigeria', 'Lagos', 99062000, ., ' ', .);
title "World's Largest Countries";
select name format=$20.,
capital format=$15.,
population format=comma15.0
from sql.newcountries;

/*----------------------------------------------------------*/
/* Insert 4.4 Rows Inserted with a Query */
/*----------------------------------------------------------*/
proc sql;
insert into sql.newcountries
values ('Pakistan', 'Islamabad', 123060000, ., ' ', .)
values ('Nigeria', 'Lagos', 99062000, ., ' ', .);
title "World's Largest Countries";
select name format=$20.,
capital format=$15.,
population format=comma15.0
from sql.newcountries;
---*---
proc sql;
create table sql.newcountries
  like sql.countries;
proc sql;
  title "World's Largest Countries";
  insert into sql.newcountries
  select * from sql.countries
  where population ge 130000000;
  select name format=$20.,
    capital format=$15.,
    population format=comma15.0
  from sql.newcountries;
/*-----------------------------
Output 4.5 A Lesser Number of Columns in Rows Inserted with a Query
-----------------------------*/
proc sql;
create table sql.newcountries
  like sql.countries;
proc sql;
  title "World's Largest Countries";
  insert into sql.newcountries (Name,Population)
  select Name,Population from sql.countries
  where population ge 130000000;
  select name format=$20.,
    population format=comma15.0
  from sql.newcountries;
/*-----------------------------
Output 4.6 Updating a Column for All Rows
-----------------------------*/
proc sql;
create table sql.newcountries like sql.countries;
insert into sql.newcountries
select * from sql.countries
where population ge 130000000;
quit;
proc sql;
  update sql.newcountries
  set population=population*1.05;
  title "Updated Population Values";
  select name format=$20.,
    capital format=$15.,
    population format=comma15.0
  from sql.newcountries;
/*-----------------------------
Output 4.7 Selectively Updating a Column
-----------------------------*/
PROC SQL;
    CREATE TABLE sql.newcountries LIKE sql.countries;
    INSERT INTO sql.newcountries
        SELECT * FROM sql.countries
        WHERE population GE 130000000;
    QUIT;
PROC SQL;
    UPDATE sql.newcountries
        SET population = population * 1.05
        WHERE name LIKE 'B%';
    UPDATE sql.newcountries
        SET population = population * 1.07
        WHERE name IN ('China', 'Russia');
     TITLE "SELECTIVELY UPDATED POPULATION VALUES";
    SELECT name FORMAT=$20.,
            capital FORMAT=$15.,
            population FORMAT=COMMA15.0
    FROM sql.newcountries;

You can accomplish the same result with a CASE expression:
    UPDATE sql.newcountries
        SET population = population * CASE WHEN name LIKE 'B%' THEN 1.05
                                WHEN name IN ('China', 'Russia') THEN 1.07
                                ELSE 1
                               END;

*/------------------------------------------------------------------------------------*/
Log 4.4 SAS Log for the DELETE Statement

-------------------------------------------------------------------*/0
/*/ Create and populate NewCountries */
PROC SQL;
    CREATE TABLE sql.newcountries LIKE sql.countries;
    INSERT INTO sql.newcountries
        SELECT * FROM sql.countries
        WHERE population GE 130000000;
    QUIT;
PROC SQL;
    DELETE
        FROM sql.newcountries
        WHERE name LIKE 'R%';

Output 4.8 Adding a New Column

-------------------------------------------------------------------*/
/*/ Create and populate NewCountries */
PROC SQL;
create table sql.newcountries like sql.countries;
insert into sql.newcountries
select * from sql.countries
where population ge 130000000;
quit;

proc sql;
   alter table sql.newcountries
       add density num label='Population Density' format=6.2;
   title "Population Density Table";
   select name format=$20.,
       capital format=$15.,
       population format=comma15.0,
       density
   from sql.newcountries;
proc sql;
   create table sql.newcountries as
       select *, population/area as density
       label='Population Density'
       format=6.2
   from sql.newcountries;
proc sql;
   alter table sql.newcountries
       add density num label='Population Density' format=6.2;
   title "Population Density Table";
   select name format=$20.,
       capital format=$15.,
       population format=comma15.0,
       density
   from sql.newcountries;
proc sql;
   update sql.newcountries
       set density=population/area;
   title "Population Density Table";
   select name format=$20.,
       capital format=$15.,
       population format=comma15.0,
       density
   from sql.newcountries;
proc sql;
   create table sql.newcountries as
       select *, population/area as density
       label='Population Density'
       format=6.2
   from sql.newcountries;

/*-----------------------------------------------*
Output 4.9 Fillin in the New Column's Values

Note: After the NewCountries table is created, the two SQL statements
produce the same output.
-----------------------------------------------*/

/* Create and populate Newcountries */
proc sql;
   create table sql.newcountries like sql.countries;
   insert into sql.newcountries
      select * from sql.countries
      where population ge 130000000;
   alter table sql.newcountries
      add density num label='Population Density' format=6.2;
   quit;
   proc sql;
      update sql.newcountries
         set density=population/area;
      title "Population Density Table";
      select name format=$20.,
         capital format=$15.,
         population format=comma15.0,
         density
      from sql.newcountries;
   proc sql;
      create table sql.newcountries as
         select *, population/area as density
         label='Population Density'
         format=6.2
      from sql.newcountries;

/*-----------------------------------------------*
Output 4.10 Modifying a Column Format
-----------------------------------------------*/
/* Create and populate NewCountries */
proc sql;
    create table sql.newcountries like sql.countries;
    insert into sql.newcountries
        select * from sql.countries
        where population ge 130000000;
quit;

proc sql;
    title "World's Largest Countries";
    alter table sql.newcountries
        modify population format=comma15.;
    select name, population from sql.newcountries;
/*-------------------------------------------------------------------*/

proc sql;
    title "World's Largest Countries";
    alter table sql.newcountries
        modify name char(60) format=$60.;
    update sql.newcountries
        set name='The United Nations member country is '||name;
    select name from sql.newcountries;
/*-------------------------------------------------------------------*/

proc means data=sql.countries mean maxdec=2;
    title "Mean Area for All Countries";
    var area;
run;
/*-------------------------------------------------------------------*/

proc sql;
    create table sql.mystates
        (state      char(15),
         population num,
         continent  char(15),
        /* constraint specifications */
        constraint prim_key    primary key(state),
        constraint population  check(population gt 0),
        constraint continent   check(continent in ('North America', 'Oceania')));

create table sql.uspostal
    (name      char(15),
     code      char(2) not null,  /* constraint specified as */
     /* a column attribute */
     /* constraint specified as */

constraint for_key foreign key(name) /* links NAME to the */
    references sql.mystates /* primary key in MYSTATES */
    on delete restrict /* forbids deletions to STATE */
    /* unless there is no */
    /* matching NAME value */
    on update set null); /* allows updates to STATE, */
    /* changes matching NAME */
    /* values to missing */

proc sql;
    describe table sql.mystates;
    describe table constraints sql.uspostal;

/*----------------------------------------------------------*/
Output 4.14 An SQL Procedure View

-------------------------------------------------------------------*/
proc sql;
    title 'Current Population Information for Continents';
    create view sql.newcontinents as
        select continent,
            sum(population) as totpop format=comma15. label='Total Population',
            sum(area) as totarea format=comma15. label='Total Area'
        from sql.countries
        group by continent;

    select * from sql.newcontinents;

/*----------------------------------------------------------*/
Log 4.5 SAS Log from DESCRIBE VIEW Statement

-------------------------------------------------------------------*/
proc sql;
    describe view sql.newcontinents;

/*----------------------------------------------------------*/
Output 4.15 Using an In-Line View

-------------------------------------------------------------------*/
proc sql;
    title 'Countries With Population GT Caribbean Countries';
    select w.Name, w.Population format=comma15., c.TotCarib
    from (select sum(population) as TotCarib format=comma15.
            from sql.countries
            where continent = 'Central America and Caribbean') as c,
        sql.countries as w
    where w.population gt c.TotCarib;

Programming with the SQL Procedure

Some output tables restrict the number of observations. To list the entire table, remove the PROC SQL OUTOBS= option.
Log 5.1 Expanded SELECT * Statement

proc sql feedback;
   select * from sql.countries;

Logt 5.2 Comparing Run Times of Two Queries

proc sql stimer ;
   select us.name, us.population
       from sql.unitedstates as us, sql.countries as w
       where us.population gt w.population and
             w.name = 'Belgium';

   select Name, population
       from sql.unitedstates
       where population gt
             (select population from sql.countries
             where name = 'Belgium');

Output 5.1 Resetting PROC SQL Options with the RESET Statement

proc sql noprint;
   title 'Countries with Population Under 20,000';
   select Name, Population from sql.countries;
   reset print number;
   select Name, Population from sql.countries
       where population lt 20000;

Log 5.3 Definition of DICTIONARY.Tables

proc sql;
   describe table dictionary.tables;

Log 5.4 Description of Sashelp.Vstabvw

proc sql;
   describe view sashelp.vstabvw;

Output 5.2 Tables and Views Used in This Document

proc sql;
   title 'All Tables and Views in the SQL Library';
   select libname, memname, memtype, nobs
       from dictionary.tables
       where libname='SQL';
proc sql;
    title 'All Tables that Contain the Country Column';
    select libname, memname, name
        from dictionary.columns
        where name='Country' and
            libname='SQL';

proc sql noprint;
    select country, barrels
        into :country1, :barrels1
        from sql.oilrsrvs;
%put &country1 &barrels1;

proc sql outobs=12;
    reset noprint;
    select max(AvgHigh)
        into :maxtemp
        from sql.worldtemps
        where country = 'Canada';
    reset print;
    title "The Highest Temperature in Canada: &maxtemp";
    select city, AvgHigh format 4.1
        from sql.worldtemps
        where country = 'Canada';

proc sql noprint;
    select name, Population
        into :country1 - :country4, :pop1 - :pop3
        from sql.countries;
%put &country1 &pop1;
%put &country2 &pop2;
%put &country3 &pop3;
%put &country4;

proc sql noprint;
    select libname, memname, name
        from dictionary.columns
        where name='Country' and
            libname='SQL';
Example Code: Using the SQL Procedure

```sql
proc sql noprint inobs=5;
  select Name
    into :countries separated by ', '
    from sql.countries;
%put &countries;

/*---------------------------------------------*/
Log 5.8 Concatenating Values in Macro Variables

proc sql noprint inobs=5;
  select Name
    into :countries separated by ',' NOTRIM
    from sql.countries;
%put &countries;

/*---------------------------------------------*/
Log 5.9 Defining Macros to Create Tables
Output 5.5 Result Table and Message Created with SAS Macro Language Interface

   Note: The %ADDREF() macros at the end of this code should not be included in the PROC SQL step. See the text.

proc sql;
create table sql.referee
  (Name      char(15),
   Subject   char(15));

/* define the macro */
%macro addref(name,subject);
%local count;
/* are there three referees in the table? */
reset noprint;
  select count(*)
    into :count
    from sql.referee
    where subject="&subject";
%if &count ge 3 %then %do;
    reset print;
    title "ERROR: &name not inserted for subject – &subject..";
    title2 " There are 3 referees already."
    select * from sql.referee where subject="&subject";
    reset noprint;
%end;
%else %do;
  insert into sql.referee(name,subject) values("&name","&subject");
  %put NOTE: &name has been added for subject – &subject..;
%end;
```
%mend;
%addref(fay,sailing);
%addref(einstein,relativity);
%addref(smythe,sailing);
%addref(naish,sailing);

/*----------------------------------------------*/
Log 5.10 Using the PROC SQL Automatic Macro Variables
proc sql noprint;
select * from sql.countries;
%put SQLOBS=*&sqlobs* SQLOOPS=*&sqloops* SQLRC=*&sqlrc*;

/*----------------------------------------------*/
Output 5.6 USCityCoords Table Showing Repeating State Values
proc sql outobs=10;
title 'US Cities';
select State, City, Latitude, Longitude
from sql.uscitycoords
order by state;
run;

/*----------------------------------------------*/
Output 5.7 PROC REPORT Output Showing the First Occurrence Only of Each State Value
proc sql noprint;
create table sql.cityreport as
select *
from sql.uscitycoords
order by state;
proc report data=sql.cityreport
headline
headskip;
title 'Coordinates of U.S. Cities in Pacific Rim States';
column state city ('Coordinates' latitude longitude);
define state / order format=$2. width=5 'State';
define city / order format=$15. width=15 'City';
define latitude / display format=4. width=8 'Latitude';
define longitude / display format=4. width=9 'Longitude';
where state='AK' or state='HI' or state='WA' or state='OR' or state='CA';
run;

/*----------------------------------------------*/
Output 5.8 Output from Querying a DBMS Table
Note: This example assumes a Payroll table exists in an
libname mydblib oracle user=user-id
    password=password
    path=path-name schema=schema-name;

proc sql;
    select jobcode label='Jobcode',
        sum(salary) as total
    label='Total for Group'
    format=dollar11.2
    from mydblib.payroll
    group by jobcode;
quit;

libname mydblib oracle user=user-id password=password
proc sql;
    create view LON as
        select flight, dates, idnum
        from mydblib.schedule
        where dest='LON';
quit;
proc print data=work.LON noobs;
run;

proc sql outobs=15;
    connect to oracle as ora2 (user=user-id password=password);
    select * from connection to ora2 (select lname, fname, state from staff);
    disconnect from ora2;
quit;

ods html body='odsout.htm';
proc sql outobs=12;
    title 'U.S. Cities with Their States and Coordinates';
    select *
    from sql.uscitycoords;
ods html close;
Practical Problem-Solving with PROC SQL

Some output tables restrict the number of observations. To list the entire table, remove the PROC SQL OUTOBS= option.

```/*------------------------------*/
Output 6.1 Sample Input Table for Weighted Average
```
```data Sample;
do i=1 to 10;
  Value=2983*ranuni(135);
  Weight=120 + 33*rannor(135);
  if mod(i,2)=0 then Gender='M';
  else Gender='F';
  output;
end;
drop i;
```
```proc print data=Sample;
title 'Sample Data for Weighted Average';
run;
```
```/*------------------------------*/
Output 6.2 PROC SQL Output for Weighted Averages
```
```proc sql;
title 'Weighted Averages from Sample Data';
select Gender, sum(Value*Weight)/sum(Weight) as WeightedAverage
from (select Gender, Value,
case
  when Weight gt 0 then Weight
  else 0
  end as Weight
from Sample)
group by Gender;
```
```/*------------------------------*/
Output 6.3 Sample Input Tables for Table Comparison
```
```data oldstaff;
  input id $ Last : $10. First $ Middle $ Phone $ Location $;
datalines;
5463 Olsen Mary K.  661-0012 R2342
6574 Hogan Terence H.  661-3243 R4456
7896 Bridges Georgina W.  661-8897 S2988
4352 Anson Sanford .  661-4432 S3412
5674 Leach Archie G.  661-4328 S3533
7902 Wilson Fran R.  661-8332 R4454
0001 Singleton Adam O.  661-0980 R4547
9786 Thompson Jack .  661-6781 R2343
```
data newstaff;
  input id $ Last $10. First $ Middle $ Phone $ Location $;
datalines;
5463 Olsen Mary K. 661-0012 R2342
6574 Hogan Terence H. 661-3243 R4456
7896 Bridges Georgina W. 661-2231 S2987
4352 Anson Sanford . 661-4432 S3412
5674 Leach Archie G. 661-4328 S3533
7902 Wilson Fran R. 661-8332 R4454
0001 Singleton Adam O. 661-0980 R4457
9786 Thompson John C. 661-6781 R2343
2123 Chen Bill W. 661-8099 R4432
;
/*-------------------------------------------------------------------
  Output 6.4 Rows That Have Changed
  Note: This example uses the data sets that were created in Output 6.3.
-------------------------------------------------------------------*/
proc sql;
title 'Updated Rows';
select * from newstaff
except
select * from oldstaff;
 /*-------------------------------------------------------------------
  Output 6.5 Sample Input Tables for Overlaying Missing Values
-------------------------------------------------------------------*/
data league1;
input @1 Fullname $20. @21 Bowler $4. @29 AvgScore 3.;
cards;
Alexander Delarge   4224    164
John T Chance       4425
Jack T Colton       4264
       1412    141
Andrew Shepherd     4189    185
;
data league2;
input @1 FirstName $10. @12 LastName $15. @28 AMFNo $4. @38 AvgScore 3.;
cards;
Alex       Delarge         4224      156
Mickey     Raymond         1412
        4264      174
Jack       Chance          4425
Patrick    O'Malley        4118      164
;
proc sql;
title 'Bowling Averages from League1';
select * from league1;
title 'Bowling Averages from League2';
select * from league2;
Output 6.6 PROC SQL Output for Overlaying Missing Values

Note: This example uses the data sets that were created in Output 6.5.

```sql
proc sql;
  title "Averages from Last Year's League When Possible";
  title2 "Supplemented when Available from Prior Year's League";
  select coalesce(lastyr.fullname,trim(prioryr.firstname) || ' ' || prioryr.lastname) as Name format=$26.,
      coalesce(lastyr.bowler,prioryr.amfno) as Bowler,
      coalesce(lastyr.avgscore,prioryr.avgscore) as Average format=8.
  from league1 as lastyr full join league2 as prioryr
  on lastyr.bowler=prioryr.amfno
  order by Bowler;
```

Output 6.7 Input Table for Computing Subtotal Percentages
(Partial Output)

```sql
data survey;
  input State $ Answer $ @@;
  datalines;
  NY YES NY YES NY YES NY YES NY NO NY NO NY NO NC YES
  NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC YES NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO NC NO
  PA YES PA YES PA YES PA YES PA YES PA YES PA YES PA YES PA YES
  PA YES PA YES PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO
  PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO PA NO
  VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES
  VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES
  VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES VA YES
  VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO
  VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO VA NO;
proc print data=Survey(obs=10);
  title 'Sample Data for Subtotal Percentages';
run;
```

Output 6.8 PROC SQL Output That Computes Percentages within Subtotals

Note: This example uses the data set Survey that was created in Output 6.7.

```sql
proc sql;
  title1 'Survey Responses';
  select survey.Answer, State, count(State) as Count,
      calculated Count/Subtotal as Percent format=percent8.2
  from survey,
      (select Answer, count(*) as Subtotal from survey
       group by Answer) as survey2
  where survey.Answer=survey2.Answer
      group by survey.Answer, State;
quit;
```
data Duplicates;
  input LastName $ FirstName $ City $ State $;
datalines;
Smith John Richmond Virginia
Johnson Mary Miami Florida
Smith John Richmond Virginia
Reed Sam Portland Oregon
Davis Karen Chicago Illinois
Davis Karen Chicago Illinois
Thompson Jennifer Houston Texas
Smith John Richmond Virginia
Johnson Mary Miami Florida
;

proc print data=Duplicates;
  title 'Sample Data for Counting Duplicates';
run;

proc sql;
  title 'Duplicate Rows in Duplicates Table';
  select *, count(*) as Count
  from Duplicates
  group by LastName, FirstName, City, State
  having count(*) > 1;

data Employees;
  input ID $ LastName $ FirstName $ Supervisor $;
datalines;
1001 Smith John 1002
1002 Johnson Mary None
1003 Reed Sam None
1004 Davis Karen 1003
1005 Thompson Jennifer 1002
1006 Peterson George 1002
1007 Jones Sue 1003
1008 Murphy Janice 1003
1009 Garcia Joe 1002
;

proc print data=Employees;
  title 'Sample Data for Expanding a Hierarchy';
run;
/*------------------------------------------*/
Output 6.12 PROC SQL Output for Expanding a Hierarchy

Note: This example uses the data set Employees that was created in Output 6.11.

---------------------------------------------------------------------*/
proc sql;
title 'Expanded Employee and Supervisor Data';
select A.ID label="Employee ID",
      trim(A.FirstName)||' ||A.LastName label="Employee Name",
      B.ID label="Supervisor ID",
      trim(B.FirstName)||' ||B.LastName label="Supervisor Name"
from Employees A, Employees B
      where A.Supervisor=B.ID and A.Supervisor is not missing;

/*---------------------------------------------------------------------*/
Output 6.13 Sample Input Table for Summarizing Data from Multiple Columns

---------------------------------------------------------------------*/
data Sales;
   input Salesperson $ January February March;
datalines;
Smith 1000 650 800
Johnson 0 900 900
Reed 1200 700 850
Davis 1050 900 1000
Thompson 750 850 1000
Peterson 900 600 500
Jones 800 900 1200
Murphy 700 800 700
Garcia 400 1200 1150
;
proc print data=Sales;
title 'Sample Data for Summarizing Data from Multiple Columns';
run;
/*------------------------------------------*/
Output 6.14 PROC SQL Output for Summarizing Data from Multiple Columns

Note: This example uses the data set Sales that was created in Output 6.13.

---------------------------------------------------------------------*/
proc sql;
title 'Total First Quarter Sales';
select sum(January)  as JanTotal,
      sum(February) as FebTotal,
      sum(March)    as MarTotal,
      sum(calculated JanTotal, calculated FebTotal,
           calculated MarTotal) as GrandTotal format=dollar10.
from Sales;
/*------------------------------------------*/
Output 6.15 Sample Input Table for Creating a Summary Report
data sales;
  input Site $ Product $ Invoice $ InvoiceAmount InvoiceDate $;
datalines;
V1009 VID010 V7679 598.5 980126
V1019 VID010 V7688 598.5 980126
V1032 VID005 V7771 1070 980309
V1043 VID014 V7780 1070 980309
V421 VID003 V7831 2000 980330
V421 VID010 V7832 750 980330
V570 VID003 V7762 2000 980302
V659 VID003 V7730 1000 980223
V783 VID003 V7815 750 980323
V985 VID003 V7733 2500 980223
V966 VID001 V5020 1167 980215
V98 VID003 V7750 2000 980223
;
proc sql;
  title 'Sample Data to Create Summary Sales Report';
  select * from sales;
quit;

/*------------------------------------------
Output 6.16 PROC SQL Output for a Summary Report
Note: This example uses the data set Sales that was created in Output 6.15.
------------------------------------------*/
proc sql;
  title 'First Quarter Sales by Product';
  select Product,
      sum(Jan) label='Jan',
      sum(Feb) label='Feb',
      sum(Mar) label='Mar'
  from (select Product,
      case
        when substr(InvoiceDate,3,2)='01' then
          InvoiceAmount end as Jan,
      case
        when substr(InvoiceDate,3,2)='02' then
          InvoiceAmount end as Feb,
      case
        when substr(InvoiceDate,3,2)='03' then
          InvoiceAmount end as Mar
  from work.sales)
  group by Product;

/*------------------------------------------
Output 6.17 Sample Input Data for a Customized Sort
------------------------------------------*/
data chores;
  input Project $ Hours Season $;
datalines;
weeding 48 summer
pruning 12 winter
mowing 36 summer
mulching 17 fall
raking 24 fall
raking 16 spring
planting 8 spring
planting 8 fall
sweeping 3 winter
edging 16 summer
seeding 6 spring
tilling 12 spring
aerating 6 spring
feeding 7 summer
rolling 4 winter

proc sql;
title 'Garden Chores';
select * from chores;
quit;

/*-------------------------------------------------------------------
  Output 6.18 PROC SQL Output for a Customized Sort Sequence
  Note: This example uses the data set Chores that was created in Output 6.17.
-------------------------------------------------------------------*/
proc sql;
title 'Garden Chores by Season in Logical Order';
select Project, Hours, Season
from (select Project, Hours, Season,
case
  when Season = 'spring' then 1
  when Season = 'summer' then 2
  when Season = 'fall' then 3
  when Season = 'winter' then 4
  else .
end as Sorter
from chores)
order by Sorter;

/*-------------------------------------------------------------------
  Output 6.19 Sample Input Data to Conditionally Change a Table
-------------------------------------------------------------------*/
data incentives;
  input @1 Name $18. @20 Department $2. Payrate
  Gadgets Whatnots;
datalines;
Lao Che    M2     8.00  10193  1105
Jack Colton U2     6.00  9994  2710
Mickey Raymond M1   12.00  6103  1930
Dean Proffit  M2    11.00  3000  1999
Antoinette Lily E1  20.00  2203  4610
Sydney Wade     E2    15.00  4205  3010
Alan Traherne   U2     4.00  5020  3000
Elizabeth Bennett E1  16.00  17003 3003
proc sql;
  title 'Sales Data for Incentives Program';
  select * from incentives;
quit;

/*---------------------------------------------------------
Output 6.20 PROC SQL Output for Conditionally Updating a Table
Note: This example uses the data set Incentives that was created in
Output 6.19.
---------------------------------------------------------*/
proc sql;
  update incentives
    set payrate = case
      when gadgets > 10000 then
        payrate + 5.00
      when gadgets > 5000 then
        case
          when department in ('E1', 'E2') then
            payrate + 2.00
          else payrate + 3.00
        end
      else payrate
    end;
  update incentives
    set payrate = case
      when whatnots > 2000 then
        case
          when department in ('E2', 'M2', 'U2') then
            payrate + 1.00
          else payrate + 0.50
        end
      else payrate
    end;
  title 'Adjusted Payrates Based on Sales of Gadgets and Whatnots';
  select * from incentives;

/*---------------------------------------------------------
Output 6.21 Table with Updated Population Data
---------------------------------------------------------*/
proc sql;
  title 'Updated U.S. Population Data';
  select state, population format=comma10. label='Population' from sql.newpop;

/*---------------------------------------------------------
Output 6.22 Sql.UnitedStates with Updated Population Data (Partial Output)
---------------------------------------------------------*/
proc sql;
  title 'UnitedStates';
  update sql.unitedstates as u
    set population=(select population from sql.newpop as n
      where u.name=n.state)
where u.name in (select state from sql.newpop);
select Name format=$17., Capital format=$15.,
    Population, Area, Continent format=$13., Statehood format=date9.
from sql.unitedstates;
/* use this code to generate output so you don't
   overwrite the sql.unitedstates table */
options ls=84;
proc sql outobs=10;
  title 'UnitedStates';
  create table work.unitedstates as
  select * from sql.unitedstates;
  update work.unitedstates as u
    set population=(select population from sql.newpop as n
       where u.name=n.state)
    where u.name in (select state from sql.newpop);
  select Name format=$17., Capital format=$15.,
     Population, Area, Continent format=$13., Statehood format=date9.
  from work.unitedstates
 ;

/*-----------------------------------------------
   Output 6.23 Features (Partial Output)
-----------------------------------------------*/
libname sql 'SAS-library';
proc sql outobs=10;
  title 'Features';
  select Name format=$15., Type,Location format =$15.,Area,
     Height, Depth, Length
  from sql.features;
/*-----------------------------------------------
   Log 6.1 SAS Log After Creating a Separate Data Set for Each Unique
   Value of a Column
-----------------------------------------------*/
proc sql noprint;
  select count(distinct type)
    into :n
  from sql.features;
  select distinct type
    into :type1 - :type%left(&n)
  from sql.features;
quit;
%macro makeds;
%do i=1 %to &n;
  data &&type&i (drop=type);
    set sql.features;
    if type="&&type&i";
  run;
%end;
%mend makeds;
%makeds;
Appendix 4
Data Sets for Examples in SQL Procedure Reference

Overview

This section provides the DATA steps to create the tables used in the PROC SQL examples in this guide.

Note: If you copy and paste a DATA set to a SAS editor, the spacing between columns might change. Review the spacing between columns to ensure that the data can be read by the INPUT statement.
Employees

data Employees;
  input IdNum $4. +2 LName $11. FName $11. JobCode $3. +1 Salary 5. +1 Phone $12.;
datalines;
  1876 CHIN JACK TA1 42400 212/588-5634
  1114 GREENWALD JANICE ME3 38000 212/588-1092
  1556 PENNINGTON MICHAEL ME1 29860 718/383-5681
  1354 PARKER MARY FA3 65800 914/455-2337
  1130 WOOD DEBORAH PT2 36514 212/587-0013
;

Houses

data houses;
  input House $ x y;
datalines;
  house1 1 1
  house2 3 3
  house3 2 3
  house4 7 7
;

Match_11

data match_11;
  input Pair Low Age Lwt Race Smoke Ptd Ht UI @@;
  select(race);
    when (1) do;
      race1=0;
      race2=0;
    end;
    when (2) do;
      race1=1;
      race2=0;
    end;
    when (3) do;
      race1=0;
      race2=1;
    end;
  end;
datalines;
  1 0 14 135 1 0 0 0 0 1 1 14 101 3 1 1 0 0
  2 0 15 98 2 0 0 0 0 2 1 15 115 3 0 0 0 1
  3 0 16 95 3 0 0 0 0 3 1 16 130 3 0 0 0 0

Appendix 4 • Data Sets for Examples in SQL Procedure Reference
Proclib.Delay

data proclib.delay;
  input flight $3. +5 date date7. +2 orig $3. +3 dest $3. +3
delaycat $15. +2 destype $15. +8 delay;
  informat date date7.;
  format date date7.;
datalines;
114 01MAR08 LGA LAX 1-10 Minutes Domestic 8
202 01MAR08 LGA ORD No Delay Domestic -5
219 01MAR08 LGA LON 11+ Minutes International 18
622 01MAR08 LGA FRA No Delay International -5
132 01MAR08 LGA YYYZ 11+ Minutes International 14
271 01MAR08 LGA PAR 1-10 Minutes International 5
302 01MAR08 LGA WAS No Delay Domestic -2
114 02MAR08 LGA LAX No Delay Domestic 0
202 02MAR08 LGA ORD 1-10 Minutes Domestic 5
219 02MAR08 LGA LON 11+ Minutes International 18
622 02MAR08 LGA FRA No Delay International 0
132 02MAR08 LGA YYYZ 11+ Minutes International 5
271 02MAR08 LGA PAR 1-10 Minutes International 4
302 02MAR08 LGA WAS No Delay Domestic 0
114 03MAR08 LGA LAX No Delay Domestic -1
202 03MAR08 LGA ORD No Delay Domestic -1
219 03MAR08 LGA LON 1-10 Minutes International 4
622 03MAR08 LGA FRA No Delay International -2
132 03MAR08 LGA YYYZ 11-10 Minutes International 6
271 03MAR08 LGA PAR 1-10 Minutes International 2
302 03MAR08 LGA WAS 1-10 Minutes Domestic 5
114 04MAR08 LGA LAX 11+ Minutes Domestic 15
202 04MAR08 LGA ORD No Delay Domestic -3
219 04MAR08 LGA LON 1-10 Minutes International 3
622 04MAR08 LGA FRA 11+ Minutes International 10
132 04MAR08 LGA YYYZ No Delay International -5
271 04MAR08 LGA PAR 1-10 Minutes International 5
302 04MAR08 LGA WAS 1-10 Minutes Domestic 7
114 05MAR08 LGA LAX No Delay Domestic -2
202 05MAR08 LGA ORD 1-10 Minutes Domestic 2
219 05MAR08 LGA LON 1-10 Minutes International 3
622 05MAR08 LGA FRA No Delay International -6
132 05MAR08 LGA YYYZ 1-10 Minutes International 3
271 05MAR08 LGA PAR 1-10 Minutes International 5
114 06MAR08 LGA LAX No Delay Domestic -1
202 06MAR08 LGA ORD No Delay Domestic -3
219 06MAR08 LGA LON 11+ Minutes International 27
132 06MAR08 LGA YYYZ 1-10 Minutes International 7
302 06MAR08 LGA WAS 1-10 Minutes Domestic 1
114 07MAR08 LGA LAX No Delay Domestic -1
202 07MAR08 LGA ORD No Delay Domestic -2
219 07MAR08 LGA LON 11+ Minutes International 15
622 07MAR08 LGA FRA 11+ Minutes International 21
132 07MAR08 LGA YYYZ No Delay International -2
### Proclib.Houses

The contents of this data set are different from the “Houses” on page 440 data set. This data set is intended only for the “Example: INTO Clause” on page 256.

```sas
libname proclib 'SAS-library';

data proclib.houses;
input Style $ 1-8 SqFeet 15-18;
datalines;
CONDO          900
CONDO         1000
RANCH         1200
RANCH         1400
SPLIT         1600
SPLIT         1800
TWOSTORY      2100
TWOSTORY      3000
TWOSTORY      1940
TWOSTORY      1860
;
```

### Proclib.March

```sas
data proclib.march;
input flight $3. +5 date date7. +3 depart time5. +2 orig $3. +3 dest $3. +7 miles +6 boarded +6 capacity;
format date date7. depart time5. ;
informat date date7. depart time5. ;
datalines;
114     01MAR08    7:10  LGA   LAX       2475       172       210
202     01MAR08   10:43  LGA   ORD       740       151       210
219     01MAR08    9:31  LGA   LON       3442       198       250
622     01MAR08   12:19  LGA   FRA       3857       207       250
132     01MAR08   15:35  LGA   YYZ        366       115       178
271     01MAR08   13:17  LGA   PAR       3635       138       250
302     01MAR08   20:22  LGA   WAS        229        78       180
271     01MAR08   13:17  LGA   PAR       3635       138       250
114     02MAR08    7:10  LGA   LAX       2475       172       210
202     02MAR08   10:43  LGA   ORD       740       151       210
219     02MAR08    9:31  LGA   LON       3442       198       250
622     02MAR08   12:19  LGA   FRA       3857       207       250
132     02MAR08   15:35  LGA   YYZ        366       115       178
302     02MAR08   20:22  LGA   WAS        229        78       180
271     02MAR08   13:17  LGA   PAR       3635       138       250
114     03MAR08    7:10  LGA   LAX       2475       197       210
202     03MAR08   10:43  LGA   ORD       740       118       210
```
Proclib.Paylist2

proc sql;
  create table proclib.paylist2
  (IdNum char(4),
   Gender char(1),
   Jobcode char(3),
   Salary num,
   Birth num informat=date7.
     format=date7.,
   Hired num informat=date7.
     format=date7.);
  insert into proclib.paylist2
  values('1919','M','TA2',34376,'12SEP66'd,'04JUN87'd)
  values('1653','F','ME2',31896,'15OCT64'd,'09AUG92'd)
  values('1350','F','FA3',36886,'31AUG55'd,'29JUL91'd)
  values('1401','M','TA3',38822,'13DEC55'd,'17NOV93'd)
  values('1499','M','ME1',23025,'26APR74'd,'07JUN92'd);
proc sql;
title 'PROCLIB.PAYLIST2 Table';
select * from proclib.paylist2;

Proclib.Payroll

This data set is updated in “Example 3: Updating Data in a PROC SQL Table” on page 273. Its updated data is used in subsequent examples.

data proclib.payroll;
  input IdNumber $4. +3 Gender $1. +4 Jobcode $3. +9 Salary 5. +2 Birth date7. +2 Hired date7.;
  informat birth date7. hired date7.;
  format birth date7. hired date7.;
  datalines;
1919 M TA2 34376 12SEP60 04JUN87
1653 F ME2 35108 15OCT64 09AUG80
1400 M ME1 29769 05NOV67 16OCT90
1350 F FA3 32886 31AUG65 25JUL90
1401 M TA3 38822 13DEC92 17NOV85
1499 M ME3 43025 26APR54 07JUN80
1101 M SCP 18723 06JUN62 01OCT90
1333 M PT2 88606 30MAR61 10FEB81
1402 M TA2 32615 17JAN85 02DEC90
1479 F TA3 38785 11FEB90 05OCT89
1403 M ME1 28072 28JAN65 21DEC91
1739 M PT1 66517 25DEC64 27JAN91
1658 M SCP 17943 08APR67 29FEB92
1428 F PT1 68767 04APR60 16NOV91
1782 M ME2 35345 04DEC70 22FEB92
1244 M ME2 36925 31AUG63 17JAN88
1383 M BCK 25823 25JAN68 20OCT92
1574 M FA2 28572 27APR60 20DEC92
1789 M SCP 18326 25JAN57 11APR78
1404 M PT2 91376 24FEB92 01JAN80
1437 F FA3 33104 20SEP60 31AUG84
1639 F TA3 40260 26JUN57 28JAN84
1269 M NA1 41690 03MAY72 28NOV92
1065 M ME2 35090 26JAN44 07JAN87
1876 M TA3 39675 20MAY85 27APR85
1037 F TA1 28558 10APR64 13SEP92
1129 F ME2 34929 08DEC61 17AUG91
1988 M FA3 32217 30NOV92 18SEP94
1405 M SCP 18056 05MAR66 26JAN92
1430 F TA2 32925 28FEB92 27APR87
1983 F FA3 33419 28FEB67 27APR85
1134 F TA2 33462 05MAR69 21DEC88
1118 M PT3 111379 16JAN44 18DEC80
1438 F TA3 39223 15MAR65 18NOV92
1125 F FA2 28888 08NOV68 11DEC87
1475 F FA2 27787 15DEC61 13JUL90
1117 M TA3 39771 05JUN90 13AUG92
1935 F NA2 51081 28MAR84 16OCT81
1124 F FA1 23177 10JUL58 01OCT90
<table>
<thead>
<tr>
<th>ID</th>
<th>Gender</th>
<th>Name</th>
<th>Code</th>
<th>Date1</th>
<th>Date2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1422</td>
<td>F</td>
<td>FA1</td>
<td>22454</td>
<td>04JUN64</td>
<td>06APR91</td>
</tr>
<tr>
<td>1616</td>
<td>F</td>
<td>TA2</td>
<td>34137</td>
<td>01MAR70</td>
<td>04JUN93</td>
</tr>
<tr>
<td>1406</td>
<td>M</td>
<td>MB2</td>
<td>35185</td>
<td>08MAR61</td>
<td>17FEB87</td>
</tr>
<tr>
<td>1120</td>
<td>M</td>
<td>MB1</td>
<td>28619</td>
<td>11SEP72</td>
<td>07OCT93</td>
</tr>
<tr>
<td>1094</td>
<td>M</td>
<td>FA1</td>
<td>22268</td>
<td>02APR70</td>
<td>17APR91</td>
</tr>
<tr>
<td>1389</td>
<td>M</td>
<td>BCK</td>
<td>25028</td>
<td>15JUL59</td>
<td>18AUG90</td>
</tr>
<tr>
<td>1905</td>
<td>M</td>
<td>PT1</td>
<td>65111</td>
<td>16APR72</td>
<td>29MAY92</td>
</tr>
<tr>
<td>1407</td>
<td>M</td>
<td>PT1</td>
<td>68096</td>
<td>23MAR69</td>
<td>18MAR90</td>
</tr>
<tr>
<td>1114</td>
<td>F</td>
<td>TA2</td>
<td>32928</td>
<td>18SEP69</td>
<td>27JUN87</td>
</tr>
<tr>
<td>1439</td>
<td>F</td>
<td>PT1</td>
<td>70736</td>
<td>06MAR64</td>
<td>10SEP90</td>
</tr>
<tr>
<td>1409</td>
<td>M</td>
<td>MB3</td>
<td>41551</td>
<td>19APR50</td>
<td>22OCT81</td>
</tr>
<tr>
<td>1408</td>
<td>M</td>
<td>TA2</td>
<td>34138</td>
<td>29MAR60</td>
<td>14OCT87</td>
</tr>
<tr>
<td>1121</td>
<td>M</td>
<td>ME1</td>
<td>29112</td>
<td>26SEP71</td>
<td>07DEC91</td>
</tr>
<tr>
<td>1911</td>
<td>F</td>
<td>TA1</td>
<td>27645</td>
<td>07MAY72</td>
<td>12DEC92</td>
</tr>
<tr>
<td>1102</td>
<td>M</td>
<td>TA2</td>
<td>34542</td>
<td>01OCT59</td>
<td>15APR91</td>
</tr>
<tr>
<td>1356</td>
<td>M</td>
<td>ME2</td>
<td>36869</td>
<td>26SEP57</td>
<td>22FEB83</td>
</tr>
<tr>
<td>1545</td>
<td>M</td>
<td>PT1</td>
<td>66130</td>
<td>12AUG59</td>
<td>29MAY90</td>
</tr>
<tr>
<td>1292</td>
<td>F</td>
<td>MB2</td>
<td>36691</td>
<td>28OCT64</td>
<td>02JUL89</td>
</tr>
<tr>
<td>1440</td>
<td>F</td>
<td>MB2</td>
<td>35757</td>
<td>27SEP62</td>
<td>09APR91</td>
</tr>
<tr>
<td>1368</td>
<td>M</td>
<td>FA2</td>
<td>27808</td>
<td>11JUN61</td>
<td>03NOV84</td>
</tr>
<tr>
<td>1369</td>
<td>M</td>
<td>TA2</td>
<td>33705</td>
<td>28DEC61</td>
<td>13MAR87</td>
</tr>
<tr>
<td>1411</td>
<td>M</td>
<td>FA2</td>
<td>27265</td>
<td>25MAR61</td>
<td>01DEC89</td>
</tr>
<tr>
<td>1113</td>
<td>F</td>
<td>FA1</td>
<td>22367</td>
<td>15JAN68</td>
<td>17OCT91</td>
</tr>
<tr>
<td>1704</td>
<td>M</td>
<td>BCK</td>
<td>25465</td>
<td>30AUG66</td>
<td>28JUN87</td>
</tr>
<tr>
<td>1900</td>
<td>M</td>
<td>ME2</td>
<td>35105</td>
<td>25May62</td>
<td>27OCT87</td>
</tr>
<tr>
<td>1126</td>
<td>F</td>
<td>TA3</td>
<td>40899</td>
<td>28MAY63</td>
<td>21NOV80</td>
</tr>
<tr>
<td>1677</td>
<td>M</td>
<td>BCK</td>
<td>26007</td>
<td>05NOV63</td>
<td>27MAR89</td>
</tr>
<tr>
<td>1441</td>
<td>F</td>
<td>FA2</td>
<td>27158</td>
<td>19NOV69</td>
<td>23MAR91</td>
</tr>
<tr>
<td>1421</td>
<td>M</td>
<td>TA2</td>
<td>31555</td>
<td>08JAN59</td>
<td>28FEB90</td>
</tr>
<tr>
<td>1119</td>
<td>M</td>
<td>TA1</td>
<td>26924</td>
<td>20JUN62</td>
<td>06SEP88</td>
</tr>
<tr>
<td>1834</td>
<td>M</td>
<td>BCK</td>
<td>26896</td>
<td>08FEB72</td>
<td>02JUL92</td>
</tr>
<tr>
<td>1777</td>
<td>M</td>
<td>PT3</td>
<td>109630</td>
<td>23SEP51</td>
<td>21JUN81</td>
</tr>
<tr>
<td>1663</td>
<td>M</td>
<td>BCK</td>
<td>26452</td>
<td>11JAN67</td>
<td>11AUG91</td>
</tr>
<tr>
<td>1106</td>
<td>M</td>
<td>PT2</td>
<td>89632</td>
<td>06NOV57</td>
<td>16AUG84</td>
</tr>
<tr>
<td>1103</td>
<td>F</td>
<td>FA1</td>
<td>23738</td>
<td>16FEB68</td>
<td>23JUL92</td>
</tr>
<tr>
<td>1477</td>
<td>M</td>
<td>FA2</td>
<td>28566</td>
<td>21MAR64</td>
<td>07MAR88</td>
</tr>
<tr>
<td>1476</td>
<td>F</td>
<td>TA2</td>
<td>34803</td>
<td>30MAY66</td>
<td>17MAR87</td>
</tr>
<tr>
<td>1379</td>
<td>M</td>
<td>MB3</td>
<td>42264</td>
<td>08AUG61</td>
<td>10JUN84</td>
</tr>
<tr>
<td>1104</td>
<td>M</td>
<td>SCP</td>
<td>17946</td>
<td>25APR63</td>
<td>10JUN91</td>
</tr>
<tr>
<td>1009</td>
<td>M</td>
<td>TA1</td>
<td>28880</td>
<td>02MAR59</td>
<td>26MAR92</td>
</tr>
<tr>
<td>1412</td>
<td>M</td>
<td>ME1</td>
<td>27799</td>
<td>18JUN56</td>
<td>05DEC91</td>
</tr>
<tr>
<td>1115</td>
<td>F</td>
<td>FA3</td>
<td>32699</td>
<td>22AUG60</td>
<td>29FEB80</td>
</tr>
<tr>
<td>1128</td>
<td>F</td>
<td>TA2</td>
<td>32777</td>
<td>23MAR65</td>
<td>20OCT90</td>
</tr>
<tr>
<td>1442</td>
<td>F</td>
<td>PT2</td>
<td>84536</td>
<td>05SEP66</td>
<td>12APR88</td>
</tr>
<tr>
<td>1417</td>
<td>M</td>
<td>NA2</td>
<td>52270</td>
<td>27JUN64</td>
<td>07MAR89</td>
</tr>
<tr>
<td>1478</td>
<td>M</td>
<td>PT2</td>
<td>84203</td>
<td>09AUG59</td>
<td>24OCT90</td>
</tr>
<tr>
<td>1673</td>
<td>M</td>
<td>BCK</td>
<td>25477</td>
<td>27FEB70</td>
<td>15JUL91</td>
</tr>
<tr>
<td>1839</td>
<td>F</td>
<td>NA1</td>
<td>43433</td>
<td>29NOV70</td>
<td>03JUL93</td>
</tr>
<tr>
<td>1347</td>
<td>M</td>
<td>TA3</td>
<td>40079</td>
<td>21SEP67</td>
<td>06SEP84</td>
</tr>
<tr>
<td>1423</td>
<td>F</td>
<td>ME2</td>
<td>35773</td>
<td>14MAY68</td>
<td>19AUG90</td>
</tr>
<tr>
<td>1200</td>
<td>F</td>
<td>ME1</td>
<td>27816</td>
<td>10JAN71</td>
<td>14AUG92</td>
</tr>
<tr>
<td>1970</td>
<td>F</td>
<td>FA1</td>
<td>22615</td>
<td>25SEP64</td>
<td>12MAR91</td>
</tr>
<tr>
<td>1521</td>
<td>M</td>
<td>MB3</td>
<td>41526</td>
<td>12APR63</td>
<td>13JUL88</td>
</tr>
<tr>
<td>1354</td>
<td>F</td>
<td>SCP</td>
<td>18335</td>
<td>29MAR61</td>
<td>16JUN92</td>
</tr>
<tr>
<td>1424</td>
<td>F</td>
<td>FA2</td>
<td>28978</td>
<td>04AUG69</td>
<td>11DEC89</td>
</tr>
</tbody>
</table>
1132  F    FA1         22413  30MAY72  22OCT93
1845  M    BCK         25996  20NOV59  22MAR80
1556  M    PT1         71349  22JUN64  11DEC91
1413  M    FA2         27435  16SEP65  02JAN90
1123  F    TA1         28407  31OCT72  05DEC92
1907  M    TA2         33329  15NOV60  06JUL87
1436  F    TA2         34475  11JUN64  12MAR87
1385  M    MB3         43900  16JAN62  01APR86
1432  F    MB2         35327  03NOV61  10FEB85
1111  M    NA1         40586  14JUL62  28OCT90
1116  F    FA1         22862  28SEP69  21MAR91
1352  M    NA2         53798  02DEC60  16OCT86
1555  F    FA2         27499  16MAR68  21MAR92
1038  F    TA1         26533  09NOV64  23NOV92
1420  M    MB3         43071  19FEB65  22JUL87
1561  M    TA2         34514  30NOV63  25NOV91
1434  F    FA2         28622  11JUL62  28OCT90
1414  M    FA1         23644  24MAR72  12APR92
1112  M    TA1         26905  29NOV64  07DEC92
1390  M    FA2         27761  19FEB65  23JUN91
1332  M    NA1         42178  17SEP70  04JUN91
1890  M    PT2         91908  20JUL51  25NOV79
1429  F    TA1         27939  28FEB60  07AUG92
1107  M    PT2         89977  09JUN54  10FEB79
1908  F    TA2         32995  10DEC69  23APR90
1830  F    PT2         84471  27MAY57  29JAN83
1882  M    MB3         41538  10JUL63  24AUG86
1050  M    MB2         35167  19FEB65  23JUN90
1425  F    FA1         23979  28DEC71  28FEB93
1928  M    PT2         89858  16SEP54  13JUL90
1480  F    TA3         39583  03SEP57  25MAR81
1100  M    BCK         25004  01DEC60  07MAY88
1995  F    MB1         28810  24AUG73  19SEP93
1135  F    FA2         27321  20SEP60  31MAR90
1415  M    FA2         28278  09MAR58  12FEB88
1076  M    PT1         66558  14OCT55  03OCT91
1426  F    TA2         32991  05DEC66  25JUN90
1564  F    SCP         18833  12APR62  01JUL92
1221  F    FA2         27896  22SEP67  04OCT91
1133  M    TA1         27701  13JUL66  12FEB92
1435  F    TA3         38808  12MAY59  08FEB80
1418  M    MB1         28005  29MAR57  06JAN92
1017  M    TA3         40858  28DEC57  16OCT81
1443  F    NA1         42274  17NOV68  29AUG91
1131  F    TA2         32575  26DEC71  19APR91
1427  F    TA2         34046  31OCT70  30JAN90
1036  F    TA3         39392  19MAY65  23OCT84
1130  F    FA1         23916  16MAY71  05JUN92
1127  F    TA2         33011  09NOV64  07DEC86
1433  F    FA3         32982  08JUL66  17JAN87
1431  F    FA3         33230  09JUN64  05APR88
1122  F    FA2         27956  01MAY63  27NOV88
1105  M    MB2         34805  01MAR62  13AUG90

;
Proclib.Payroll2

data proclib.payroll2;
  input idnum $4. +3 gender $1. +4 jobcode $3. +9 salary 5. +2 birth date7. +2 hired date7.;
  informat birth date7. hired date7.;
  format birth date7. hired date7.;
  datalines;
  1639 F TA3 42260 26JUN57 28JAN84
  1065 M ME3 38090 26JAN44 07JAN87
  1561 M TA3 36514 30NOV63 07OCT87
  1221 F FA3 29896 22SEP67 04OCT91
  1447 F FA1 22123 07AUG72 29OCT92
  1998 M SCP 23100 10SEP70 02NOV92
  1036 F TA3 42465 19MAY65 23OCT91
  1106 M PT3 94039 06NOV57 16AUG84
  1129 F MB3 36758 08DEC61 17AUG91
  1350 F FA3 36098 31AUG65 29JUL90
  1369 M TA3 36598 28DEC61 13MAR87
  1076 M PT1 69742 14OCT55 03OCT91
 ;

Proclib.Schedule2

data proclib.schedule2;
  input flight $3. +5 date date7. +2 dest $3. +3 idnum $4.;
  format date date7.;
  informat date date7.;
  datalines;
  132 01MAR94 BOS 1118
  132 01MAR94 BOS 1402
  219 02MAR94 PAR 1616
  219 02MAR94 PAR 1478
  622 03MAR94 LON 1430
  622 03MAR94 LON 1882
  271 04MAR94 NYC 1430
  271 04MAR94 NYC 1118
  579 05MAR94 RDU 1126
  579 05MAR94 RDU 1106
 ;

Proclib.Staff

data proclib.staff;
  input idnum $4. +3 lname $15. +2 fname $15. +2 city $15. +2
<table>
<thead>
<tr>
<th>State</th>
<th>Name</th>
<th>Surname</th>
<th>Address</th>
<th>City</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT</td>
<td>ADAMS</td>
<td>GERALD</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-1255</td>
</tr>
<tr>
<td>CT</td>
<td>ALIBRANDI</td>
<td>MARIA</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-7715</td>
</tr>
<tr>
<td>NY</td>
<td>ALHERTANI</td>
<td>ABDULLAH</td>
<td>NEW YORK</td>
<td></td>
<td>212/586-0808</td>
</tr>
<tr>
<td>NY</td>
<td>ALVAREZ</td>
<td>MERCEDES</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-1549</td>
</tr>
<tr>
<td>NJ</td>
<td>ALVAREZ</td>
<td>CARLOS</td>
<td>PATerson</td>
<td></td>
<td>201/732-8787</td>
</tr>
<tr>
<td>NJ</td>
<td>BAREFOOT</td>
<td>JOSEPH</td>
<td>PRINCETON</td>
<td></td>
<td>201/812-5665</td>
</tr>
<tr>
<td>NY</td>
<td>BAUCOM</td>
<td>WALTER</td>
<td>NEW YORK</td>
<td></td>
<td>212/586-0806</td>
</tr>
<tr>
<td>NY</td>
<td>BANADYGA</td>
<td>JUSTIN</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-1777</td>
</tr>
<tr>
<td>NY</td>
<td>BALLETTI</td>
<td>MARIE</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-8816</td>
</tr>
<tr>
<td>CT</td>
<td>BOWDEN</td>
<td>EARL</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-3434</td>
</tr>
<tr>
<td>NY</td>
<td>BAREFOOT</td>
<td>JOSEPH</td>
<td>NEW YORK</td>
<td></td>
<td>212/587-1247</td>
</tr>
<tr>
<td>CT</td>
<td>BRIEUHAS</td>
<td>JEREMY</td>
<td>NEW YORK</td>
<td></td>
<td>212/587-3622</td>
</tr>
<tr>
<td>CT</td>
<td>BRADY</td>
<td>CHRISTINE</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-1212</td>
</tr>
<tr>
<td>CT</td>
<td>BREWCZAK</td>
<td>JAKOB</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-0019</td>
</tr>
<tr>
<td>NY</td>
<td>BUCCI</td>
<td>ANTHONY</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-3334</td>
</tr>
<tr>
<td>NY</td>
<td>BURNETTE</td>
<td>THOMAS</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-3569</td>
</tr>
<tr>
<td>NY</td>
<td>CAHILL</td>
<td>MARSHALL</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-2338</td>
</tr>
<tr>
<td>NJ</td>
<td>CARAWAY</td>
<td>DAVIS</td>
<td>NEW YORK</td>
<td></td>
<td>212/587-9000</td>
</tr>
<tr>
<td>NY</td>
<td>COHEN</td>
<td>LEE</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-2946</td>
</tr>
<tr>
<td>CT</td>
<td>CARTER</td>
<td>DOROTHY</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-4117</td>
</tr>
<tr>
<td>CT</td>
<td>CASTON</td>
<td>KAREN</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-8839</td>
</tr>
<tr>
<td>CT</td>
<td>CASTON-COHEN</td>
<td>KAREN</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-8839</td>
</tr>
<tr>
<td>CT</td>
<td>CHOW</td>
<td>JANE</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-8868</td>
</tr>
<tr>
<td>NY</td>
<td>COUNIHAN</td>
<td>BRENDA</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-2313</td>
</tr>
<tr>
<td>NY</td>
<td>COOPER</td>
<td>ANTHONY</td>
<td>NEW YORK</td>
<td></td>
<td>212/587-1228</td>
</tr>
<tr>
<td>NJ</td>
<td>DACKO</td>
<td>JASON</td>
<td>PATerson</td>
<td></td>
<td>201/732-3232</td>
</tr>
<tr>
<td>CT</td>
<td>DABROWSKI</td>
<td>SANDRA</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-1647</td>
</tr>
<tr>
<td>NY</td>
<td>DEAN</td>
<td>SHARON</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-1647</td>
</tr>
<tr>
<td>CT</td>
<td>DELGADO</td>
<td>MARIA</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-1528</td>
</tr>
<tr>
<td>NY</td>
<td>DENNIS</td>
<td>ROGER</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-1122</td>
</tr>
<tr>
<td>CT</td>
<td>DABBOUSSI</td>
<td>KAMILLA</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-2229</td>
</tr>
<tr>
<td>NY</td>
<td>DUNLAP</td>
<td>DONNA</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-2094</td>
</tr>
<tr>
<td>NY</td>
<td>ELGES</td>
<td>MARGARETE</td>
<td>NEW YORK</td>
<td></td>
<td>718/383-2828</td>
</tr>
<tr>
<td>NY</td>
<td>EDGERTON</td>
<td>JOSHUA</td>
<td>NEW YORK</td>
<td></td>
<td>212/588-1239</td>
</tr>
<tr>
<td>CT</td>
<td>FERNANDEZ</td>
<td>KATRINA</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-2962</td>
</tr>
<tr>
<td>NY</td>
<td>FIELDS</td>
<td>DIANA</td>
<td>WHITE PLAINS</td>
<td></td>
<td>914/455-2998</td>
</tr>
<tr>
<td>NJ</td>
<td>FUJIHARA</td>
<td>KYOKO</td>
<td>PRINCETON</td>
<td></td>
<td>201/812-0902</td>
</tr>
<tr>
<td>NY</td>
<td>FUENTAS</td>
<td>CARLA</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-3329</td>
</tr>
<tr>
<td>CT</td>
<td>FOSTER</td>
<td>GERALD</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-6363</td>
</tr>
<tr>
<td>NY</td>
<td>GARCIA</td>
<td>JACK</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-4930</td>
</tr>
<tr>
<td>CT</td>
<td>GOMEZ</td>
<td>ALAN</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-7181</td>
</tr>
<tr>
<td>NY</td>
<td>GOLDSTEIN</td>
<td>LEVI</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-9326</td>
</tr>
<tr>
<td>NY</td>
<td>GRAHAM</td>
<td>ALVIN</td>
<td>NEW YORK</td>
<td></td>
<td>212/586-8815</td>
</tr>
<tr>
<td>NY</td>
<td>GREGORSKI</td>
<td>DANIEL</td>
<td>MT. VERNON</td>
<td></td>
<td>914/468-1616</td>
</tr>
<tr>
<td>NY</td>
<td>GREENWALD</td>
<td>JANICE</td>
<td>NEW YORK</td>
<td></td>
<td>212/588-1092</td>
</tr>
<tr>
<td>CT</td>
<td>HARRIS</td>
<td>CHARLES</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-0937</td>
</tr>
<tr>
<td>CT</td>
<td>HASENHAUER</td>
<td>CHRISTINA</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-4987</td>
</tr>
<tr>
<td>CT</td>
<td>HAVELKA</td>
<td>RAYMOND</td>
<td>STAMFORD</td>
<td></td>
<td>203/781-9697</td>
</tr>
<tr>
<td>NJ</td>
<td>HENDERSON</td>
<td>WILLIAM</td>
<td>PRINCETON</td>
<td></td>
<td>201/812-4789</td>
</tr>
<tr>
<td>NY</td>
<td>HERNANDEZ</td>
<td>ROBERTO</td>
<td>NEW YORK</td>
<td></td>
<td>718/384-3313</td>
</tr>
<tr>
<td>CT</td>
<td>HOWARD</td>
<td>GRETCHEN</td>
<td>BRIDGEPORT</td>
<td></td>
<td>203/675-0007</td>
</tr>
<tr>
<td>IdNum</td>
<td>Lname</td>
<td>Fname</td>
<td>City</td>
<td>State</td>
<td>Hphone</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>-----------------</td>
</tr>
<tr>
<td>1561</td>
<td>SANDERS</td>
<td>RAYMOND</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/588-6615</td>
</tr>
<tr>
<td>1434</td>
<td>SANDERSON</td>
<td>EDITH</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-1333</td>
</tr>
<tr>
<td>1414</td>
<td>SANDERSON</td>
<td>NATHAN</td>
<td>BRIDGEPORT</td>
<td>CT</td>
<td>203/675-1715</td>
</tr>
<tr>
<td>1112</td>
<td>SANYERS</td>
<td>RANDY</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-4895</td>
</tr>
<tr>
<td>1390</td>
<td>SMART</td>
<td>JONATHAN</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-1141</td>
</tr>
<tr>
<td>1332</td>
<td>STEPHENSON</td>
<td>ADAM</td>
<td>BRIDGEPORT</td>
<td>CT</td>
<td>203/675-1497</td>
</tr>
<tr>
<td>1890</td>
<td>STEPHENSON</td>
<td>ROBERT</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-9874</td>
</tr>
<tr>
<td>1429</td>
<td>THOMPSON</td>
<td>ALICE</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-3857</td>
</tr>
<tr>
<td>1107</td>
<td>THOMPSON</td>
<td>WAYNE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-3785</td>
</tr>
<tr>
<td>1908</td>
<td>TRENTON</td>
<td>MELISSA</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-6262</td>
</tr>
<tr>
<td>1830</td>
<td>TRIPP</td>
<td>KATHY</td>
<td>BRIDGEPORT</td>
<td>CT</td>
<td>203/675-2479</td>
</tr>
<tr>
<td>1882</td>
<td>TUCKER</td>
<td>ALAN</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-0216</td>
</tr>
<tr>
<td>1050</td>
<td>TUTTLE</td>
<td>THOMAS</td>
<td>WHITE PLAINS</td>
<td>NY</td>
<td>914/455-2119</td>
</tr>
<tr>
<td>1425</td>
<td>UNDERWOOD</td>
<td>JENNY</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-0978</td>
</tr>
<tr>
<td>1928</td>
<td>UPCHURCH</td>
<td>LARRY</td>
<td>WHITE PLAINS</td>
<td>NY</td>
<td>914/455-5009</td>
</tr>
<tr>
<td>1480</td>
<td>UPDIKE</td>
<td>THERESA</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/587-8729</td>
</tr>
<tr>
<td>1100</td>
<td>VANDUSEN</td>
<td>RICHARD</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-2531</td>
</tr>
<tr>
<td>1995</td>
<td>VARNER</td>
<td>ELIZABETH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-7113</td>
</tr>
<tr>
<td>1135</td>
<td>VEGA</td>
<td>ANNA</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-5913</td>
</tr>
<tr>
<td>1415</td>
<td>VEGA</td>
<td>FRANKLIN</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-2823</td>
</tr>
<tr>
<td>1076</td>
<td>VENTER</td>
<td>RANDALL</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-2321</td>
</tr>
<tr>
<td>1426</td>
<td>VICK</td>
<td>THERESA</td>
<td>PRINCETON</td>
<td>NJ</td>
<td>201/812-2424</td>
</tr>
<tr>
<td>1564</td>
<td>WALTERS</td>
<td>ANNE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/587-3257</td>
</tr>
<tr>
<td>1221</td>
<td>WALTERS</td>
<td>DIANE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-1918</td>
</tr>
<tr>
<td>1133</td>
<td>WANG</td>
<td>CHIN</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/587-1956</td>
</tr>
<tr>
<td>1435</td>
<td>WARD</td>
<td>ELAINE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-4987</td>
</tr>
<tr>
<td>1418</td>
<td>WATSON</td>
<td>BERNARD</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-1298</td>
</tr>
<tr>
<td>1017</td>
<td>WELCH</td>
<td>DARUS</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/586-5535</td>
</tr>
<tr>
<td>1443</td>
<td>WELLS</td>
<td>AGNES</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-5546</td>
</tr>
<tr>
<td>1131</td>
<td>WELLS</td>
<td>NADINE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/383-1045</td>
</tr>
<tr>
<td>1427</td>
<td>WHALEY</td>
<td>CAROLYN</td>
<td>MT. VERNON</td>
<td>NY</td>
<td>914/468-4528</td>
</tr>
<tr>
<td>1036</td>
<td>WONG</td>
<td>LESLIE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/587-2570</td>
</tr>
<tr>
<td>1130</td>
<td>WOOD</td>
<td>DEBORAH</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/587-0013</td>
</tr>
<tr>
<td>1127</td>
<td>WOOD</td>
<td>SANDRA</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/587-2881</td>
</tr>
<tr>
<td>1433</td>
<td>YANCEY</td>
<td>ROBIN</td>
<td>PRINCETON</td>
<td>NJ</td>
<td>201/812-1874</td>
</tr>
<tr>
<td>1431</td>
<td>YOUNG</td>
<td>DEBORAH</td>
<td>STAMFORD</td>
<td>CT</td>
<td>203/781-2987</td>
</tr>
<tr>
<td>1122</td>
<td>YOUNG</td>
<td>JOANN</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-2021</td>
</tr>
<tr>
<td>1105</td>
<td>YOUNG</td>
<td>LAWRENCE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-0008</td>
</tr>
</tbody>
</table>

Proclib.Staff2

```plaintext
data proclib.staff2;
input IdNum $4. @7 Lname $12. @20 Fname $8. @30 City $10. @42 State $2. @50 Hphone $12. ;
datalines;
1106  MARSHBURN  JASPER  STAMFORD  CT     203/781-1457
1430  DABROWSKI  SANDRA  BRIDGEPORT  CT    203/675-1647
1118  DENNIS     ROGER   NEW YORK     NY    718/383-1122
1126  KIMANI     ANNE    NEW YORK     NY    212/586-1229
1402  BLALOCK    RALPH   NEW YORK     NY    718/384-2849
1882  TUCKER     ALAN    NEW YORK     NY    718/384-0216
1479  BALLETTI   MARIE   NEW YORK     NY    718/384-8816
;```
Proclib.Superv2

```sql
data proclib.superv2;
  input supid $4. +8 state $2. +5  jobcat $2.;
  label supid='Supervisor Id' jobcat='Job Category';
datalines;
  1417   NJ   NA
  1352   NY   NA
  1106   CT   PT
  1442   NJ   PT
  1118   NY   PT
  1405   NJ   SC
  1564   NY   SC
  1639   CT   TA
  1126   NY   TA
  1882   NY   ME
;
```

Stores

```sql
data stores;
  input Store $ x y;
datalines;
  store1 5 1
  store2 5 3
  store3 3 5
  store4 7 5
;
```

Survey

```sql
data survey;
  input id $ diet $ exer $ hours xwk educ;
datalines;
  1001 yes yes 1 3 1
  1002 no  yes 1 4 2
  1003 no  no  . . n
  1004 yes yes 2 3 .x
  1005 no  yes 2 3 .x
  1006 yes yes 2 4 .x
  1007 no  yes .5 3 .
  1008 no  no  . . .
;
```
Recommended Reading

Here is the recommended reading list for this title:

- Base SAS Procedures Guide
- Combining and Modifying SAS Data Sets: Examples
- The Essential PROC SQL Handbook for SAS Users
- Little SAS Book: A Primer
- PROC SQL by Example: Using SQL within SAS
- PROC SQL: Beyond the Basics Using SAS
- SAS/ACCESS for Relational Databases: Reference
- SAS Data Set Options: Reference
- SAS Formats and Informats: Reference
- SAS Functions and CALL Routines: Reference
- SAS/GRAPH: Reference
- SAS Guide to Report Writing: Examples
- SAS Language Reference: Concepts
- SAS Macro Language: Reference
- SAS DATA Step Statements: Reference

- SAS offers instructor-led training and self-paced e-learning courses to help you get started with SAS Foundation. For more information about the courses available, see support.sas.com/training.

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books
calculated column
a column that does not exist in any of the tables that are accessed, but which is created as a result of a column expression. See also column expression.

Cartesian product
a type of join that matches each row from each joined table to each row from all other joined tables. See also cross join, join.

column alias
a temporary, alternate name for a column. Aliases can be specified in the SQL procedure's SELECT clause to name or rename columns. An alias is one word.

column expression
a set of operators and operands that, when evaluated, result in a single data value. The resulting data value can be either a character value or a numeric value.

composite index
an index that locates observations in a SAS data set by examining the values of two or more key variables. See also simple index.

conditional operator
in the SQL procedure, the part of the WHERE clause that specifies which rows are to be retrieved.

cross join
a type of join that returns the product of joined tables. A cross join is functionally the same as a Cartesian product. See also Cartesian product, join.

data set
See SAS data set.

data view
See SAS data view.

DISTINCT
a keyword that causes the SQL procedure to remove duplicate rows from the output.
equijoin
a type of join in the SQL procedure. For example, when two tables are joined in an equijoin, the value of a column in the first table must equal the value of the column in the second table in the SQL expression. See also join.

format
See SAS format.

group
a set of rows or observations that have the same value or values for one or more common columns or variables.

in-line view
a query expression that is nested in the SQL procedure's FROM clause. An in-line view produces a table internally that the outer query uses to select data. You save a programming step when you use an in-line view, because instead of creating a view and then referring to it in another query, you can specify the view in-line in the FROM clause. An in-line view can be referenced only in the query (or statement) in which it is defined.

index
See SAS index.

inner join
a join between two tables that returns all of the rows in one table that have one or more matching rows in the other table.

integrity constraint
a data validation rule that restricts the data values that can be stored for a variable in a SAS data file. Integrity constraints help preserve the validity and consistency of the data.

join
an operation that combines data from two or more tables. A join is typically created by means of SQL (Structured Query Language) code or a user interface. See also outer join.

join criteria
the set of parameters that determine how tables are to be joined. Join criteria are usually specified in a WHERE expression or in an SQL ON clause. See also join, outer join.

missing value
a type of value for a variable that contains no data for a particular row or column. By default, SAS writes a missing numeric value as a single period and a missing character value as a blank space.

natural join
a type of join that returns selected rows from tables in which one or more columns in each table have the same name and the same data type and contain the same value. See also join.

outer join
a join between two tables that returns all of the rows in one table, as well as part or all of the rows in the other table. A left or right outer join returns all of the rows in one table (the table on the left or right side of the SQL statement, respectively), as
well as the matching rows in the other table. A full outer join returns all of the rows in both of the tables. See also join.

**pass-through facility**
See SQL pass-through facility.

**PROC SQL view**
a SAS data set that is created by the SQL procedure. A PROC SQL view contains no data. Instead, it stores information that enables it to read data values from other files, which can include SAS data files, SAS/ACCESS views, DATA step views, or other PROC SQL views. The output of a PROC SQL view can be either a subset or a superset of one or more files. See also SAS data view.

**query**
a set of instructions that requests particular information from one or more data sources.

**query expression**
in PROC SQL, a SELECT statement that references at least one table and, when executed, creates a temporary table that exists only during the execution of the statement. You can combine the results of multiple table expressions with set operators to create a query expression.

**SAS data file**
a type of SAS data set that contains data values as well as descriptor information that is associated with the data. The descriptor information includes information such as the data types and lengths of the variables, as well as the name of the engine that was used to create the data. See also SAS data set, SAS data view.

**SAS data set (data set)**
a file whose contents are in one of the native SAS file formats. There are two types of SAS data sets: SAS data files and SAS data views. See also SAS data file, SAS data view.

**SAS data view (data view)**
a type of SAS data set that retrieves data values from other files. A SAS data view contains only descriptor information such as the data types and lengths of the variables (columns) plus other information that is required for retrieving data values from other SAS data sets or from files that are stored in other software vendors’ file formats.

**SAS format (format)**
a type of SAS language element that is used to write or display data values according to the data type: numeric, character, date, time, or timestamp.

**SAS index (index)**
a component of a SAS data set that enables SAS to access observations in the SAS data set quickly and efficiently. The purpose of SAS indexes is to optimize WHERE-clause processing and to facilitate BY-group processing. See also composite index, simple index.

**SAS variable (variable)**
an identifier that is output as a column in a SAS data set or in a SAS data view. The data values for each variable describe a single characteristic for all observations (rows).
simple index
an index that uses the values of only one variable to locate observations. See also composite index.

SQL
See Structured Query Language.

SQL pass-through facility (pass-through facility)
the technology that enables SQL query code to be passed to a particular DBMS for processing.

Structured Query Language (SQL)
a standardized, high-level query language that is used in relational database management systems to create and manipulate objects in a database management system. SAS implements SQL through the SQL procedure.

union join
a type of join that returns all rows with their respective values from each input table. Columns that do not exist in one table will have null (missing) values for those rows in the result table. See also join.

variable
See SAS variable.

view
a definition of a virtual data set that is named and stored for later use. A view contains no data; it merely describes or defines data that is stored elsewhere.

WHERE clause
a syntax string that is composed of the keyword WHERE, followed by one or more WHERE expressions. A WHERE clause defines the conditions to be used for selecting observations in a data set. See also WHERE expression.

WHERE expression
is a syntax string within a WHERE clause that defines the criteria for selecting observations. For example, in a membership database, the expression "WHERE member_type=Senior" returns all senior members.
Index

A
abbreviating column names 76
aggregate functions 56
 creating macro variables from result of
    HAVING clause with 70
    table of 56
    using 56
    with unique values 61
alias
    assigning a column alias 30
    referring to calculated columns by 31
    table aliases 76
ALL keyword 345
 set operators and 143
ALTER TABLE statement 232
ANSI standard
 SQL procedure and 387
 arithmetic operators 388
 asterisk (*) notation 254
 automatic macro variables 157, 162
 averages, weighted 178

B
BETWEEN condition 316
BETWEEN-AND operators
 retrieving rows 51
 Boolean operators
 retrieving rows 47
BTRIM function 316

C
CALCULATED 317
calculated columns 29
 assigning column alias to 30
 referring to by alias 31
 sorting by 40
 SQL 318
 Cartesian product 75, 328, 330
cross joins 87
CASE expression 318
 assigning values conditionally 32
case-control studies 310
CASE-OPERAND form
 assigning values conditionally 34
character strings
 converting to lowercase 343
 converting to uppercase 369
 trimming 317
COALESCE function 319
 in joins 91
 replacing missing values 35
collating sequence
 alternate 388
column alias 30
 ANSI standard extensions 147
 assigning to calculated columns 30
 referring to calculated columns 31
 specifying 146
column attributes 235, 321
 list of 27
 specifying 36
 column definitions
 creating tables from 112
 column headers
 suppressing 28
 column modifiers 388
 column names
 abbreviating 76
 qualifying 76
 column-definition component 320
column-modifier component 321
column-name component 323
 columns 4
 adding 123
 aliases 146
 altering 123, 232
 assigning values conditionally 32
 calculated 317, 318
 calculating values 29
 changing formats 125
 changing informats 125
 changing labels 125
 changing width 125
 combinations of values 304
 creating 27
deleting 126
DICTIONARY.COLUMNS 154
  grouping by multiple columns 65
  grouping by one column 64
  indexes on 236, 237, 252
inserting values 251
  length of 321
list of, with attributes 27
locating specific columns 154
modifying 388
  modifying 125
  multicolumn joins 80
renaming 125, 235, 252
replacing missing values 35
returning values 320
selecting 22, 252, 323
  selecting all 22
selecting specific 23
sorting by 38
  sorting by column position 41
  sorting by multiple columns 39
  sorting by unselected columns 42
  sorting, with missing values 43
SQL procedure 216
storing values of 255
  summarizing data in multiple columns 192
  unique values 25
  updating values 267
COMMIT statement 389
comparison operators
  inner joins with 78
  retrieving rows with 45
  truncated string 53
composite indexes 127, 238
concatenating
  query results 107
  values in macro variables 160
conditional operators
  retrieving rows with 49
CONNECT statement 236
CONNECTION TO component 324
CONSTDATETIME option 222
CONTAINS condition 324, 388
correlated subqueries 98, 357
CORRESPONDING keyword 345
COUNT(*) function 362
counting
  all rows 62
  duplicate rows 187
  nonmissing values 62
  unique values 61
CREATE INDEX statement 127, 237
CREATE TABLE statement 238
CREATE VIEW statement 243
cross joins 87, 334

D
  data files
    See tables
  data set options
    creating tables with 115
    SQL procedure with 156
  DATA step
    compared with SQL procedure 5
    match-merges 92
  DATA step views
    SQL procedure 217
data summaries 361
  DATE function
    replacing references to 146
  DATETIME function
    replacing references to 146
  DBMS
    accessing with SAS/ACCESS 167
    connecting with LIBNAME statement 168
    connecting with Pass-Through Facility 171
deploying PUT function and SAS formats 145
  DBMS connections
    ending 248
    sending DBMS statements to 249
    SQL procedure 236
    storing in views 245
  DBMS queries 324
  DBMS tables 4
    PROC SQL views of 170
    querying 169
    debugging queries 138
    DELETE statement 245
    DESCRIBE statement 247
    DESCRIBE VIEW statement 152
    DICTDIAG option 223
  DICTIONARY tables 149
    performance and 155
    reporting from 282
    retrieving information about 151
    using 153
    views and 149
  DICTIONARY.COLUMNS 154
  DICTIONARY.TABLES 153
  DISCONNECT statement 248
  DOUBLE option 223
  DQUOTE= option 223
  DROP statement 248

E
  embedded LIBNAME statements 245
  embedded SQL 390
equijoins 328
errors
caused by missing values 63

grouping errors caused by missing values 67

update errors 122
ERRORSTOP option 223

elementary tables 7
EXCEPT operator 349
    - combining queries 103, 105
EXEC option 224
EXECUTE statement 249
execution time 140
existence of a group of values 99
EXISTS condition 99, 325
EXITCODE option 224
expanded SELECT * statement 140

F
FEEDBACK option 224
    - expanding SELECT * statement with 140
fields
    See columns
files
    See tables
filtering grouped data 69
    - HAVING clause versus WHERE clause 70
    - using a simple HAVING clause 69
    - using HAVING clause with aggregate functions 70
FLOW option 224
foreign key 129
formats
    - changing column format 125
    - deploying inside a DBMS 145
    - for columns 321
FROM clause 20, 261
full outer joins 86
functions
    - FCMP procedure 389
    - SQL procedure and 389
    - sql-expression and 352

G
general integrity constraints 129
GROUP BY clause 21, 263

grouping data 64
    - by multiple columns 65
    - by one column 64
    - filtering grouped data 69
    - finding errors caused by missing values 67
    - grouping and sorting 66
with missing values 67
without summarizing 64

H
HAVING clause 21, 264
    - aggregate functions with 70
    - filtering grouped data 69
    - filtering grouped data, versus WHERE clause 70
    - hierarchical data
        - expanding in tables 189
        - host-variable references 157

I
IN condition 325
IN operator
    - multiple-value subqueries and 97
retrieving rows 50
in-database procedures
    - generating SQL for 374
in-line views 134, 262, 388
    - querying 297
temporary tables versus 143
indexes 127
    - composite 127
    - composite indexes 238
    - creating 127
    - creating with CREATE INDEX statement 127
    - deleting 128, 248
    - managing 238
    - on altered columns 236
    - on columns 237, 252
    - query performance and 142
    - simple indexes 238
    - SQL procedure 237
tips for creating 127
    - UNIQUE keyword 237
    - unique values 127
informs
    - changing column informat 125
    - for columns 321
INNER JOIN keywords 78
inner joins 75, 329
    - comparison operators for 78
    - creating with INNER JOIN keywords 78
data from multiple tables 82
multicolumn joins 80
null values and 79
order of output 77
reflexive joins 83
self-joins 83
showing relationships within a table 83
table aliases 76
INOBSE= option 225
restricting row processing 139
INSERT statement 250
inserting rows 116
with queries 118
with SET clause 116
with VALUES clause 117
integrity constraints 129
PROC SQL tables 236, 243
referential 129
INTERSECT operator 350
combining queries 103, 106
INTO clause 255
IPASSTHRU option 225
IS condition 326
IS MISSING operator
retrieving rows 50
IS NOT MISSING operator
inner joins 80
iterations
limiting 139

J
joined-table component 327
joins 74, 328
Cartesian product 75
COALESCE function in 91
combining with subqueries 101
comparing match-merges with 92
comparing with subqueries 143
cross joins 87, 334
equijoins 328
inner joins 75, 329
joining a table with itself 329
joining more than two tables 338
joining three tables 293
joining two tables 276
natural joins 89, 336
outer joins 84, 284, 332, 388
reducing size of results 143
reflexive joins 329
rows to be returned 328
specialty joins 87
subqueries compared with 341
table limit 328
types of 328
union joins 89, 335
when to use 102
WHERE expressions with 143

L
labels
changing column labels 125
for columns 321
left outer joins 84
libname engines
querying DBMS tables 169
LIBNAME statement
connecting to a DBMS 168
embedding in views 245
libnames
embedding in views 133
librefs
stored views and 244
LIKE condition 341
patterns for searching 342
searching for literals 342
searching for mixed-case strings 343
LIKE operator
retrieving rows 52
log
displaying SQL definitions 247
logical operators
retrieving rows 47
LOOPS= option 225
limiting iterations 139
LOWER function 343

M
macro facility
SQL procedure with 157
macro variables 157
concatenating values in 160
creating and using 205
creating from aggregate function results 158
creating from query results 158
creating in SQL procedure 157
creating multiple 159
set by SQL procedure 162
SYS_SQLSETLIMIT 385
macros
counting missing values 313
defining to create tables 161
match-merges 92
comparing with joins 92
when all values match 92
when position of values is important 94
when some values match 93
matching patterns 310, 341
MEAN function
summarizing data 57
WHERE clause with 57
merging
disabling remerge 146
remerging summary statistics 59
merging data
SQL procedure 363
missing values 5
counting with a macro 313
finding errors caused by 63
finding grouping errors caused by 67
grouping data with 67
overlaying 182
replacing in columns 35
retrieving rows and 50
sorting columns with 43
SQL procedure 313, 326
summarizing data with 62
WHERE clause with 54
multipocolum joins 80
multiple-value subqueries 97

N
natural joins 89, 336
nested subqueries 100
NOEXEC option
  syntax checking with 139
NOT IN operator
  multiple-value subqueries and 97
null values 5
  inner joins and 79
NUMBER option 226

O
observations
  See also rows
SQL procedure 216
ODS (Output Delivery System)
  SQL procedure with 174
ODS destinations 174
operands
  values from 351
operators
  arithmetic 388
  order of evaluation 352
  set operators 344, 389
  truncated string comparison operators 354
  values from 351
ORDER BY clause 21, 265, 388
omitting 143
query performance and 143
orthogonal expressions 388
outer joins 84, 284, 332, 388
  full outer joins 86
  including nonmatching rows 84, 85
  left outer joins 84
  right outer joins 85
OUTER UNION operator
  combining queries 103
  concatenating query results 107
OUTER UNION set operator 345
OUTOBS= option 226
  restricting row processing 139
output
  adding text to 27
  formatting with REPORT procedure 165
output objects 174
overlaying missing values 182

P
pattern matching 310, 341
percentages
  computing within subtotals 185
performance
  queries 142
primary key 129
PRINT option 226
PROC SQL statement 221
PROC SQL tables 216
  adding rows 250
  aliases 262, 329
  altering columns 232
  altering integrity constraints 232
  changing column attributes 235
  combining 280
  counting rows 362
  creating 238, 269
  creating, from query expressions 242
  creating, from query results 271
  deleting 248
  deleting rows 245, 246
  indexes on columns 236
  initial values of columns 235
  inner join 278
  inserting data 269
  inserting values 251
  integrity constraints 236, 243
  joining 276, 301, 327
  joining a table with itself 327, 329
  joining more than two tables 338
  joining three tables 293
  ordering rows 265
  recursive table references 242
  renaming columns 235
  retrieving table references 242
  selecting columns 252
  selecting rows 252
  source tables 261
  table definitions 247
  table expressions 344
  updating 267, 268, 273
  without rows 242
PROC SQL views
  See also views
adding rows 250
creating, from query expressions 242
creating, from query results 290
deleting 248
deleting rows 245, 246
embedding LIBNAME statements in 245
inserting rows 252
librefs and stored views 244
selecting columns 243, 252
selecting rows 252
sorting data retrieved by source views 261
SQL procedure 217
storing DBMS connection information 245
updating 173, 245
updating column values 267
updating tables through view definitions 247, 388
PROMPT option 227
PUT function
deploying inside a DBMS 145
mapping to SAS_PUT function 378
optimizing 144
reducing 144
reducing, based number of format values 381
reducing, based on engine type 379
reducing, based on number of observations in table 380

Q
qualifying column names 76
queries 5
adding text to output 27
ALL keyword in set operations 143
breaking into steps 143
combining with set operators 103
comparing execution time of two queries 140
creating 138
creating tables from results 271
creating views from results 290
DBMS queries 324
DBMS tables 169
debugging 138
duplicate rows and performance 143
in-line view queries 297
in-line views 134
in-line views versus temporary tables 143
indexes and 142
inserting rows with 118
limiting iterations 139
performance improvement 142
remerged data and 383
restricting row processing 139
subqueries 96
validating 71
query expression component 343
query expressions 344
ALL keyword and 345
CORRESPONDING keyword and 345
creating PROC SQL tables from 242
creating PROC SQL views from 244
EXCEPT and 349
INTERSECT and 350
OUTER UNION and 345
set operators and 344
subqueries 354
UNION and 348
validating syntax 268
query results 5
concatenating 107
creating creating macro variables from 158
creating tables from 113
deleting duplicate rows 25

R
records
See rows
REDUCEPUT option 144
REDUCEPUT= option 227
REDUCEPUTOBS option 144
REDUCEPUTOBS= option 227
REDUCEPUTVALUES option 145
REDUCEPUTVALUES= option 228
referential integrity constraints 129
reflexive joins 83, 329
relational theory 3
relations 3
REMERGE option 229
remerging data 383
SQL procedure 363
remerging summary statistics 59
disabling remerge 146
renaming columns 125
REPORT procedure
formatting SQL output 165
reports
creating summary reports 194
from DICTONARY tables 282
RESET statement 252
resetting SQL procedure options 141
resetting options 141
retrieving rows 44
based on comparison 45
identifying columns with missing values 50
rows that satisfy a condition 44
satisfying multiple conditions 47
with a WHERE clause with missing values 54
with BETWEEN-AND operators 51
with IN operator 50
with IS MISSING operator 50
with LIKE operator 52
with other conditional operators 49
with simple WHERE clause 44
with truncated string comparison operators 53
return codes
  Pass-Through Facility 172
returning a SUBSTRING 359
right outer joins 85
ROLLBACK statement 390
rows 4
See also retrieving rows
adding to tables or views 250
combining data from multiple rows into a single row 59
counting 62, 362
counting duplicates 187
deleting 123, 245, 246
deleting duplicates 25
duplicates 143
including all 89
including all combinations of 87
inserting 116, 252
inserting with queries 118
inserting with SET clause 116
inserting with VALUES clause 117
joins and 328
matching 89
nonmatching 84, 85
ordering 265
producing from first or second query 108
restricting row processing 139
returned by subqueries 325
selecting 252, 316
selecting all 86
SQL procedure 216
updating all rows with same expression 120
updating rows with different expressions 121

S
SAS data files
  See tables
SAS data views
  DICTIONARY tables 149
  SQL procedure 217
SAS System information 149
SAS_PUT() function
  mapping PUT function to 378
SAS/ACCESS
  accessing a DBMS 167
  LIBNAME statement 168
SAS/ACCESS views
  SQL procedure 217
  updating 173
Sashelp views 149
  retrieving information about 151
searching for patterns 310, 341, 342
SELECT * statement
  expanding with FEEDBACK option 140
SELECT clause 20, 253
SELECT statement 20, 252
  See also WHERE clause
FROM clause 20
GROUP BY clause 21
HAVING clause 21
ORDER BY clause 21
ordering clauses 22
SELECT clause 20
self-joins 83
SET clause
  inserting rows with 116
  set membership 325
  set operators 344, 389
  ALL keyword 143
  combining queries with 103
  simple indexes 238
single-value subqueries 96
sort order 37, 39
  customized 197
sorting data 37
  by calculated column 40
  by column 38
  by column position 41
  by multiple columns 39
  by unselected columns 42
  columns with missing values 43
  grouping and sorting 66
  sorting data retrieved by views 244
  sorting sequence 43
SORTMSG option 229
SORTSEQ= option 229
SOUNDS-LIKE operator 299
source data
  generating SQL for in-database processing of 374
SQL 3
  generating for in-database processing of source data 374
SQL components 315
  BETWEEN condition 316
BTRIM function 317
CALCULATED 317
CASE expression 318
COALESCE function 320
column-definition 320
column-modifier 321
column-name 323
CONNECTION TO 324
CONTAINS condition 324
EXISTS condition 325
IN condition 325
IS condition 326
joined-table 327
LIKE condition 341
LOWER function 343
query expression 343
sql-expression 351
SUBSTRING function 359
summary-function 360
table-expression 368
UPPER function 369
SQL procedure
inner join 278
SQL procedure 3, 216
See also SQL components
ANSI standard and 387
coding conventions 217
collating sequence 388
column modifiers 388
combinations of column values 304
combining two tables 280
compared with DATA step 5
counting missing values with a macro 313
creating macro variables 157
creating queries 138
creating tables and inserting data 269
creating tables from query results 271
creating views from query results 290
cumulative time for 140
data set options with 156
data types and dates 320
debugging queries 138
DICTIONARY tables 149
table examples 7
formatting output 165
functions supported by 389
identifiers and naming conventions 390
indexes 237
joining three tables 293
joining two tables 276, 301
macro facility with 157
macro variables set by 162
matching case rows and control rows 310
missing values 313, 326
ODS with 174
orthogonal expressions 388
outer joins 284
PROC SQL tables 216
querying in-line views 297
reporting from DICTIONARY tables 282
reserved words 387
resetting options 141, 252
retrieving values 299
statistical functions 389
syntax 218
syntax checking 139
task tables 220, 221
terminology 4, 216
three-valued logic 390
timing individual statements 140
undo policy 384
updating PROC SQL tables 273
user privileges 390
using tables in other procedures 208
views 217
SQL Procedure Pass-Through Facility 171
connecting to a DBMS 171
example 172
return codes 172
sql-expression component 351
correlated subqueries and 357
functions and 352
operators and order of evaluation 352
query expressions and 354
subqueries and efficiency 358
truncated string comparison operators and 354
USER and 352
SQL, embedded 390
SQLCONSTDATETIME system option 373
SQLEXITCODE macro variable 163
SQLGENERATION= system option 374
SQLIPONEATTEMPT system option 377
SQLMAPPUTO= system option 378
SQLOBJS macro variable 163
SQLPROC macro variable 139, 163
SQLRC macro variable 163
SQLREDUCEPUT= system option 144, 379
SQLREDUCEPUTOBS= system option 144, 380
SQLREDUCEPUTVALUES= system option 145, 381
SQLREMERGE system option 383
SQLUNDOPOLICY 384
SQLUNDOPOLICY= system option 163, 384
### Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLXMSG macro variable</td>
<td>164</td>
</tr>
<tr>
<td>SQLXOPENERRS macro variable</td>
<td>163</td>
</tr>
<tr>
<td>SQLXRC macro variable</td>
<td>164</td>
</tr>
<tr>
<td>statistical functions</td>
<td>389</td>
</tr>
<tr>
<td>statistical summaries</td>
<td>56, 360</td>
</tr>
<tr>
<td>statistics</td>
<td></td>
</tr>
<tr>
<td>based on number of arguments</td>
<td>362</td>
</tr>
<tr>
<td>STIMER option</td>
<td>230</td>
</tr>
<tr>
<td>timing SQL procedure</td>
<td>140</td>
</tr>
<tr>
<td>STOPONTRUNC option</td>
<td>230</td>
</tr>
<tr>
<td>string comparison operators</td>
<td></td>
</tr>
<tr>
<td>truncated</td>
<td>354</td>
</tr>
<tr>
<td>Structured Query Language</td>
<td></td>
</tr>
<tr>
<td>See SQL</td>
<td></td>
</tr>
<tr>
<td>subqueries</td>
<td>96, 354</td>
</tr>
<tr>
<td>combining with joins</td>
<td>101</td>
</tr>
<tr>
<td>compared with joins</td>
<td>341</td>
</tr>
<tr>
<td>comparing with joins</td>
<td>143</td>
</tr>
<tr>
<td>correlated</td>
<td>357</td>
</tr>
<tr>
<td>correlated subqueries</td>
<td>98</td>
</tr>
<tr>
<td>efficiency and</td>
<td>358</td>
</tr>
<tr>
<td>multiple nesting levels</td>
<td>100</td>
</tr>
<tr>
<td>multiple-value</td>
<td>97</td>
</tr>
<tr>
<td>returning rows</td>
<td>325</td>
</tr>
<tr>
<td>single-value</td>
<td>96</td>
</tr>
<tr>
<td>testing for existence of a group of values</td>
<td>99</td>
</tr>
<tr>
<td>when to use</td>
<td>102</td>
</tr>
<tr>
<td>subsetting data</td>
<td>262, 365</td>
</tr>
<tr>
<td>SUBSTRING function</td>
<td>359</td>
</tr>
<tr>
<td>subtotals</td>
<td></td>
</tr>
<tr>
<td>computing percentages within</td>
<td>185</td>
</tr>
<tr>
<td>SUM function</td>
<td></td>
</tr>
<tr>
<td>summarizing data</td>
<td>58, 361</td>
</tr>
<tr>
<td>summarizing data</td>
<td>56</td>
</tr>
<tr>
<td>aggregate functions</td>
<td>56, 361</td>
</tr>
<tr>
<td>aggregate functions with unique values</td>
<td>61, 361</td>
</tr>
<tr>
<td>aggregate functions, using</td>
<td>56, 361</td>
</tr>
<tr>
<td>combining data from multiple rows into</td>
<td>59, 361</td>
</tr>
<tr>
<td>single row</td>
<td>59</td>
</tr>
<tr>
<td>displaying sums</td>
<td>58</td>
</tr>
<tr>
<td>in multiple columns</td>
<td>192</td>
</tr>
<tr>
<td>remerging summary statistics</td>
<td>59, 361</td>
</tr>
<tr>
<td>SQL procedure</td>
<td>361</td>
</tr>
<tr>
<td>with missing values</td>
<td>62</td>
</tr>
<tr>
<td>with WHERE clause</td>
<td>57</td>
</tr>
<tr>
<td>summary functions</td>
<td></td>
</tr>
<tr>
<td>See also aggregate functions</td>
<td></td>
</tr>
<tr>
<td>disabling remerging of data</td>
<td>146</td>
</tr>
<tr>
<td>summary reports</td>
<td></td>
</tr>
<tr>
<td>creating</td>
<td>194</td>
</tr>
<tr>
<td>summary statistics</td>
<td></td>
</tr>
<tr>
<td>remerging</td>
<td>59</td>
</tr>
<tr>
<td>summary-function component</td>
<td>360, 361</td>
</tr>
<tr>
<td>counting rows</td>
<td>362</td>
</tr>
<tr>
<td>remerging data</td>
<td>363</td>
</tr>
<tr>
<td>statistics based on number of arguments</td>
<td>362</td>
</tr>
<tr>
<td>summarizing data</td>
<td>361</td>
</tr>
<tr>
<td>sums</td>
<td></td>
</tr>
<tr>
<td>displaying</td>
<td>58</td>
</tr>
<tr>
<td>syntax checking</td>
<td>139</td>
</tr>
<tr>
<td>SYS_SQLSETLIMIT macro variable</td>
<td>385</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td></td>
</tr>
<tr>
<td>table aliases</td>
<td>262, 329</td>
</tr>
<tr>
<td>abbreviating column names</td>
<td>76</td>
</tr>
<tr>
<td>inner joins</td>
<td>76</td>
</tr>
<tr>
<td>table definitions</td>
<td>247</td>
</tr>
<tr>
<td>table expressions</td>
<td>344</td>
</tr>
<tr>
<td>table-expression component</td>
<td>368</td>
</tr>
<tr>
<td>tables</td>
<td>4</td>
</tr>
<tr>
<td>See also PROC SQL tables</td>
<td></td>
</tr>
<tr>
<td>Cartesian product</td>
<td>75</td>
</tr>
<tr>
<td>comparing</td>
<td>180</td>
</tr>
<tr>
<td>copying</td>
<td>115</td>
</tr>
<tr>
<td>counting duplicate rows</td>
<td>187</td>
</tr>
<tr>
<td>creating</td>
<td>112</td>
</tr>
<tr>
<td>creating from column definitions</td>
<td>112</td>
</tr>
<tr>
<td>creating from query results</td>
<td>113</td>
</tr>
<tr>
<td>creating with data set options</td>
<td>115</td>
</tr>
<tr>
<td>creating with macros</td>
<td>161</td>
</tr>
<tr>
<td>creating without rows</td>
<td>112</td>
</tr>
<tr>
<td>creating, like an existing table</td>
<td>115</td>
</tr>
<tr>
<td>DBMS tables</td>
<td>4</td>
</tr>
<tr>
<td>deleting</td>
<td>128</td>
</tr>
<tr>
<td>example tables</td>
<td>7</td>
</tr>
<tr>
<td>expanding hierarchical data</td>
<td>189</td>
</tr>
<tr>
<td>inserting rows</td>
<td>116</td>
</tr>
<tr>
<td>integrity constraints</td>
<td>129</td>
</tr>
<tr>
<td>joining a table to itself</td>
<td>83</td>
</tr>
<tr>
<td>omitting ORDER BY clause when creating</td>
<td>143</td>
</tr>
<tr>
<td>selecting all columns</td>
<td>22</td>
</tr>
<tr>
<td>selecting columns</td>
<td>22</td>
</tr>
<tr>
<td>selecting specific columns</td>
<td>23</td>
</tr>
<tr>
<td>SQL tables in other procedures</td>
<td>208</td>
</tr>
<tr>
<td>SQL tables in SAS</td>
<td>128</td>
</tr>
<tr>
<td>structure of</td>
<td>27</td>
</tr>
<tr>
<td>temporary tables versus in-line views</td>
<td>143</td>
</tr>
<tr>
<td>update errors</td>
<td>122</td>
</tr>
<tr>
<td>updating all rows with same expression</td>
<td>120</td>
</tr>
<tr>
<td>updating conditionally</td>
<td>200</td>
</tr>
<tr>
<td>updating rows with different expressions</td>
<td>121</td>
</tr>
<tr>
<td>updating values</td>
<td>120</td>
</tr>
<tr>
<td>updating with values from another table</td>
<td>203</td>
</tr>
</tbody>
</table>
temporary tables
  in-line views versus 143
terminology 4
text
  adding to output 27
THREADS option 230
three-valued logic 390
TIME function
  replacing references to 146
timing procedure statements 140
TODAY function
  replacing references to 146
truncated string comparison operators 53, 354

U
UBUFSIZE= option 231
undo policy
  SQL procedure 384
UNDO_POLICY= option 163, 230
union joins 89, 335
UNION operator 348
  combining queries 103, 104
UNIQUE keyword 127, 237
unique values
  aggregate functions with 61
counting 61
counting all rows 62
counting nonmissing values 62
  in columns 25
UPDATE statement 267
updating tables
  conditionally 200
  errors 122
  updating values 120
  with values from another table 203
updating views 132, 173
UPPER function 369
USER literal 352
user-defined macro variables 157

V
VALIDATE statement 268

W
WARNRECURS option 231
weighted averages 178
WHERE clause 21, 262
  filtering grouped data, versus HAVING clause 70
MEAN function with 57
missing values with 54
retrieving rows conditionally 44
summarizing data 57
WHERE expressions
  joins with 143
  width
    changing column width 125
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.