SAS® Certification Prep Guide
Advanced Programming for SAS®9
## Contents

*About This Book and CD*  
- What’s New  
- Purpose  
- Audience  
- Prerequisites

*How to Use This Book and CD*  
- Syntax Conventions for This Book

*SAS Certification Practice Exam: Advanced Programming for SAS*

*SAS Advanced Programming Exam for SAS*

*Additional Resources*

---

### Part 1 SQL Processing With SAS

**Chapter 1 △ Performing Queries Using PROC SQL**  
- Overview  
- PROC SQL Basics  
- Writing a PROC SQL Step  
- Selecting Columns  
- Specifying the Table  
- Specifying Subsetting Criteria  
- Ordering Rows  
- Querying Multiple Tables  
- Summarizing Groups of Data  
- Creating Output Tables  
- Additional Features  
- Summary  
- Quiz

**Chapter 2 △ Performing Advanced Queries Using PROC SQL**  
- Overview  
- Viewing SELECT Statement Syntax  
- Displaying All Columns  
- Limiting the Number of Rows Displayed  
- Eliminating Duplicate Rows from Output  
- Subsetting Rows by Using Conditional Operators  
- Subsetting Rows by Using Calculated Values  
- Enhancing Query Output  
- Summarizing and Grouping Data  
- Subsetting Data by Using Subqueries  
- Subsetting Data by Using Noncorrelated Subqueries  
- Subsetting Data by Using Correlated Subqueries
### Chapter 10 △ Processing Macro Variables at Execution Time

**Overview**  
326

- Creating a Macro Variable During DATA Step Execution  
327
- Creating Multiple Macro Variables During DATA Step Execution  
341
- Referencing Macro Variables Indirectly  
344
- Obtaining Macro Variable Values During DATA Step Execution  
350
- Creating Macro Variables During PROC SQL Step Execution  
352
- Working with PROC SQL Views  
359
- Using Macro Variables in SCL Programs  
360

**Summary**  
363

**Quiz**  
366

---

### Chapter 11 △ Creating and Using Macro Programs

**Overview**  
373

- Basic Concepts  
373
- Developing and Debugging Macros  
379
- Using Macro Parameters  
382
- Understanding Symbol Tables  
388
- Processing Statements Conditionally  
397
- Processing Statements Iteratively  
406
- Using Arithmetic and Logical Expressions  
411

**Summary**  
414

**Quiz**  
418

---

### Chapter 12 △ Storing Macro Programs

**Overview**  
424

- Understanding Session-Compiled Macros  
424
- Storing Macro Definitions in External Files  
425
- Storing Macro Definitions in Catalog SOURCE Entries  
427
- Using the Autocall Facility  
431
- Using Stored Compiled Macros  
435

**Summary**  
442

**Quiz**  
444

---

### Part 3 Advanced SAS Programming Techniques

---

### Chapter 13 △ Creating Samples and Indexes

**Overview**  
452

- Creating a Systematic Sample from a Known Number of Observations  
453
- Creating a Systematic Sample from an Unknown Number of Observations  
455
- Creating a Random Sample with Replacement  
456
- Creating a Random Sample without Replacement  
459
- Using Indexes  
460
About This Book and CD

What’s New

For future updates, go to http://supportprod.unx.sas.com/publishing/bbu/companion_site/61642.html and select the Updates link.

Purpose

The SAS Certification Prep Guide: Advanced Programming for SAS 9 prepares you to take the SAS Advanced Programming exam for SAS 9. New and experienced SAS users who want to prepare for this exam will find this guide to be an invaluable, convenient, and comprehensive resource that covers all of the topics tested on the exam.

Major topics include SQL processing with SAS, the SAS macro language, advanced SAS programming techniques, and optimizing SAS programs. You will also become familiar with the enhancements and new functionality that are available in SAS 9.

The book includes quizzes that enable you to test your understanding of material in each chapter. Additionally, solutions to all quizzes are included at the back of the book.

Audience


Prerequisites


If you want to test yourself to see if you have the necessary prerequisite base programming knowledge, go to support.sas.com/certify, where you will find information about certification credentials, exam preparation, and more!
How to Use This Book and CD

The SAS Certification Prep Guide: Advanced Programming for SAS 9 includes a companion CD, which can be found in the envelope inside the back cover of this book. The CD will enable you to practice your new skills.

Syntax Conventions for This Book

This is an example of how the general form of SAS code is shown in the book:

<table>
<thead>
<tr>
<th>General form, basic PROC SQL step to perform a query:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PROC SQL;</code></td>
</tr>
<tr>
<td><code>SELECT column-1,...,column-n&gt;</code></td>
</tr>
<tr>
<td>`FROM table-1</td>
</tr>
<tr>
<td><code>WHERE expression&gt;</code></td>
</tr>
<tr>
<td><code>GROUP BY column-1,...,column-n&gt;</code></td>
</tr>
<tr>
<td><code>ORDER BY column-1,...,column-n&gt;</code>;</td>
</tr>
<tr>
<td>where</td>
</tr>
<tr>
<td><code>PROC SQL</code></td>
</tr>
<tr>
<td>invokes the SQL procedure</td>
</tr>
<tr>
<td><code>SELECT</code></td>
</tr>
<tr>
<td>specifies the column(s) to be selected</td>
</tr>
<tr>
<td><code>FROM</code></td>
</tr>
<tr>
<td>specifies the table(s) to be queried</td>
</tr>
<tr>
<td><code>WHERE</code></td>
</tr>
<tr>
<td>subsets the data based on a condition</td>
</tr>
<tr>
<td><code>GROUP BY</code></td>
</tr>
<tr>
<td>classifies the data into groups based on the specified column(s)</td>
</tr>
<tr>
<td><code>ORDER BY</code></td>
</tr>
<tr>
<td>sorts the rows that the query returns by the value(s) of the specified column(s).</td>
</tr>
</tbody>
</table>

For example, in the general form above,

`SELECT, FROM, WHERE, GROUP BY, and ORDER BY` are in uppercase because they must be spelled as shown

`column-1, table-1, view-1, and expression` are in italics because each represents a value that you supply

`<...column-n>` is enclosed in angle brackets because it is optional syntax

`table-1` and `view-1` are separated by a vertical bar (`|`) to indicate that they are mutually exclusive.

The general forms of SAS statements and commands that are shown in this book include only the syntax that you need to know to prepare for the certification exam. For complete syntax, see the appropriate SAS reference guide.
SAS Certification Practice Exam: Advanced Programming for SAS 9

The SAS Certification Practice Exam: Advanced Programming for SAS 9 was designed to help you prepare for the SAS Advanced Programming exam for SAS 9. This practice exam was constructed to test the same knowledge and skills as the official certification exam. You can access this exam under the SAS Certification category at support.sas.com/selfpaced. (There is an additional fee charged for this practice exam.)

For information about how to register for the official SAS Advanced Programming exam for SAS 9, see the SAS Global Certification Program Web site at support.sas.com/certify/.

Additional Resources

Other resources might be helpful when learning SAS programming. You can refer to them as needed to enhance your understanding of the material covered in this book. You can access SAS help, documentation, and other resources from your SAS software or on the Web.

<table>
<thead>
<tr>
<th>From SAS Software</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Help</td>
<td>SAS 9: Select Help ➤ Help and Documentation.</td>
</tr>
<tr>
<td>Documentation</td>
<td>SAS 9: Select Help ➤ SAS Help and Documentation.</td>
</tr>
<tr>
<td></td>
<td>SAS Enterprise Guide: Access online documentation on the Web (see On the Web below).</td>
</tr>
</tbody>
</table>
## On the Web

<table>
<thead>
<tr>
<th>Website</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>support.sas.com/resources</td>
<td>System Requirements</td>
</tr>
<tr>
<td></td>
<td>Install Center</td>
</tr>
<tr>
<td></td>
<td>Product Documentation</td>
</tr>
<tr>
<td></td>
<td>Papers</td>
</tr>
<tr>
<td></td>
<td>Samples &amp; SAS Notes</td>
</tr>
<tr>
<td></td>
<td>Focus Areas</td>
</tr>
<tr>
<td>support.sas.com/learn</td>
<td>Bookstore</td>
</tr>
<tr>
<td></td>
<td>Training</td>
</tr>
<tr>
<td></td>
<td>Certification</td>
</tr>
<tr>
<td></td>
<td>SAS Learning Edition</td>
</tr>
<tr>
<td></td>
<td>Higher Education Resources</td>
</tr>
<tr>
<td></td>
<td>SAS OnDemand for Academics</td>
</tr>
<tr>
<td>support.sas.com/community</td>
<td>Users Groups</td>
</tr>
<tr>
<td></td>
<td>Events</td>
</tr>
<tr>
<td></td>
<td>E-newsletters</td>
</tr>
<tr>
<td></td>
<td>RSS &amp; Blogs</td>
</tr>
<tr>
<td></td>
<td>Discussion Forums</td>
</tr>
</tbody>
</table>
PART 1

SQL Processing With SAS

Chapter 1..............Performing Queries Using PROC SQL  3
Chapter 2..............Performing Advanced Queries Using PROC SQL  25
Chapter 3..............Combining Tables Horizontally Using PROC SQL  79
Chapter 4..............Combining Tables Vertically Using PROC SQL  125
Chapter 5..............Creating and Managing Tables Using PROC SQL  159
Chapter 6..............Creating and Managing Indexes Using PROC SQL  221
Chapter 7..............Creating and Managing Views Using PROC SQL  243
Chapter 8..............Managing Processing Using PROC SQL  261
CHAPTER 1
Performing Queries Using PROC SQL

Overview 4
   Introduction 4
   Objectives 4
PROC SQL Basics 4
   How PROC SQL Is Unique 5
Writing a PROC SQL Step 6
   The SELECT Statement 7
Selecting Columns 8
   Creating New Columns 9
Specifying the Table 10
Specifying Subsetting Criteria 10
Ordering Rows 10
   Ordering by Multiple Columns 12
Querying Multiple Tables 12
   Specifying Columns That Appear in Multiple Tables 13
   Specifying Multiple Table Names 14
   Subsetting Rows 14
   Ordering Rows 15
Summarizing Groups of Data 16
   Example 16
   Summary Functions 16
Creating Output Tables 17
   Example 17
Additional Features 18
Summary 19
Text Summary 19
   PROC SQL Basics 19
   Writing a PROC SQL Step 19
   Selecting Columns 19
   Specifying the Table 19
   Specifying Subsetting Criteria 19
   Ordering Rows 19
   Querying Multiple Tables 20
   Summarizing Groups of Data 20
   Creating Output Tables 20
   Additional Features 20
Syntax 20
Sample Programs 20
   Querying a Table 20
   Summarizing Groups of Data 21
   Creating a Table from the Results of a Query on Two Tables 21
Overview

Introduction

Sometimes you need quick answers to questions about your data. You might want to query (retrieve data from) a single SAS data set or a combination of data sets to

- examine relationships between data values
- view a subset of your data
- compute values quickly.

The SQL procedure (PROC SQL) provides an easy, flexible way to query and combine your data. This chapter shows you how to create a basic query using one or more tables (data sets). You’ll also learn how to create a new table from your query.

Objectives

In this chapter, you learn to

- invoke the SQL procedure
- select columns
- define new columns
- specify the table(s) to be read
- specify subsetting criteria
- order rows by values of one or more columns
- group results by values of one or more columns
- end the SQL procedure
- summarize data
- generate a report as the output of a query
- create a table as the output of a query.

PROC SQL Basics

PROC SQL is the SAS implementation of Structured Query Language (SQL), which is a standardized language that is widely used to retrieve and update data in tables and in views that are based on those tables.
The following chart shows terms used in data processing, SAS, and SQL that are synonymous. The SQL terms are used in this chapter.

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

PROC SQL can often be used as an alternative to other SAS procedures or the DATA step. You can use PROC SQL to

- retrieve data from and manipulate SAS tables
- add or modify data values in a table
- add, modify, or drop columns in a table
- create tables and views
- join multiple tables (whether or not they contain columns with the same name)
- generate reports.

Like other SAS procedures, PROC SQL also enables you to combine data from two or more different types of data sources and present them as a single table. For example, you can combine data from two different types of external databases, or you can combine data from an external database and a SAS data set.

How PROC SQL Is Unique

PROC SQL differs from most other SAS procedures in several ways:

- Unlike other PROC statements, many statements in PROC SQL are composed of clauses. For example, the following PROC SQL step contains two statements: the PROC SQL statement and the SELECT statement. The SELECT statement contains several clauses: SELECT, FROM, WHERE, and ORDER BY.

```sql
proc sql;
  select empid, jobcode, salary,
       salary*.06 as bonus
  from sasuser.payrollmaster
  where salary<32000
  order by jobcode;
```

- The PROC SQL step does not require a RUN statement. PROC SQL executes each query automatically. If you use a RUN statement with a PROC SQL step, SAS
ignores the RUN statement, executes the statements as usual, and generates the note shown below in the SAS log.

Table 1.1  SAS Log

```
1884 proc sql;
1885   select empid,jobcode,salary,
1886         salary*.06 as bonus
1887   from sasuser.payrollmaster
1888   where salary<32000
1889   order by jobcode;
1890 run;
NOTE: PROC SQL statements are executed immediately;
      The RUN statement has no effect.
```

Unlike many other SAS procedures, PROC SQL continues to run after you submit a step. To end the procedure, you must submit another PROC step, a DATA step, or a QUIT statement, as shown:

```
proc sql;
   select empid,jobcode,salary,
       salary*.06 as bonus
   from sasuser.payrollmaster
   where salary<32000
   order by jobcode;
quit;
```

When you submit a PROC SQL step without ending it, the status line displays the message PROC SQL running.

Note: As a precaution, SAS Enterprise Guide automatically adds a QUIT statement to your code when you submit it to SAS. However, you should get in the habit of adding the QUIT statement to your code.

---

**Writing a PROC SQL Step**

Before creating a query, you must first reference the library in which your table is stored. Then you write a PROC SQL step to query your table.
General form, basic PROC SQL step to perform a query:

```
PROC SQL;
  SELECT column-1,...column-n
  FROM table-1|view-1,...table-n|view-n
  <WHERE expression>
  <GROUP BY column-1,... column-n>
  <ORDER BY column-1,... column-n>;
```

where

- **PROC SQL**
  - invokes the SQL procedure
- **SELECT**
  - specifies the column(s) to be selected
- **FROM**
  - specifies the table(s) to be queried
- **WHERE**
  - subsets the data based on a condition
- **GROUP BY**
  - classifies the data into groups based on the specified column(s)
- **ORDER BY**
  - sorts the rows that the query returns by the value(s) of the specified column(s).

**CAUTION:**

Unlike other SAS procedures the order of clauses with a SELECT statement in PROC SQL is important. Clauses must appear in the order shown above.

**Note:** A query can also include a HAVING clause, which is introduced at the end of this chapter. To learn more about the HAVING clause, see Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25.

---

**The SELECT Statement**

The SELECT statement, which follows the PROC SQL statement, retrieves and displays data. It is composed of **clauses**, each of which begins with a keyword and is followed by one or more components. The SELECT statement in the following sample code contains four clauses: the required clauses SELECT and FROM, and the optional clauses WHERE and ORDER BY. The end of the statement is indicated by a semicolon.

```
proc sql;
  |select empid,jobcode,salary,
  |   salary*.06 as bonus
  |----from sasuser.payrollmaster
  |----where salary<32000
  |----order by jobcode;
```

**Note:** A PROC SQL step that contains one or more SELECT statements is referred to as a PROC SQL query. The SELECT statement is only one of several statements that can be used with PROC SQL.

The following PROC SQL query creates the output report that is shown below:
A PROC SQL query produces a result set that can be output as a report, a table, or a PROC SQL view.

<table>
<thead>
<tr>
<th>Type of Output</th>
<th>PROC SQL Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>report</td>
<td>SELECT</td>
</tr>
<tr>
<td>table</td>
<td>CREATE TABLE and SELECT</td>
</tr>
<tr>
<td>PROC SQL view</td>
<td>CREATE VIEW and SELECT</td>
</tr>
</tbody>
</table>

Note: The CREATE TABLE statement is introduced later in this chapter. You can learn about creating tables in Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159. You can learn more about PROC SQL views in Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243.

You will learn more about the SELECT statement in the following sections.

Selecting Columns

To specify which column(s) to display in a query, you write a SELECT clause, the first clause in the SELECT statement. After the keyword SELECT, list one or more column names and separate the column names with commas. In the SELECT clause, you can both specify existing columns (columns that are already stored in a table) and create new columns.

The following SELECT clause specifies the columns EmpID, JobCode, Salary, and bonus. The columns EmpID, JobCode, and Salary are existing columns. The column named bonus is a new column.

```sql
proc sql;
    select empid, jobcode, salary,
            salary*.06 as bonus
    from sasuser.payrollmaster
    where salary<32000
    order by jobcode;
```
Creating New Columns

You can create new columns that contain either text or a calculation. New columns will appear in output, along with any existing columns that are selected. Keep in mind that new columns exist only for the duration of the query, unless a table or a view is created.

To create a new column, include any valid SAS expression in the SELECT clause list of columns. You can optionally assign a column alias, a name, to a new column by using the keyword AS followed by the name that you would like to use.

Note: A column alias must follow the rules for SAS names.

In the sample PROC SQL query, shown below, an expression is used to calculate the new column: the values of Salary are multiplied by .06. The keyword AS is used to assign the column alias bonus to the new column.

```sql
proc sql;
  select empid, jobcode, salary,
       salary*.06 as bonus
  from sasuser.payrollmaster
  where salary<32000
  order by jobcode;
```

A column alias is useful because it allows you to reference the column elsewhere in the query.

Note: You can learn more about referencing a calculated column from other clauses in Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25.

Also, the column alias will appear as a column heading in the output. The following output shows how the calculated column bonus is displayed. Notice that the column alias bonus appears in lowercase, exactly as it is specified in the SELECT clause.

```
<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
<th>bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>FA1</td>
<td>$31,561</td>
<td>1899.66</td>
</tr>
<tr>
<td>1422</td>
<td>FA1</td>
<td>$31,436</td>
<td>1896.10</td>
</tr>
<tr>
<td>1113</td>
<td>FA1</td>
<td>$31,314</td>
<td>1878.84</td>
</tr>
<tr>
<td>1132</td>
<td>FA1</td>
<td>$31,378</td>
<td>1882.68</td>
</tr>
<tr>
<td>1094</td>
<td>FA1</td>
<td>$31,175</td>
<td>1870.5</td>
</tr>
<tr>
<td>1789</td>
<td>SCP</td>
<td>$25,656</td>
<td>1539.36</td>
</tr>
<tr>
<td>1564</td>
<td>SCP</td>
<td>$26,398</td>
<td>1581.96</td>
</tr>
<tr>
<td>1354</td>
<td>SCP</td>
<td>$25,689</td>
<td>1540.14</td>
</tr>
<tr>
<td>1101</td>
<td>SCP</td>
<td>$26,212</td>
<td>1572.72</td>
</tr>
<tr>
<td>1858</td>
<td>SCP</td>
<td>$25,120</td>
<td>1507.2</td>
</tr>
<tr>
<td>1405</td>
<td>SCP</td>
<td>$25,278</td>
<td>1518.88</td>
</tr>
<tr>
<td>1104</td>
<td>SCP</td>
<td>$25,124</td>
<td>1507.44</td>
</tr>
</tbody>
</table>
```

In the SELECT clause, you can optionally specify a label for an existing or a new column. If both a label and a column alias are specified for a new column, the label will be displayed as the column heading in the output. If only a column alias is specified, it is important that you specify the column alias exactly as you want it to appear in the output.
Note: You can learn about creating new columns that contain text and about specifying labels for columns in Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25.

Specifying the Table

After writing the SELECT clause, you specify the table to be queried in the FROM clause. Type the keyword FROM, followed by the name of the table, as shown:

```sql
proc sql;
select empid,jobcode,salary,
    salary*.06 as bonus
from sasuser.payrollmaster
where salary<32000
order by jobcode;
```

The PROC SQL step above queries the permanent SAS table Payrollmaster, which is stored in a SAS library to which the libref Sasuser has been assigned.

Specifying Subsetting Criteria

To subset data based on a condition, use a WHERE clause in the SELECT statement. As in the WHERE statement and the WHERE command used in other SAS procedures, the expression in the WHERE clause can be any valid SAS expression. In the WHERE clause, you can specify any column(s) from the underlying table(s). The columns specified in the WHERE clause do not have to be specified in the SELECT clause.

In the following PROC SQL query, the WHERE clause selects rows in which the value of the column Salary is less than 32,000. The output is also shown.

```sql
proc sql;
select empid,jobcode,salary,
    salary*.06 as bonus
from sasuser.payrollmaster
where salary<32000
order by jobcode;
```

Ordering Rows

The order of rows in the output of a PROC SQL query cannot be guaranteed, unless you specify a sort order. To sort rows by the values of specific columns, you can use the
ORDER BY clause in the SELECT statement. Specify the keywords ORDER BY, followed by one or more column names separated by commas.

In the following PROC SQL query, the ORDER BY clause sorts rows by values of the column **JobCode**:

```sql
proc sql;
  select empid, jobcode, salary,
       salary*.06 as bonus
  from sasuser.payrollmaster
  where salary<32000
  order by jobcode;
```

**Note:** In this example, the ORDER BY clause is the last clause in the SELECT statement, so the ORDER BY clause ends with a semicolon.

In the output of the sample query, shown below, the rows are sorted by the values of **JobCode**. By default, the ORDER BY clause sorts rows in *ascending* order.

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
<th>bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>FA1</td>
<td>$31,661</td>
<td>1898.66</td>
</tr>
<tr>
<td>1422</td>
<td>FA1</td>
<td>$31,436</td>
<td>1886.16</td>
</tr>
<tr>
<td>1113</td>
<td>FA1</td>
<td>$31,314</td>
<td>1878.84</td>
</tr>
<tr>
<td>1132</td>
<td>FA1</td>
<td>$31,370</td>
<td>1882.69</td>
</tr>
<tr>
<td>1094</td>
<td>FA1</td>
<td>$31,175</td>
<td>1870.5</td>
</tr>
<tr>
<td>1789</td>
<td>SCP</td>
<td>$25,656</td>
<td>1539.35</td>
</tr>
<tr>
<td>1564</td>
<td>SCP</td>
<td>$25,366</td>
<td>1581.96</td>
</tr>
<tr>
<td>1354</td>
<td>SCP</td>
<td>$25,669</td>
<td>1540.14</td>
</tr>
<tr>
<td>1101</td>
<td>SCP</td>
<td>$25,212</td>
<td>1572.72</td>
</tr>
<tr>
<td>1668</td>
<td>SCP</td>
<td>$25,120</td>
<td>1507.2</td>
</tr>
<tr>
<td>1405</td>
<td>SCP</td>
<td>$25,278</td>
<td>1516.83</td>
</tr>
<tr>
<td>1104</td>
<td>SCP</td>
<td>$25,124</td>
<td>1507.44</td>
</tr>
</tbody>
</table>

To sort rows in *descending* order, specify the keyword DESC following the column name. For example, the preceding ORDER BY clause could be modified as follows:

```sql
order by jobcode desc;
```

In the ORDER BY clause, you can alternatively reference a column by its *position* in the SELECT clause list rather than by name. Use an integer to indicate the column's position. The ORDER BY clause in the preceding PROC SQL query has been modified, below, to specify the column **JobCode** by its position in the SELECT clause list (2) rather than by name:

```sql
proc sql;
  select empid, jobcode, salary,
       salary*.06 as bonus
  from sasuser.payrollmaster
  where salary<32000
  order by 2;
```
Ordering by Multiple Columns

To sort rows by the values of two or more columns, list multiple column names (or numbers) in the ORDER BY clause, and use commas to separate the column names (or numbers). In the following PROC SQL query, the ORDER BY clause sorts by the values of two columns, JobCode and EmpID:

```sql
proc sql;
  select empid,jobcode,salary,
       salary*.06 as bonus
  from sasuser.payrollmaster
  where salary<32000
  order by jobcode,empid;
```

The rows are sorted first by JobCode and then by EmpID, as shown in the following output.

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
<th>bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1364</td>
<td>FA1</td>
<td>$31,75</td>
<td>1870.5</td>
</tr>
<tr>
<td>1113</td>
<td>FA1</td>
<td>$31,314</td>
<td>1878.84</td>
</tr>
<tr>
<td>1132</td>
<td>FA1</td>
<td>$31,378</td>
<td>1882.68</td>
</tr>
<tr>
<td>1422</td>
<td>FA1</td>
<td>$31,436</td>
<td>1896.16</td>
</tr>
<tr>
<td>1970</td>
<td>FA1</td>
<td>$31,661</td>
<td>1909.66</td>
</tr>
<tr>
<td>1101</td>
<td>SCP</td>
<td>$25,212</td>
<td>1572.72</td>
</tr>
<tr>
<td>1104</td>
<td>SCP</td>
<td>$25,124</td>
<td>1507.44</td>
</tr>
<tr>
<td>1354</td>
<td>SCP</td>
<td>$25,659</td>
<td>1540.14</td>
</tr>
<tr>
<td>1405</td>
<td>SCP</td>
<td>$25,278</td>
<td>1516.68</td>
</tr>
<tr>
<td>1564</td>
<td>SCP</td>
<td>$26,366</td>
<td>1581.96</td>
</tr>
<tr>
<td>1658</td>
<td>SCP</td>
<td>$25,120</td>
<td>1507.2</td>
</tr>
<tr>
<td>1709</td>
<td>SCP</td>
<td>$25,656</td>
<td>1539.36</td>
</tr>
</tbody>
</table>

Note: You can mix the two types of column references, names and numbers, in the ORDER BY clause. For example, the preceding ORDER BY clause could be rewritten as follows:

```sql
order by 2,empid;
```

Querying Multiple Tables

This topic deals with the more complex task of extracting data from two or more tables.

Previously, you learned how to write a PROC SQL step to query a single table. Suppose you now want to examine data that is stored in two tables. PROC SQL allows you to combine tables horizontally, in other words, to combine rows of data.
In SQL terminology, combining tables horizontally is called \textit{joining} tables. Joins do not alter the original tables.

Suppose you want to create a report that displays the following information for employees of a company: employee identification number, last name, original salary, and new salary. There is no single table that contains all of these columns, so you will have to join the two tables \textit{Sasuser.Salcomps} and \textit{Sasuser.Newsals}. In your query, you want to select four columns, two from the first table and two from the second table. You also need to be sure that the rows you join belong to the same employee. To check this, you want to match employee identification numbers for rows that you merge and to select only the rows that match.

This type of join is known as an \textit{inner join}. An inner join returns a result set for all of the rows in a table that have one or more matching rows in another table.

\textit{Note}: For more information about PROC SQL joins, see Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79.

Now let's see how you write a PROC SQL step to combine tables. To join two tables for a query, you can use a PROC SQL step such as the one below. This step uses the \textit{SELECT} statement to join data from the tables \textit{Salcomps} and \textit{Newsals}. Both of these tables are stored in a SAS library to which the libref \textit{Sasuser} has been assigned.

\begin{verbatim}
proc sql;
  select salcomps.empid,lastname,
       newsals.salary,newsalary
  from sasuser.salcomps,sasuser.newsals
  where salcomps.empid=newsals.empid
  order by lastname;
\end{verbatim}

Let's take a closer look at each clause of this PROC SQL step.

---

\textbf{Specifying Columns That Appear in Multiple Tables}

When you join two or more tables, list the columns that you want to select from \textit{both} tables in the \textit{SELECT} clause. Separate all column names with commas.

If the tables that you are querying contain same-named columns and you want to list one of these columns in the \textit{SELECT} clause, you must specify a table name as a prefix for that column.

\textit{Note}: Prefixing a table name to a column name is called \textit{qualifying} the column name.

The following PROC SQL step joins the two tables \textit{Sasuser.Salcomps} and \textit{Sasuser.Newsals}, both of which contain columns named \textit{EmpID} and \textit{Salary}. To tell PROC SQL where to read the columns \textit{EmpID} and \textit{Salary}, the \textit{SELECT} clause specifies the table name \textit{Salcomps} as a prefix for \textit{Empid}, and \textit{Newsals} as a prefix for \textit{Salary}. 
Specifying Multiple Table Names

When you join multiple tables in a PROC SQL query, you specify each table name in the FROM clause, as shown below:

```sql
proc sql;
  select salcomps.empid,lastname,
         newsals.salary,newssalary
  from sasuser.salcomps,sasuser.newsals
  where salcomps.empid=newsals.empid
  order by lastname;
```

As in the SELECT clause, you separate names in the FROM clause (in this case, table names) with commas.

Subsetting Rows

As in a query on a single table, the WHERE clause in the SELECT statement selects rows from two or more tables, based on a condition. When you join multiple tables, be sure that the WHERE clause specifies columns with data whose values match, to avoid unwanted combinations.

In the following example, the WHERE clause selects only rows in which the value for EmpID in Sasuser.Salcomps matches the value for EmpID in Sasuser.Newsals. Qualified column names must be used in the WHERE clause to specify each of the two EmpID columns.

```sql
proc sql;
  select salcomps.empid,lastname,
         newsals.salary,newssalary
  from sasuser.salcomps,sasuser.newsals
  where salcomps.empid=newsals.empid
  order by lastname;
```

The output is shown, in part, below.
Performing Queries Using PROC SQL

Ordering Rows

Note: In the table Sasuser.Newals, the Salary column has the label Employee Salary, as shown in this output.

CAUTION: If you join tables that don’t contain one or more columns with matching data values, you can produce a huge amount of output.

Ordering Rows

As in PROC SQL steps that query just one table, the ORDER BY clause specifies which column(s) should be used to sort rows in the output. In the following query, the rows will be sorted by LastName:

```
proc sql;
  select salcomps.empid, lastname,
       newsals.salary, newsalary
  from sasuser.salcomps, sasuser.newsals
  where salcomps.empid = newsals.empid
  order by lastname;
```

<table>
<thead>
<tr>
<th>EmpID</th>
<th>LastName</th>
<th>Employee Salary</th>
<th>NewSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E00042</td>
<td>ANDERSON</td>
<td>$32,000</td>
<td>$36,023.96</td>
</tr>
<tr>
<td>E00006</td>
<td>ANDERSON</td>
<td>$31,000</td>
<td>$33,753.70</td>
</tr>
<tr>
<td>E00008</td>
<td>BADINE</td>
<td>$85,000</td>
<td>$93,811.78</td>
</tr>
<tr>
<td>E00021</td>
<td>BAKER JR.</td>
<td>$43,000</td>
<td>$43,386.40</td>
</tr>
<tr>
<td>E00002</td>
<td>BOWER</td>
<td>$27,000</td>
<td>$31,163.98</td>
</tr>
<tr>
<td>E00027</td>
<td>BOWMAN</td>
<td>$21,000</td>
<td>$35,579.22</td>
</tr>
<tr>
<td>E00030</td>
<td>BREWER</td>
<td>$30,000</td>
<td>$41,055.05</td>
</tr>
<tr>
<td>E00025</td>
<td>BROCKLEBANK</td>
<td>$23,000</td>
<td>$25,673.57</td>
</tr>
<tr>
<td>E00015</td>
<td>BROWN</td>
<td>$41,000</td>
<td>$45,394.20</td>
</tr>
<tr>
<td>E00041</td>
<td>BRUTON</td>
<td>$26,000</td>
<td>$53,399.68</td>
</tr>
<tr>
<td>E00049</td>
<td>CHASE JR.</td>
<td>$39,000</td>
<td>$32,892.87</td>
</tr>
<tr>
<td>E00024</td>
<td>COCKERHAM</td>
<td>$21,000</td>
<td>$21,213.98</td>
</tr>
<tr>
<td>E00032</td>
<td>COUCH</td>
<td>$24,000</td>
<td>$26,775.81</td>
</tr>
<tr>
<td>E00018</td>
<td>CROSS</td>
<td>$33,000</td>
<td>$35,947.60</td>
</tr>
<tr>
<td>E00013</td>
<td>DABAIJO</td>
<td>$22,000</td>
<td>$23,243.79</td>
</tr>
<tr>
<td>E00009</td>
<td>DEMENT</td>
<td>$34,000</td>
<td>$35,501.12</td>
</tr>
</tbody>
</table>
Summarizing Groups of Data

So far you’ve seen PROC SQL steps that create detail reports. But you might also want to summarize data in groups. To group data for summarizing, you can use the GROUP BY clause. The GROUP BY clause is used in queries that include one or more summary functions. Summary functions produce a statistical summary for each group that is defined in the GROUP BY clause.

Let’s look at the GROUP BY clause and summary functions more closely.

Example

Suppose you want to determine the total number of miles traveled by frequent-flyer program members in each of three membership classes (Gold, Silver, and Bronze). Frequent-flyer program information is stored in the table Sasuser.Frequentflyers. To summarize your data, you can submit the following PROC SQL step:

```sql
proc sql;
  select membertype
     sum(milestraveled) as TotalMiles
  from sasuser.frequentflyers
  group by membertype;
```

In this case, the SUM function totals the values of the MilesTraveled column to create the TotalMiles column. The GROUP BY clause groups the data by the values of MemberType.

As in the ORDER BY clause, in the GROUP BY clause you specify the keywords GROUP BY, followed by one or more column names separated by commas.

The results show total miles by membership class (MemberType).

<table>
<thead>
<tr>
<th>MemberType</th>
<th>TotalMiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRONZE</td>
<td>3229225</td>
</tr>
<tr>
<td>GOLD</td>
<td>2903569</td>
</tr>
<tr>
<td>SILVER</td>
<td>4345169</td>
</tr>
</tbody>
</table>

Note: If you specify a GROUP BY clause in a query that does not contain a summary function, your clause is changed to an ORDER BY clause, and a message to that effect is written to the SAS log.

Summary Functions

To summarize data, you can use the following summary functions with PROC SQL. Notice that some functions have more than one name to accommodate both SAS and SQL conventions. Where multiple names are listed, the first name is the SQL name.

- **AVG, MEAN**: 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
MIN smallest value
NMISS number of missing values
PRT probability of a greater absolute value of student’s t
RANGE range of values
STD standard deviation
STDERR standard error of the mean
SUM sum of values
T student’s t value for testing the hypothesis that the population mean is zero
USS uncorrected sum of squares
VAR variance

Creating Output Tables

To create a new table from the results of a query, use a CREATE TABLE statement that includes the keyword AS and the clauses that are used in a PROC SQL query: SELECT, FROM, and any optional clauses, such as ORDER BY. The CREATE TABLE statement stores your query results in a table instead of displaying the results as a report.

General form, basic PROC SQL step for creating a table from a query result:

```
PROC SQL;
CREATE TABLE table-name AS
SELECT column-1,...column-n
FROM table-1|view-1,...table-n|view-n
<WHERE expression>
<GROUP BY column-1,... column-n>>
<ORDER BY column-1,... column-n>>;
```

where

- `table-name` specifies the name of the table to be created.

Note: A query can also include a HAVING clause, which is introduced at the end of this chapter. To learn more about the HAVING clause, see Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25.

Example

Suppose that after determining the total miles traveled for each frequent-flyer membership class in the `Sasuser.Frequentfliers` table, you want to store this information in the temporary table `Work.Miles`. To do so, you can submit the following PROC SQL step:
proc sql;
create table work.miles as
   select membertype
       sum(milestraveled) as TotalMiles
   from sasuser.frequentflyers
   group by membertype;

Because the CREATE TABLE statement is used, this query does not create a report. The SAS log verifies that the table was created and indicates how many rows and columns the table contains.

Table 1.2 SAS Log

| NOTE: Table WORK.MILES created, with 3 rows and 2 columns. |

Tip: In this example, you are instructed to save the data to a temporary table that will be deleted at the end of the SAS session. To save the table permanently in the Sasuser library, use the libref Sasuser instead of the libref Work in the CREATE TABLE clause.

Additional Features

To further refine a PROC SQL query that contains a GROUP BY clause, you can use a HAVING clause. A HAVING clause works with the GROUP BY clause to restrict the groups that are displayed in the output, based on one or more specified conditions.

For example, the following PROC SQL query groups the output rows by JobCode. The HAVING clause uses the summary function AVG to specify that only the groups that have an average salary that is greater than 40,000 will be displayed in the output.

proc sql;
   select jobcode,avg(salary) as Avg
   from sasuser.payrollmaster
   group by jobcode
   having avg(salary)>40000
   order by jobcode;

Note: You can learn more about the use of the HAVING clause in Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25.
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

PROC SQL Basics

PROC SQL uses statements that are written in Structured Query Language (SQL), which is a standardized language that is widely used to retrieve and update data in tables and in views that are based on those tables. When you want to examine relationships between data values, subset your data, or compute values, the SQL procedure provides an easy, flexible way to analyze your data.

PROC SQL differs from most other SAS procedures in several ways:
- Many statements in PROC SQL, such as the SELECT statement, are composed of clauses.
- The PROC SQL step does not require a RUN statement.
- PROC SQL continues to run after you submit a step. To end the procedure, you must submit another PROC step, a DATA step, or a QUIT statement.

Writing a PROC SQL Step

Before creating a query, you must assign a libref to the SAS data library in which the table to be used is stored. Then you submit a PROC SQL step. You use the PROC SQL statement to invoke the SQL procedure.

Selecting Columns

To specify which column(s) to display in a query, you write a SELECT clause as the first clause in the SELECT statement. In the SELECT clause, you can specify existing columns and create new columns that contain either text or a calculation.

Specifying the Table

You specify the table to be queried in the FROM clause.

Specifying Subsetting Criteria

To subset data based on a condition, write a WHERE clause that contains an expression.

Ordering Rows

The order of rows in the output of a PROC SQL query cannot be guaranteed, unless you specify a sort order. To sort rows by the values of specific columns, use the ORDER BY clause.
Chapter 1

Querying Multiple Tables

You can use a PROC SQL step to query data that is stored in two or more tables. In SQL terminology, this is called joining tables. Follow these steps to join multiple tables:

1. Specify column names from one or both tables in the SELECT clause and, if you are selecting a column that has the same name in multiple tables, prefix the table name to that column name.
2. Specify each table name in the FROM clause.
3. Use the WHERE clause to select rows from two or more tables, based on a condition.
4. Use the ORDER BY clause to sort rows that are retrieved from two or more tables by the values of the selected column(s).

Summarizing Groups of Data

You can use a GROUP BY clause in your PROC SQL step to summarize data in groups. The GROUP BY clause is used in queries that include one or more summary functions. Summary functions produce a statistical summary for each group that is defined in the GROUP BY clause.

Creating Output Tables

To create a new table from the results of your query, you can use the CREATE TABLE statement in your PROC SQL step. This statement enables you to store your results in a table instead of displaying the query results as a report.

Additional Features

To further refine a PROC SQL query that contains a GROUP BY clause, you can use a HAVING clause. A HAVING clause works with the GROUP BY clause to restrict the groups that are displayed in the output, based on one or more specified conditions.

Syntax

LIBNAME libref 'SAS-data-library';
PROC SQL;
CREATE TABLE table-name AS
SELECT column-1<,...column-n>
FROM table-1|view-1<,...table-n|view-n>
<WHERE expression>
<GROUP BY column-1<,...column-n>>
<ORDER BY column-1<,...column-n>>;
QUIT;

Sample Programs

Querying a Table

proc sql;
select empid,jobcode,salary,
salary*.06 as bonus
Performing Queries Using PROC SQL

Quiz 21

from sasuser.payrollmaster
where salary<32000
order by jobcode;
quit;

Summarizing Groups of Data

proc sql;
select membertype,
    sum(milestraveled) as TotalMiles
from sasuser.frequentflyers
group by membertype;
quit;

Creating a Table from the Results of a Query on Two Tables

proc sql;
create table work.miles as
    select salcomps.empid, lastname,
        newsals.salary.newsalary
    from sasuser.salcomps.sasuser.newsals
    where salcomps.empid=newsals.empid
    order by 2;
quit;

Points to Remember

☐ Do not use a RUN statement with the SQL procedure.
☐ Do not end a clause with a semicolon unless it is the last clause in the statement.
☐ When you join multiple tables, be sure to specify columns that have matching data values in the WHERE clause in order to avoid unwanted combinations.
☐ To end the SQL procedure, you can submit another PROC step, a DATA step, or a QUIT statement.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Which of the clauses in the PROC SQL program below is written incorrectly?

    proc sql;
    select style sqfeet bedrooms
    from choice.houses
    where sqfeet ge 800;

   a SELECT
   b FROM
   c WHERE
   d both a and c

2 How many statements does the program below contain?

    proc sql;
    select grapes,oranges,
grapes + oranges as sumsales
from sales.produce
order by sumsales;

a two
b three
c four
d five

3 Complete the following PROC SQL query to select the columns **Address** and **SqFeet** from the table *List.Size* and to select **Price** from the table *List.Price*. (Only the **Address** column appears in both tables.)

```sql
proc sql;
select address,sqfeet,price
from list.size,list.price;
```

a select address,sqfeet,price
b select size.address,sqfeet,price
c select price.address,sqfeet,price
d either b or c

4 Which of the clauses below correctly sorts rows by the values of the columns **Price** and **SqFeet**?

a order price, sqfeet
b order by price,sqfeet
c sort by price sqfeet
d sort price sqfeet

5 Which clause below specifies that the two tables *Produce* and *Hardware* be queried? Both tables are located in a library to which the libref *Sales* has been assigned.

a select sales.produce sales.hardware
b from sales.produce sales.hardware
c from sales.produce,sales.hardware
d where sales.produce, sales.hardware

6 Complete the SELECT clause below to create a new column named **Profit** by subtracting the values of the column **Cost** from those of the column **Price**.

```sql
select fruit,cost,price,
```

a Profit=price-cost
b price-cost as Profit
c profit=price-cost
d Profit as price-cost
7. What happens if you use a GROUP BY clause in a PROC SQL step without a summary function?
   a. The step does not execute.
   b. The first numeric column is summed by default.
   c. The GROUP BY clause is changed to an ORDER BY clause.
   d. The step executes but does not group or sort data.

8. If you specify a CREATE TABLE statement in your PROC SQL step,
   a. the results of the query are displayed, and a new table is created.
   b. a new table is created, but it does not contain any summarization that was specified in the PROC SQL step.
   c. a new table is created, but no report is displayed.
   d. results are grouped by the value of the summarized column.

9. Which statement is true regarding the use of the PROC SQL step to query data that is stored in two or more tables?
   a. When you join multiple tables, the tables must contain a common column.
   b. You must specify the table from which you want each column to be read.
   c. The tables that are being joined must be from the same type of data source.
   d. If two tables that are being joined contain a same-named column, then you must specify the table from which you want the column to be read.

10. Which clause in the following program is incorrect?

    ```sql
    proc sql;
    select sex, mean(weight) as avgweight
    from company.employees company.health
    where employees.id=health.id
    group by sex;
    ```

    a. SELECT
    b. FROM
    c. WHERE
    d. GROUP BY
Overview 27
Introduction 27
Objectives 27
Prerequisites 28
Viewing SELECT Statement Syntax 28
Displaying All Columns 29
Using SELECT * 29
Using the FEEDBACK Option 29
Limiting the Number of Rows Displayed 30
Example 30
Eliminating Duplicate Rows from Output 31
Example 31
Subsetting Rows by Using Conditional Operators 32
Using Operators in PROC SQL 32
Using the BETWEEN-AND Operator to Select within a Range of Values 34
Using the CONTAINS or Question Mark (?) Operator to Select a String 34
Example 35
Using the IN Operator to Select Values from a List 35
Using the IS MISSING or IS NULL Operator to Select Missing Values 36
Example 36
Using the LIKE Operator to Select a Pattern 37
Specifying a Pattern 38
Example 38
Using the Sounds-Like (=*) Operator to Select a Spelling Variation 39
Subsetting Rows by Using Calculated Values 40
Understanding How PROC SQL Processes Calculated Columns 40
Using the Keyword CALCULATED 40
Enhancing Query Output 42
Specifying Column Formats and Labels 43
Specifying Titles and Footnotes 44
Adding a Character Constant to Output 45
Summarizing and Grouping Data 47
Number of Arguments and Summary Function Processing 48
Groups and Summary Function Processing 48
SELECT Clause Columns and Summary Function Processing 48
Using a Summary Function with a Single Argument (Column) 49
Using a Summary Function with Multiple Arguments (Columns) 50
Using a Summary Function without a GROUP BY Clause 50
Using a Summary Function with Columns Outside of the Function 51
Using a Summary Function with a GROUP BY Clause 52
Counting Values by Using the COUNT Summary Function 53
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
<th>Chapter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting All Rows</td>
<td>53</td>
<td>Counting All Non-Missing Values in a Column</td>
</tr>
<tr>
<td>Counting All Unique Values in a Column</td>
<td>55</td>
<td>Selecting Groups by Using the HAVING Clause</td>
</tr>
<tr>
<td>Understanding Data Remerging</td>
<td>57</td>
<td>Example</td>
</tr>
<tr>
<td>Subsetting Data by Using Subqueries</td>
<td>58</td>
<td>Introducing Subqueries</td>
</tr>
<tr>
<td>Types of Subqueries</td>
<td>59</td>
<td>Subsetting Data by Using Noncorrelated Subqueries</td>
</tr>
<tr>
<td>Using Single-Value Noncorrelated Subqueries</td>
<td>60</td>
<td>Using Multiple-Value Noncorrelated Subqueries</td>
</tr>
<tr>
<td>Example</td>
<td>61</td>
<td>Using Comparisons with Subqueries</td>
</tr>
<tr>
<td>Using the ANY Operator</td>
<td>62</td>
<td>Example</td>
</tr>
<tr>
<td>Using the ALL Operator</td>
<td>64</td>
<td>Example</td>
</tr>
<tr>
<td>Subsetting Data by Using Correlated Subqueries</td>
<td>65</td>
<td>Example</td>
</tr>
<tr>
<td>Using the EXISTS and NOT EXISTS Conditional Operators</td>
<td>66</td>
<td>Example: Correlated Subquery with NOT EXISTS</td>
</tr>
<tr>
<td>Validating Query Syntax</td>
<td>67</td>
<td>Using the NOEXEC Option</td>
</tr>
<tr>
<td>Using the VALIDATE Keyword</td>
<td>68</td>
<td>Additional Features</td>
</tr>
<tr>
<td>Summary</td>
<td>70</td>
<td>Text Summary</td>
</tr>
<tr>
<td>Viewing SELECT Statement Syntax</td>
<td>70</td>
<td>Displaying All Columns</td>
</tr>
<tr>
<td>Limiting the Number of Rows Displayed</td>
<td>70</td>
<td>Eliminating Duplicate Rows from Output</td>
</tr>
<tr>
<td>Subsetting Rows by Using Conditional Operators</td>
<td>70</td>
<td>Subsetting Rows by Using Calculated Values</td>
</tr>
<tr>
<td>Enhancing Query Output</td>
<td>71</td>
<td>Summarizing and Grouping Data</td>
</tr>
<tr>
<td>Subsetting Data by Using Subqueries</td>
<td>71</td>
<td>Subsetting Data by Using Noncorrelated Subqueries</td>
</tr>
<tr>
<td>Subsetting Data by Using Correlated Subqueries</td>
<td>71</td>
<td>Validating Query Syntax</td>
</tr>
<tr>
<td>Additional Features</td>
<td>71</td>
<td>Syntax</td>
</tr>
<tr>
<td>Sample Programs</td>
<td>72</td>
<td>Displaying all Columns in Output and an Expanded Column List in the SAS Log</td>
</tr>
<tr>
<td>Eliminating Duplicate Rows from Output</td>
<td>72</td>
<td>Subsetting Rows by Using Calculated Values</td>
</tr>
<tr>
<td>Subsetting Data by Using a Noncorrelated Subquery</td>
<td>73</td>
<td>Subsetting Data by Using a Correlated Subquery</td>
</tr>
<tr>
<td>Points to Remember</td>
<td>73</td>
<td>Quiz</td>
</tr>
</tbody>
</table>
Overview

Introduction

The SELECT statement is the primary tool of PROC SQL. Using the SELECT statement, you can identify, manipulate, and retrieve columns of data from one or more tables and views.

You should already know how to create basic PROC SQL queries by using the SELECT statement and most of its subordinate clauses. To build on your existing skills, this chapter presents a variety of useful query techniques, such as the use of subqueries to subset data.

The PROC SQL query shown below illustrates some of the new query techniques that you will learn:

```sql
proc sql outobs=20;
  title 'Job Groups with Average Salary';
  title2 ' > Company Average';
  select jobcode,
    avg(salary) as AvgSalary format=dollar11.2,
    count(*) as Count
  from sasuser.payrollmaster
  group by jobcode
  having avg(salary) >
    (select avg(salary)
     from sasuser.payrollmaster)
  order by avgsalary desc;
```

### Job Groups with Average Salary

> Company Average

<table>
<thead>
<tr>
<th>JobCode</th>
<th>AvgSalary</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT3</td>
<td>$154,706.50</td>
<td>2</td>
</tr>
<tr>
<td>PT2</td>
<td>$122,263.40</td>
<td>10</td>
</tr>
<tr>
<td>PT1</td>
<td>$95,071.13</td>
<td>8</td>
</tr>
<tr>
<td>NA2</td>
<td>$73,336.00</td>
<td>3</td>
</tr>
<tr>
<td>ME3</td>
<td>$59,374.86</td>
<td>7</td>
</tr>
<tr>
<td>NA1</td>
<td>$58,845.00</td>
<td>5</td>
</tr>
<tr>
<td>TA3</td>
<td>$55,551.42</td>
<td>12</td>
</tr>
</tbody>
</table>

Objectives

In this chapter, you learn to

- display all rows, eliminate duplicate rows, and limit the number of rows displayed
- subset rows using other conditional operators and calculated values
- enhance the formatting of query output
Prerequisites

Before you begin this chapter, you should complete the following chapter:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3.

Viewing SELECT Statement Syntax

The SELECT statement and its subordinate clauses are the building blocks for constructing all PROC SQL queries.

General form, SELECT statement:

```
SELECT column-1<, ... column-n>
FROM table-1 | view-1<, ... table-n | view-n>
<WHERE expression>
<GROUP BY column-1<, ... column-n>>
<HAVING expression>
<ORDER BY column-1<, ... column-n>>;
```

where

- **SELECT** specifies the column(s) that will appear in the output
- **FROM** specifies the table(s) or view(s) to be queried
- **WHERE** uses an expression to subset or restrict the data based on one or more condition(s)
- **GROUP BY** classifies the data into groups based on the specified column(s)
- **HAVING** uses an expression to subset or restrict groups of data based on a group condition
- **ORDER BY** sorts the rows that the query returns by the value(s) of the specified column(s).

Note: The clauses in a PROC SQL SELECT statement *must* be specified in the order shown.

You should be familiar with all of the SELECT statement clauses except for the HAVING clause. The use of the HAVING clause is presented later in this chapter.

Now, let’s look at some ways you can limit and subset the number of columns that will be displayed in query output.
Displaying All Columns

You already know how to select specific columns for output by listing them in the SELECT statement. However, for some tasks, you will find it useful to display all columns of a table concurrently. For example, before you create a complex query, you might want to see the contents of the table you are working with.

Using SELECT *

To display all columns in the order in which they are stored in a table, use an asterisk (*) in the SELECT clause. All rows will also be displayed, by default, unless you limit or subset them.

The following SELECT step displays all columns and rows in the table Sasuser.Staffchanges, which lists all employees in a company who have had changes in their employment status. As shown in the output, below, the table contains six columns and six rows.

```sql
proc sql;
  select * from sasuser.staffchanges;
```

Using the FEEDBACK Option

When you specify SELECT *, you can also use the FEEDBACK option in the PROC SQL statement, which writes the expanded list of columns to the SAS log. For example, the PROC SQL query shown below contains the FEEDBACK option:

```sql
proc sql feedback;
  select * from sasuser.staffchanges;
```

This query produces the following feedback in the SAS log.

Table 2.1  SAS Log

<table>
<thead>
<tr>
<th>EmpID</th>
<th>LastName</th>
<th>FirstName</th>
<th>City</th>
<th>State</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>1036</td>
<td>CARTER</td>
<td>KAREN</td>
<td>STANFORD</td>
<td>CT</td>
<td>303/783-6819</td>
</tr>
<tr>
<td>1086</td>
<td>CHAPMAN</td>
<td>NEIL</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-6516</td>
</tr>
<tr>
<td>1591</td>
<td>SANDERS</td>
<td>RAYMOND</td>
<td>NEW YORK</td>
<td>NY</td>
<td>212/688-6215</td>
</tr>
<tr>
<td>1221</td>
<td>WALTERS</td>
<td>DIANE</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-6918</td>
</tr>
<tr>
<td>1447</td>
<td>BRIDESTON</td>
<td>AMY</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-1213</td>
</tr>
<tr>
<td>1980</td>
<td>POWELL</td>
<td>JIM</td>
<td>NEW YORK</td>
<td>NY</td>
<td>718/384-6042</td>
</tr>
</tbody>
</table>

The FEEDBACK option is a debugging tool that lets you see exactly what is being submitted to the SQL processor. The resulting message in the SAS log not only expands
asterisks (*) into column lists, but it also resolves macro variables and places parentheses around expressions to show their order of evaluation.

## Limiting the Number of Rows Displayed

When you create PROC SQL queries, you will sometimes find it useful to limit the number of rows that PROC SQL displays in the output. To indicate the maximum number of rows to be displayed, you can use the OUTOBS= option in the PROC SQL statement. OUTOBS= is similar to the OBS= data set option.

**General form, PROC SQL statement with OUTOBS= option:**

```sql
PROC SQL OUTOBS= n;
```

where

- `n` specifies the number of rows.

**Note:** The OUTOBS= option restricts the rows that are displayed, but not the rows that are read. To restrict the number of rows that PROC SQL takes as input from any single source, use the INOBS= option. For more information about the INOBS= option, see Chapter 8, “Managing Processing Using PROC SQL,” on page 261.

### Example

Suppose you want to quickly review the kinds of values that are stored in a table, without printing out all the rows. The following PROC SQL query selects data from the table `Sasuser.Flightschedule`, which contains over 200 rows. To print only the first 10 rows of output, you add the OUTOBS= option to the PROC SQL statement.

```sql
proc sql outobs=10;
    select flightnumber, date
    from sasuser.flightschedule;
```

When you limit the number of rows that are displayed, a message similar to the following appears in the SAS log.
Table 2.2 SAS Log

**WARNING:** Statement terminated early due to OUTOBS=10 option.

*Note:* The OUTOBS= and INOBS= options will affect tables that are created by using the CREATE TABLE statement and your report output. ▲

*Note:* In many of the examples in this chapter, OUTOBS= is used to limit the number of rows that are displayed in output. ▲

---

**Eliminating Duplicate Rows from Output**

In some situations, you might want to display only the unique values or combinations of values in the column(s) listed in the SELECT clause. You can eliminate duplicate rows from your query results by using the keyword *DISTINCT* in the SELECT clause. The DISTINCT keyword applies to all columns, and only those columns, that are listed in the SELECT clause. Let’s see how this works.

---

**Example**

Suppose you want to display a list of the unique flight numbers and destinations of all international flights that are flown during the month.

The following SELECT statement in PROC SQL selects the columns **FlightNumber** and **Destination** in the table *Sasuser.Internationalflights*:

```
proc sql outobs=12;
    select flightnumber, destination
    from sasuser.internationalflights;
```

Here is the output.

<table>
<thead>
<tr>
<th>FlightNumber</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>182</td>
<td>YYZ</td>
</tr>
<tr>
<td>219</td>
<td>LHR</td>
</tr>
<tr>
<td>367</td>
<td>CPH</td>
</tr>
<tr>
<td>622</td>
<td>FRA</td>
</tr>
<tr>
<td>321</td>
<td>LHR</td>
</tr>
<tr>
<td>132</td>
<td>YYZ</td>
</tr>
<tr>
<td>271</td>
<td>CDG</td>
</tr>
<tr>
<td>182</td>
<td>YYZ</td>
</tr>
<tr>
<td>219</td>
<td>LHR</td>
</tr>
<tr>
<td>367</td>
<td>CPH</td>
</tr>
<tr>
<td>622</td>
<td>FRA</td>
</tr>
<tr>
<td>321</td>
<td>LHR</td>
</tr>
</tbody>
</table>

As you can see, there are several duplicate pairs of values for **FlightNumber** and **Destination** in the first 12 rows alone. For example, flight number 182 to YYZ appears in rows 1 and 8. The entire table contains many more rows with duplicate values for each flight number and destination because each flight has a regular schedule.
To remove rows that contain duplicate values, add the keyword DISTINCT to the SELECT statement, following the keyword SELECT, as shown in the following example:

```sql
proc sql;
   select distinct flightnumber, destination
       from sasuser.internationalflights
       order by 1;
```

With duplicate values removed, the output will contain many fewer rows, so the OUTOBS= option has been removed from the PROC SQL statement. Also, to sort the output by **FlightNumber** (column 1 in the SELECT clause list), the ORDER BY clause has been added.

Here is the output from the modified program.

<table>
<thead>
<tr>
<th>FlightNumber</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>YYZ</td>
</tr>
<tr>
<td>182</td>
<td>YYZ</td>
</tr>
<tr>
<td>219</td>
<td>LHR</td>
</tr>
<tr>
<td>271</td>
<td>CDG</td>
</tr>
<tr>
<td>387</td>
<td>CPH</td>
</tr>
<tr>
<td>622</td>
<td>FRA</td>
</tr>
<tr>
<td>821</td>
<td>LHR</td>
</tr>
</tbody>
</table>

There are no duplicate rows in the output. There are seven unique **FlightNumber-Destination** value pairs in this table.

---

**Subsetting Rows by Using Conditional Operators**

In the WHERE clause of a PROC SQL query, you can specify any valid SAS expression to subset or restrict the data that is displayed in output. The expression may contain any of various types of operators, such as the following.

<table>
<thead>
<tr>
<th>Type of Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparison</td>
<td><code>where membertype='GOLD'</code></td>
</tr>
<tr>
<td>logical</td>
<td><code>where visits&lt;=3 or status='new'</code></td>
</tr>
<tr>
<td>concatenation</td>
<td>`where name=trim(last)</td>
</tr>
</tbody>
</table>

**Note:** For a complete list of operators that can be used in SAS expressions, see the SAS documentation.

---

**Using Operators in PROC SQL**

Comparison, logical, and concatenation operators are used in PROC SQL as they are used in other SAS procedures. For example, the following WHERE clause contains

- the logical operator **AND**, which joins multiple conditions
two comparison operators: an equal sign (=) and a greater than symbol (>).

```
proc sql;
    select ffid, name, state, pointsused
    from sasuser.frequentflyers
    where membertype='GOLD' and pointsused>0
    order by pointsused;
```

In PROC SQL queries, you can also use the following **conditional operators**. All of these operators except for ANY, ALL, and EXISTS, can also be used in other SAS procedures.

<table>
<thead>
<tr>
<th>Conditional Operator</th>
<th>Tests for ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN-AND</td>
<td>values that occur within an inclusive range</td>
<td><code>where salary BETWEEN 70000 and 80000</code></td>
</tr>
<tr>
<td>CONTAINS or ?</td>
<td>values that contain a specified string</td>
<td><code>where name CONTAINS 'ER'</code></td>
</tr>
<tr>
<td>IN</td>
<td>values that match one of a list of values</td>
<td><code>where code IN ('PT', 'NA', 'FA')</code></td>
</tr>
<tr>
<td>IS MISSING or IS NULL</td>
<td>missing values</td>
<td><code>where dateofbirth IS MISSING</code></td>
</tr>
<tr>
<td>LIKE (with %, _)</td>
<td>values that match a specified pattern</td>
<td><code>where address LIKE '% P%PLACE'</code></td>
</tr>
<tr>
<td>=*</td>
<td>values that sound like a specified value</td>
<td><code>where lastname=*= 'Smith'</code></td>
</tr>
<tr>
<td>ANY</td>
<td>values that meet a specified condition with respect to any one of the values returned by a subquery</td>
<td><code>where dateofbirth &lt; any (select dateofbirth from sasuser.payrollmaster where jobcode='FA3')</code></td>
</tr>
<tr>
<td>ALL</td>
<td>values that meet a specified condition with respect to all the values returned by a subquery</td>
<td><code>where dateofbirth &lt; all (select dateofbirth from sasuser.payrollmaster where jobcode='FA3')</code></td>
</tr>
<tr>
<td>EXISTS</td>
<td>the existence of values returned by a subquery</td>
<td><code>where exists (select * from sasuser.flightschedule where fa.empid= flightschedule.empid)</code></td>
</tr>
</tbody>
</table>

**Tip:** To create a negative condition, you can precede any of these conditional operators, except for ANY and ALL, with the `NOT` operator.

Most of these conditional operators, and their uses, are covered in the next several sections. ANY, ALL, and EXISTS are discussed later in the chapter.
Using the BETWEEN-AND Operator to Select within a Range of Values

To select rows based on a range of numeric or character values, you use the BETWEEN-AND operator in the WHERE clause. The BETWEEN-AND operator is inclusive, so the values that you specify as limits for the range of values are included in the query results, in addition to any values that occur between the limits.

General form, BETWEEN-AND operator:

\[ \text{BETWEEN value-1 AND value-2} \]

where

- \( \text{value-1} \) is the value at the one end of the range
- \( \text{value-2} \) is the value at the other end of the range.

**Note:** When specifying the limits for the range of values, it is not necessary to specify the smaller value first.

Following are several examples of WHERE clauses that contain the BETWEEN-AND operator. The last example shows the use of the NOT operator with the BETWEEN-AND operator.

<table>
<thead>
<tr>
<th>Example</th>
<th>Returns rows in which...</th>
</tr>
</thead>
<tbody>
<tr>
<td>where date between '01mar2000'd and '07mar2000'd</td>
<td>the value of Date is 01mar2000, 07mar2000, or any date value in between</td>
</tr>
<tr>
<td>where salary between 70000 and 80000</td>
<td>the value of Salary is 70000, 80000, or any numeric value in between</td>
</tr>
<tr>
<td>where salary not between 70000 and 80000</td>
<td>the value of Salary is not between or equal to 70000 and 80000</td>
</tr>
</tbody>
</table>

Using the CONTAINS or Question Mark (?) Operator to Select a String

The CONTAINS or question mark (?) operator is usually used to select rows for which a character column includes a particular string. These operators are interchangeable.
Performing Advanced Queries Using PROC SQL

Using the IN Operator to Select Values from a List

General form, CONTAINS operator:

```sql
sql-expression  CONTAINS  sql-expression
```

where

```sql
sql-expression
```

is a character column, string (character constant), or expression. A string is a sequence of characters to be matched that must be enclosed in quotation marks.

**Note:** PROC SQL retrieves a row for output no matter where the string (or second sql-expression) occurs within the column's (or first sql-expression's) values. Matching is case sensitive when making comparisons.

**Note:** The CONTAINS or question mark (?) operator is not part of the ANSI standard; it is a SAS enhancement.

**Example**

The following PROC SQL query uses CONTAINS to select rows in which the Name column contains the string ER. As the output shows, all rows that contain ER anywhere within the Name column are displayed.

```
proc sql outobs=10;
    select name
        from sasuser.frequentflyers
    where name contains 'ER';
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>COOPER, LESLIE</td>
</tr>
<tr>
<td>COOPER, ANTHONY</td>
</tr>
<tr>
<td>COOK, JENNIFER</td>
</tr>
<tr>
<td>FOSTER, GERALD</td>
</tr>
<tr>
<td>HAYLEY, JEREMY</td>
</tr>
<tr>
<td>BURKE, CHRISTOPHER</td>
</tr>
<tr>
<td>AVERY, JERRY</td>
</tr>
<tr>
<td>EDGEKTON, JOSHUA</td>
</tr>
<tr>
<td>SAYERS, RANDY</td>
</tr>
<tr>
<td>WANG, CHRISTOPHER</td>
</tr>
</tbody>
</table>

**Using the IN Operator to Select Values from a List**

To select only the rows that match one of the values in a list of fixed values, either numeric or character, use the IN operator.
Using the IS MISSING or IS NULL Operator to Select Missing Values

To select rows that contain missing values, both character and numeric, use the IS MISSING or IS NULL operator. These operators are interchangeable.

General form, IS MISSING or IS NULL operator:

\[
\text{column IS MISSING} \\
\text{column IS NULL} \\
\]

where

\[
\text{column} \\
\text{specifies the selected column name.} \\
\]

Example

Suppose you want to find out whether the table Sasuser. Marchflights has any missing values in the column Boarded. You can use the following PROC SQL query to retrieve rows from the table that have missing values:

```sql
proc sql;
select boarded, transferred, 
    nonrevenue, deplaned 
from sasuser.marchflights 
where boarded is missing;
```

The output shows that two rows in the table have missing values for Boarded.
Tip: Alternatively, you can specify missing values without using the IS MISSING or IS NULL operator, as shown in the following examples:

```sql
where boarded = .
where flight = '
```

However, the advantage of using the IS MISSING or IS NULL operator is that you don’t have to specify the data type (character or numeric) of the column.

### Using the LIKE Operator to Select a Pattern

To select rows that have values that match a specific pattern of characters rather than a fixed character string, use the `LIKE` operator. For example, using the LIKE operator, you can select all rows in which the `LastName` value starts with `H`. (If you wanted to select all rows in which the last name contains the string `HAR`, you would use the CONTAINS operator.)

**General form, LIKE operator:**

```
column LIKE 'pattern'
```

where

- `column` specifies the column name
- `pattern` specifies the pattern to be matched and contains one or both of the special characters underscore (`_`) and percent sign (`%`). The entire pattern must be enclosed in quotation marks and matching is case sensitive.

When you use the LIKE operator in a query, PROC SQL uses pattern matching to compare each value in the specified column with the pattern that you specify using the LIKE operator in the WHERE clause. The query output displays all rows in which there is a match.

You specify a pattern using one or both of the special characters shown below.

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>underscore ( _)</td>
<td>any single character</td>
</tr>
<tr>
<td>percent sign (%)</td>
<td>any sequence of zero or more characters</td>
</tr>
</tbody>
</table>

*Note:* The underscore ( _) and percent sign (%) are sometimes referred to as wildcard characters.
Specifying a Pattern

To specify a pattern, combine one or both of the special characters with any other characters that you want to match. The special characters can appear before, after, or on both sides of other characters.

Let’s look at how the special characters can be combined to specify a pattern. Suppose you are working with a table column that contains the following list of names:

- Diana
- Diane
- Dianna
- Dianthus
- Dyan

Here are several different patterns that you can use to select one or more of the names from the list. Each pattern uses one or both of the special characters.

<table>
<thead>
<tr>
<th>LIKE Pattern</th>
<th>Name(s) Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIKE ‘D_an’</td>
<td>Dyan</td>
</tr>
<tr>
<td>LIKE ‘D_an_’</td>
<td>Diana, Diane</td>
</tr>
<tr>
<td>LIKE ‘D_an__’</td>
<td>Dianna</td>
</tr>
<tr>
<td>LIKE ‘D_an%’</td>
<td>all names from the list</td>
</tr>
</tbody>
</table>

Example

The following PROC SQL query uses the LIKE operator to find all frequent-flyer club members whose street name begins with *P* and ends with the word *PLACE*. The following PROC SQL step performs this query:

```sql
proc sql;
    select ffid, name, address
    from sasuser.frequentflyers
    where address like '% P%PLACE';
```

The pattern ‘% P%PLACE’ specifies the following sequence:

- any number of characters (%)
- a space
- the letter *P*
- any number of characters (%)
- the word *PLACE*.

Here are the results of this query.
Performing Advanced Queries Using PROC SQL

Using the Sounds-Like (=*) Operator to Select a Spelling Variation

To select rows that contain a value that sounds like another value that you specify, use the sounds-like operator (=*) in the WHERE clause.

General form, sounds-like (=*) operator:

\[
\text{sql-expression} =* \text{sql-expression}
\]

where

- \text{sql-expression} is a character column, string (character constant), or expression. A string is a sequence of characters to be matched that must be enclosed in quotation marks.

The sounds-like (=*) operator uses the SOUNDEX algorithm to compare each value of a column (or other sql-expression) with the word or words (or other sql-expression) that you specify. Any rows that contain a spelling variation of the value that you specified are selected for output.

For example, here is a WHERE clause that contains the sounds-like operator:

\[
\text{where lastname } =* \text{ 'Smith'};
\]

The sounds-like operator does not always select all possible values. For example, suppose you use the preceding WHERE clause to select rows from the following list of names that sound like Smith:

- Schmitt
- Smith
- Smithson
- Smitt
- Smythe

Two of the names in this list will not be selected: Schmitt and Smithson.

Note: The SOUNDEX algorithm is English-biased and is less useful for languages other than English. For more information about the SOUNDEX algorithm, see the SAS documentation.
Subsetting Rows by Using Calculated Values

Understanding How PROC SQL Processes Calculated Columns

You should already know how to define a new column by using the SELECT clause and performing a calculation. For example, the following PROC SQL query creates the new column **Total** by adding the values of three existing columns: **Boarded**, **Transferred**, and **Nonrevenue**:

```sql
proc sql outobs=10;
  select flightnumber, date, destination,
       boarded + transferred + nonrevenue
       as Total
  from sasuser.marchflights
```

You can also use a calculated column in the WHERE clause to subset rows. However, because of the way in which SQL queries are processed, you cannot just specify the column alias in the WHERE clause. To see what happens, let's take the preceding PROC SQL query and add a WHERE clause in the SELECT statement to reference the calculated column **Total**, as shown below:

```sql
proc sql outobs=10;
  select flightnumber, date, destination,
       boarded + transferred + nonrevenue
       as Total
  from sasuser.marchflights
  where total < 100;
```

When this query is executed, the following error message is displayed in the SAS log.

Table 2.3 SAS Log

```
  519 proc sql outobs=10;
  520   select flightnumber, date, destination,
  521      boarded + transferred + nonrevenue
  522       as Total
  523    from sasuser.marchflights
  524   where total < 100;
 ERROR: The following columns were not found in the contributing tables: total.
```

This error message is generated because, in SQL queries, the WHERE clause is processed prior to the SELECT clause. The SQL processor looks in the table for each column named in the WHERE clause. To see what happens, let's take the preceding PROC SQL query and add a WHERE clause in the SELECT statement to reference the calculated column **Total**, as shown below:

```sql
proc sql outobs=10;
  select flightnumber, date, destination,
       boarded + transferred + nonrevenue
       as Total
  from sasuser.marchflights
  where total < 100;
```

Using the Keyword **CALCULATED**

When you use a column alias in the WHERE clause to refer to a calculated value, you must use the keyword **CALCULATED** along with the alias. The **CALCULATED** keyword informs PROC SQL that the value is calculated within the query. Now, the PROC SQL query looks like this:

```sql
proc sql outobs=10;
  select flightnumber, date, destination,
       boarded + transferred + nonrevenue
       as Total
  from sasuser.marchflights
  where total < 100;
```
Performing Advanced Queries Using PROC SQL

Using the Keyword CALCULATED

This query executes successfully and produces the following output.

<table>
<thead>
<tr>
<th>FlightNumber</th>
<th>Date</th>
<th>Destination</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>982</td>
<td>01MAR2000</td>
<td>DFW</td>
<td>70</td>
</tr>
<tr>
<td>416</td>
<td>01MAR2000</td>
<td>WAS</td>
<td>93</td>
</tr>
<tr>
<td>829</td>
<td>01MAR2000</td>
<td>WAS</td>
<td>96</td>
</tr>
<tr>
<td>416</td>
<td>02MAR2000</td>
<td>WAS</td>
<td>90</td>
</tr>
<tr>
<td>302</td>
<td>02MAR2000</td>
<td>WAS</td>
<td>93</td>
</tr>
<tr>
<td>132</td>
<td>03MAR2000</td>
<td>YZ</td>
<td>88</td>
</tr>
<tr>
<td>921</td>
<td>03MAR2000</td>
<td>DFW</td>
<td>85</td>
</tr>
<tr>
<td>290</td>
<td>05MAR2000</td>
<td>WAS</td>
<td>55</td>
</tr>
<tr>
<td>523</td>
<td>05MAR2000</td>
<td>ORD</td>
<td>59</td>
</tr>
<tr>
<td>416</td>
<td>05MAR2000</td>
<td>WAS</td>
<td>31</td>
</tr>
</tbody>
</table>

Note: As an alternative to using the keyword CALCULATED, repeat the calculation in the WHERE clause. However, this method is inefficient because PROC SQL has to perform the calculation twice. In the preceding query, the alternate WHERE statement would be:

```
where boarded + transferred + nonrevenue <100;
```

You can also use the CALCULATED keyword in other parts of a query. In the following example, the SELECT clause calculates the new column Total and then calculates a second new column based on Total. To create the second calculated column, you have to specify the keyword CALCULATED in the SELECT clause.

```
proc sql outobs=10;
  select flightnumber, date, destination,
        boarded + transferred + nonrevenue
        as Total,
        calculated total/2 as Half
  from sasuser.marchflights;
```

This query produces the following output.
Enhancing Query Output

When you are using PROC SQL, you might find that the data in a table is not formatted as you would like it to appear. Fortunately, with PROC SQL you can use enhancements, such as the following, to improve the appearance of your query output:

- column labels and formats
- titles and footnotes
- columns that contain a character constant.

You know how to use the first two enhancements with other SAS procedures. Now let's see how to enhance PROC SQL query output, working with the following query:

```sql
proc sql outobs=15;
select empid, jobcode, salary,
    salary * .10 as Bonus
from sasuser.payrollmaster
where salary>75000
order by salary desc;
```

This query limits output to 15 observations. The SELECT clause selects three existing columns from the table `Sasuser.Payrollmaster`, and calculates a fourth (`Bonus`). The WHERE clause retrieves only rows in which salary is greater than 75,000. The ORDER BY clause sorts by the `Salary` column and uses the keyword DESC to sort in descending order.

Here is the output from this query:

<table>
<thead>
<tr>
<th>FlightNumber</th>
<th>Date</th>
<th>Destination</th>
<th>Total</th>
<th>Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>182</td>
<td>01MAR2000</td>
<td>YYZ</td>
<td>123</td>
<td>61.5</td>
</tr>
<tr>
<td>114</td>
<td>01MAR2000</td>
<td>LAX</td>
<td>196</td>
<td>98</td>
</tr>
<tr>
<td>202</td>
<td>01MAR2000</td>
<td>ORD</td>
<td>167</td>
<td>83.5</td>
</tr>
<tr>
<td>219</td>
<td>01MAR2000</td>
<td>LHR</td>
<td>222</td>
<td>111</td>
</tr>
<tr>
<td>439</td>
<td>01MAR2000</td>
<td>LAX</td>
<td>185</td>
<td>92.5</td>
</tr>
<tr>
<td>367</td>
<td>01MAR2000</td>
<td>DPH</td>
<td>163</td>
<td>81.5</td>
</tr>
<tr>
<td>250</td>
<td>01MAR2000</td>
<td>WAS</td>
<td>119</td>
<td>59.5</td>
</tr>
<tr>
<td>523</td>
<td>01MAR2000</td>
<td>ORD</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>982</td>
<td>01MAR2000</td>
<td>DFW</td>
<td>70</td>
<td>35</td>
</tr>
<tr>
<td>622</td>
<td>01MAR2000</td>
<td>FRA</td>
<td>227</td>
<td>113.5</td>
</tr>
</tbody>
</table>

Note: The CALCULATED keyword is a SAS enhancement and is not specified in the ANSI Standard for SQL.
Performing Advanced Queries Using PROC SQL

Specifying Column Formats and Labels

By default, PROC SQL formats output using column attributes that are already saved in the table or, if none are saved, the default attributes. To control the formatting of columns in output, you can specify SAS data set options, such as LABEL= and FORMAT=, after any column name specified in the SELECT clause. When you define a new column in the SELECT clause, you can assign a label rather than an alias, if you prefer.

<table>
<thead>
<tr>
<th>Data Set Option</th>
<th>Specifies...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL=</td>
<td>the label to be displayed for the column</td>
<td>select hiredate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>label='Date of Hire'</td>
</tr>
<tr>
<td>FORMAT=</td>
<td>the format used to display column data</td>
<td>select hiredate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>format=date9.</td>
</tr>
</tbody>
</table>

Note: The data set options LABEL= and FORMAT= are not part of the ANSI standard. These options are SAS enhancements.
Tip: To force PROC SQL to ignore permanent labels in a table, specify the NOLABEL system option.

Your first task is to specify column labels for the first two columns. Below, the LABEL= option has been added after both EmpID and JobCode, and the text of each label is enclosed in quotation marks. For easier reading, each of the four columns in the SELECT clause is now listed on its own line.

```sql
proc sql outobs=15;
    select empid label='Employee ID',
           jobcode label='Job Code',
           salary,
           salary * .10 as Bonus
    from sasuser.payrollmaster
    where salary>75000
    order by salary desc;
```

Next, you will add a format for the Bonus column. Because the Bonus values are dollar amounts, you will use the format Dollar12.2. The FORMAT= option has been added to the SELECT clause, below, immediately following the column alias Bonus:

```sql
proc sql outobs=15;
    select empid label='Employee ID',
           jobcode label='Job Code',
           salary,
           salary * .10 as Bonus
           format=dollar12.2
    from sasuser.payrollmaster
    where salary>75000
    order by salary desc;
```

Now that column formats and labels have been specified, let’s add a title to this PROC SQL query.

---

Specifying Titles and Footnotes

You should already know how to specify and cancel titles and footnotes with other SAS procedures. When you specify titles and footnotes with a PROC SQL query, you must place the TITLE and FOOTNOTE statements in either of the following locations:

- before the PROC SQL statement
- between the PROC SQL statement and the SELECT statement.

In the following PROC SQL query, two title lines have been added between the PROC SQL statement and the SELECT statement:

```sql
proc sql outobs=15;
    title 'Current Bonus Information';
    title2 'Employees with Salaries > $75,000';
    select empid label='Employee ID',
           jobcode label='Job Code',
           salary,
           salary * .10 as Bonus
           format=dollar12.2
    from sasuser.payrollmaster
    where salary>75000
    order by salary desc;
```

Now that these changes have been made, let’s look at the enhanced query output.
Performing Advanced Queries Using PROC SQL

Adding a Character Constant to Output

The first two columns have new labels, the Bonus values are consistently formatted, and two title lines are displayed at the top of the output.

### Adding a Character Constant to Output

Another way of enhancing PROC SQL query output is to define a column that contains a character constant. To do this, you include a text string in quotation marks in the SELECT clause.

**Tip:** You can define a column that contains a numeric constant in a similar way, by listing a numeric value (without quotation marks) in the SELECT clause.

Let’s look at the preceding PROC SQL query output again and determine where you can add a text string.
Let's remove the column label **Bonus** and display the text *bonus is:* in a new column to the left of the **Bonus** column. This is how you want the columns and rows to appear in the query output.

### Current Bonus Information
**Employees with Salaries > $75,000**

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Job Code</th>
<th>Salary</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>PT3</td>
<td>$155,931 $15,593.10</td>
<td></td>
</tr>
<tr>
<td>1777</td>
<td>PT3</td>
<td>$153,482 $15,348.20</td>
<td></td>
</tr>
<tr>
<td>1404</td>
<td>PT2</td>
<td>$127,926 $12,792.60</td>
<td></td>
</tr>
<tr>
<td>1107</td>
<td>PT2</td>
<td>$125,988 $12,598.80</td>
<td></td>
</tr>
<tr>
<td>1928</td>
<td>PT2</td>
<td>$125,801 $12,580.10</td>
<td></td>
</tr>
<tr>
<td>1108</td>
<td>PT2</td>
<td>$125,485 $12,548.50</td>
<td></td>
</tr>
<tr>
<td>1333</td>
<td>PT2</td>
<td>$124,048 $12,404.80</td>
<td></td>
</tr>
<tr>
<td>1890</td>
<td>PT2</td>
<td>$120,254 $12,025.40</td>
<td></td>
</tr>
<tr>
<td>1410</td>
<td>PT2</td>
<td>$118,559 $11,855.90</td>
<td></td>
</tr>
<tr>
<td>1442</td>
<td>PT2</td>
<td>$118,350 $11,835.00</td>
<td></td>
</tr>
<tr>
<td>1830</td>
<td>PT2</td>
<td>$118,259 $11,825.90</td>
<td></td>
</tr>
<tr>
<td>1478</td>
<td>PT2</td>
<td>$117,984 $11,798.40</td>
<td></td>
</tr>
<tr>
<td>1568</td>
<td>PT1</td>
<td>$99,889  $9,888.90</td>
<td></td>
</tr>
<tr>
<td>1439</td>
<td>PT1</td>
<td>$99,030  $9,030.00</td>
<td></td>
</tr>
<tr>
<td>1428</td>
<td>PT1</td>
<td>$98,274  $9,827.40</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Job Code</th>
<th>Salary</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>PT3</td>
<td>$155,931 bonus is: $15,593.10</td>
<td></td>
</tr>
<tr>
<td>1777</td>
<td>PT3</td>
<td>$153,482 bonus is: $15,348.20</td>
<td></td>
</tr>
<tr>
<td>1404</td>
<td>PT2</td>
<td>$127,926 bonus is: $12,792.60</td>
<td></td>
</tr>
<tr>
<td>1107</td>
<td>PT2</td>
<td>$125,988 bonus is: $12,598.80</td>
<td></td>
</tr>
<tr>
<td>1928</td>
<td>PT2</td>
<td>$125,801 bonus is: $12,580.10</td>
<td></td>
</tr>
<tr>
<td>1108</td>
<td>PT2</td>
<td>$125,485 bonus is: $12,548.50</td>
<td></td>
</tr>
<tr>
<td>1333</td>
<td>PT2</td>
<td>$124,048 bonus is: $12,404.80</td>
<td></td>
</tr>
<tr>
<td>1890</td>
<td>PT2</td>
<td>$120,254 bonus is: $12,025.40</td>
<td></td>
</tr>
<tr>
<td>1410</td>
<td>PT2</td>
<td>$116,559 bonus is: $11,855.90</td>
<td></td>
</tr>
<tr>
<td>1442</td>
<td>PT2</td>
<td>$116,350 bonus is: $11,835.00</td>
<td></td>
</tr>
<tr>
<td>1830</td>
<td>PT2</td>
<td>$116,259 bonus is: $11,825.90</td>
<td></td>
</tr>
<tr>
<td>1478</td>
<td>PT2</td>
<td>$117,984 bonus is: $11,798.40</td>
<td></td>
</tr>
<tr>
<td>1568</td>
<td>PT1</td>
<td>$99,889  bonus is: $9,888.90</td>
<td></td>
</tr>
<tr>
<td>1439</td>
<td>PT1</td>
<td>$99,030  bonus is: $9,030.00</td>
<td></td>
</tr>
<tr>
<td>1428</td>
<td>PT1</td>
<td>$98,274  bonus is: $9,827.40</td>
<td></td>
</tr>
</tbody>
</table>
To specify a new column that contains a character constant, you include the text string in quotation marks in the SELECT clause list. Your modified PROC SQL query is shown below:

```sql
proc sql outobs=15;
title 'Current Bonus Information';
title2 'Employees with Salaries > $75,000';
select empid label='Employee ID',
   jobcode label='Job Code',
   salary,
   'bonus is:',
   salary * .10 format=dollar12.2
from sasuser.payrollmaster
where salary>75000
order by salary desc;
```

In the SELECT clause list, the text string `bonus is:` has been added between `Salary` and `Bonus`. Note that the code `as Bonus` has been removed from the last line of the SELECT clause. Now that the character constant has been added, the column alias `Bonus` is no longer needed.

---

**Summarizing and Grouping Data**

Instead of just listing individual rows, you can use a summary function (also called an aggregate function) to produce a statistical summary of data in a table. For example, in the SELECT clause in the following query, the AVG function calculates the average (or mean) miles traveled by frequent-flyer club members. The GROUP BY clause tells PROC SQL to calculate and display the average for each membership group (`MemberType`).

```sql
proc sql;
   select membertype, avg(milestraveled) as AvgMilesTraveled
   from sasuser.frequentflyers
   group by membertype;
```

You should already be familiar with the list of summary functions that can be used in a PROC SQL query. PROC SQL calculates summary functions and outputs results in different ways depending on a combination of factors. Four key factors are

- whether the summary function specifies one or multiple columns as arguments
- whether the query contains a GROUP BY clause
- if the summary function is specified in a SELECT clause, whether there are additional columns listed that are outside of a summary function
- whether the WHERE clause, if there is one, contains only columns that are specified in the SELECT clause.

To ensure that your PROC SQL queries produce the intended output, it is important to understand how the factors listed above affect the processing of summary functions. Let's look at an overview of all the factors, followed by a detailed example that illustrates each factor.
Number of Arguments and Summary Function Processing

Summary functions specify one or more arguments in parentheses. In the examples shown in this chapter, the arguments are always columns in the table being queried.

Note: The ANSI-standard summary functions, such as AVG and COUNT, can only be used with a single argument. The SAS summary functions, such as MEAN and N, can be used with either single or multiple arguments.

The following chart shows how the number of columns specified as arguments affects the way that PROC SQL calculates a summary function.

<table>
<thead>
<tr>
<th>If a summary function...</th>
<th>Then the calculation is...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>specifies one column as argument</td>
<td>performed down the column</td>
<td><code>proc sql;</code>&lt;br&gt;<code>select avg(salary) as AvgSalary</code>&lt;br&gt;<code>from sasuser.payrollmaster;</code></td>
</tr>
<tr>
<td>specifies multiple columns as arguments</td>
<td>performed across columns for each row</td>
<td><code>proc sql outobs=10;</code>&lt;br&gt;<code>select sum(boarded, transferred, nonrevenue) as Total</code>&lt;br&gt;<code>from sasuser.marchflights;</code></td>
</tr>
</tbody>
</table>

Groups and Summary Function Processing

Summary functions perform calculations on groups of data. When PROC SQL processes a summary function, it looks for a GROUP BY clause:

<table>
<thead>
<tr>
<th>If a GROUP BY clause...</th>
<th>Then PROC SQL...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>is not present in the query</td>
<td>applies the function to the entire table</td>
<td><code>proc sql outobs=10;</code>&lt;br&gt;<code>select jobcode, avg(salary) as AvgSalary</code>&lt;br&gt;<code>from sasuser.payrollmaster;</code></td>
</tr>
<tr>
<td>is present in the query</td>
<td>applies the function to each group specified in the GROUP BY clause</td>
<td><code>proc sql outobs=10;</code>&lt;br&gt;<code>select jobcode, avg(salary) as AvgSalary</code>&lt;br&gt;<code>from sasuser.payrollmaster group by jobcode;</code>&lt;br&gt;&quot;If a query contains a GROUP BY clause, all columns in the SELECT clause that do not contain a summary function should be listed in the GROUP BY clause or unexpected results might be returned.&quot;</td>
</tr>
</tbody>
</table>

SELECT Clause Columns and Summary Function Processing

A SELECT clause that contains a summary function can also list additional columns that are not specified in the summary function. The presence of these additional columns in the SELECT clause list causes PROC SQL to display the output differently.
Performing Advanced Queries Using PROC SQL

Using a Summary Function with a Single Argument (Column)

<table>
<thead>
<tr>
<th>If a SELECT clause...</th>
<th>Then PROC SQL...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains summary function(s) and no columns outside of summary functions</td>
<td>calculates a single value by using the summary function for the entire table or, if groups are specified in the GROUP BY clause, for each group combines or rolls up the information into a single row of output for the entire table or, if groups are specified, for each group</td>
<td><code>proc sql;</code> <code>select avg(salary) as AvgSalary from sasuser.payrollmaster;</code></td>
</tr>
<tr>
<td>contains summary function(s) and additional columns outside of summary functions</td>
<td>calculates a single value for the entire table or, if groups are specified, for each group, and displays all rows of output with the single or grouped value(s) repeated</td>
<td><code>proc sql;</code> <code>select jobcode, gender, avg(salary) as AvgSalary from sasuser.payrollmaster group by jobcode,gender;</code></td>
</tr>
</tbody>
</table>

Note: WHERE clause columns also affect summary function processing. If there is a WHERE clause that references only columns that are specified in the SELECT clause, PROC SQL combines information into a single row of output. However, this condition is not covered in this chapter. For more information, see the SAS documentation for the SQL procedure.

In the next few sections, you will look more closely at the query examples shown above to see how the first three factors impact summary function processing.

Let’s compare two PROC SQL queries that contain a summary function: one with a single argument and the other with multiple arguments. To keep things simple, these queries do not contain a GROUP BY clause.

Using a Summary Function with a Single Argument (Column)

Below is a PROC SQL query that displays the average salary of all employees listed in the table `Sasuser.Payrollmaster`:

```sql
proc sql;
  select avg(salary) as AvgSalary
  from sasuser.payrollmaster;
```

The SELECT statement contains the summary function AVG with `Salary` as its argument. Because there is only one column as an argument, the function calculates the statistic **down the Salary column** to display a single value: the mean salary for all employees. The output is shown here.

<table>
<thead>
<tr>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>54079.62</td>
</tr>
</tbody>
</table>
Using a Summary Function with Multiple Arguments (Columns)

Now let’s look at a PROC SQL query that contains a summary function with multiple columns as arguments. This query calculates the total number of passengers for each flight in March by adding the number of boarded, transferred, and nonrevenue passengers:

```sql
proc sql outobs=10;
    select sum(boarded, transferred, nonrevenue)
        as Total
    from sasuser.marchflights;
```

The SELECT clause contains the summary function SUM with three columns as arguments. Because the function contains multiple arguments, the statistic is calculated across the three columns for each row to produce the following output.

<table>
<thead>
<tr>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
<tr>
<td>196</td>
</tr>
<tr>
<td>167</td>
</tr>
<tr>
<td>222</td>
</tr>
<tr>
<td>185</td>
</tr>
<tr>
<td>163</td>
</tr>
<tr>
<td>119</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>227</td>
</tr>
</tbody>
</table>

*Note:* Without the OUTOBS= option, all rows in the table would be displayed in the output. △

Now let’s see how a PROC SQL query with a summary function is affected by including a GROUP BY clause and including columns outside of a summary function.

Using a Summary Function without a GROUP BY Clause

Once again, here is the PROC SQL query that displays the average salary of all employees listed in the table Sasuser.Payrollmaster. This query contains a summary function but, since the goal is to display the average across all employees, there is no GROUP BY clause.

```sql
proc sql outobs=20;
    select avg(salary) as AvgSalary
        from sasuser.payrollmaster;
```

Note that the SELECT clause lists only one column: a new column that is defined by a summary function calculation. There are no columns listed outside of the summary function.

Here is the query output.
Using a Summary Function with Columns Outside of the Function

Suppose you calculate an average for each job group and group the results by job code. Your first step is to add an existing column (JobCode) to the SELECT clause list. The modified query is shown here:

```
proc sql outobs=20;
    select jobcode, avg(salary) as AvgSalary
    from sasuser.payrollmaster;
```

Let’s see what the query output looks like now that the SELECT statement contains a column (JobCode) that is not a summary function argument.

<table>
<thead>
<tr>
<th>JobCode</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA2</td>
<td>54079.62</td>
</tr>
<tr>
<td>ME2</td>
<td>54079.62</td>
</tr>
<tr>
<td>ME1</td>
<td>54079.62</td>
</tr>
<tr>
<td>FA3</td>
<td>54079.62</td>
</tr>
<tr>
<td>TA3</td>
<td>54079.62</td>
</tr>
<tr>
<td>ME3</td>
<td>54079.62</td>
</tr>
<tr>
<td>SCP</td>
<td>54079.62</td>
</tr>
<tr>
<td>PT2</td>
<td>54079.62</td>
</tr>
<tr>
<td>TA2</td>
<td>54079.62</td>
</tr>
<tr>
<td>TA3</td>
<td>54079.62</td>
</tr>
<tr>
<td>ME1</td>
<td>54079.62</td>
</tr>
<tr>
<td>PT1</td>
<td>54079.62</td>
</tr>
<tr>
<td>SCP</td>
<td>54079.62</td>
</tr>
<tr>
<td>PT1</td>
<td>54079.62</td>
</tr>
<tr>
<td>ME2</td>
<td>54079.62</td>
</tr>
<tr>
<td>ME2</td>
<td>54079.62</td>
</tr>
<tr>
<td>BCK</td>
<td>54079.62</td>
</tr>
<tr>
<td>FA2</td>
<td>54079.62</td>
</tr>
<tr>
<td>SCP</td>
<td>54079.62</td>
</tr>
<tr>
<td>PT2</td>
<td>54079.62</td>
</tr>
</tbody>
</table>

Note: Remember that this PROC SQL query uses the OUTOBS= option to limit the output to 20 rows. Without this limitation, the output of this query would display all 148 rows in the table. △

As this result shows, adding a column to the SELECT clause that is not within a summary function causes PROC SQL to output all rows instead of a single value. To generate this output, PROC SQL

- calculated the average salary down the column as a single value (54079.62)
displayed all rows in the output, because JobCode is not specified in a summary function.

Therefore, the single value for AvgSalary is repeated for each row.

Note: When this query is submitted, the SAS log displays a message indicating that data remerging has occurred. Data remerging is explained later in this chapter.

While this result is interesting, you have not yet reached your goal: grouping the data by JobCode. The next step is to add the GROUP BY clause.

Using a Summary Function with a GROUP BY Clause

Below is the PROC SQL query from the previous page, to which has been added a GROUP BY clause that specifies the column JobCode. (In the SELECT clause, JobCode is specified but is not used as a summary function argument.) Other changes to the query include removing the OUTOBS= option (it is unnecessary) and specifying a format for the AvgSalary column.

```
proc sql;
  select jobcode,
       avg(salary) as AvgSalary format=dollar11.2
  from sasuser.payrollmaster
  group by jobcode;
```

Let’s see how the addition of the GROUP BY clause affects the output.

<table>
<thead>
<tr>
<th>JobCode</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK</td>
<td>$36,111.89</td>
</tr>
<tr>
<td>FA1</td>
<td>$32,255.18</td>
</tr>
<tr>
<td>FA2</td>
<td>$38,131.50</td>
</tr>
<tr>
<td>FA3</td>
<td>$46,107.57</td>
</tr>
<tr>
<td>ME1</td>
<td>$39,900.50</td>
</tr>
<tr>
<td>ME2</td>
<td>$40,807.64</td>
</tr>
<tr>
<td>ME3</td>
<td>$59,374.86</td>
</tr>
<tr>
<td>NA1</td>
<td>$56,845.00</td>
</tr>
<tr>
<td>NA2</td>
<td>$73,336.00</td>
</tr>
<tr>
<td>PT1</td>
<td>$96,071.13</td>
</tr>
<tr>
<td>PT2</td>
<td>$122,253.40</td>
</tr>
<tr>
<td>PT3</td>
<td>$154,706.50</td>
</tr>
<tr>
<td>SCP</td>
<td>$26,632.14</td>
</tr>
<tr>
<td>TA1</td>
<td>$38,809.89</td>
</tr>
<tr>
<td>TA2</td>
<td>$47,004.90</td>
</tr>
<tr>
<td>TA3</td>
<td>$56,551.42</td>
</tr>
</tbody>
</table>

Success! The summary function has been calculated for each JobCode group, and the results are grouped by JobCode.
Counting Values by Using the COUNT Summary Function

Sometimes you want to count the number of rows in an entire table or in groups of rows. In PROC SQL, you can use the COUNT summary function to count the number of rows that have nonmissing values. There are three main ways to use the COUNT function.

<table>
<thead>
<tr>
<th>Using this form of COUNT...</th>
<th>Returns...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(*)</td>
<td>the total number of rows in a group or in a table</td>
<td>select count(*) as Count</td>
</tr>
<tr>
<td>COUNT(column)</td>
<td>the total number of rows in a group or in a table for which there is a nonmissing value in the selected column</td>
<td>select count(jobcode) as Count</td>
</tr>
<tr>
<td>COUNT(DISTINCT column)</td>
<td>the total number of unique values in a column</td>
<td>select count(distinct jobcode) as Count</td>
</tr>
</tbody>
</table>

**CAUTION:**

The COUNT summary function counts only the nonmissing values; missing values are ignored. Many other summary functions also ignore missing values. For example, the AVG function returns the average of the nonmissing values only. When you use a summary function with data that contains missing values, the results might not provide the information that you expect. It is a good idea to familiarize yourself with the data before you use summary functions in queries.

**Tip:** To count the number of missing values, use the NMISS function. For more information about the NMISS function, see the SAS documentation.

Let’s look more closely at the three ways of using the COUNT function.

Counting All Rows

Suppose you want to know how many employees are listed in the table Sasuser.Payrollmaster. This table contains a separate row for each employee, so counting the number of rows in the table gives you the number of employees. The following PROC SQL query accomplishes this task:

```sql
proc sql;
  select count(*) as Count
  from sasuser.payrollmaster;
```

**Note:** The COUNT summary function is the only function that allows you to use an asterisk (*) as an argument.
You can also use COUNT(*) to count rows within groups of data. To do this, you specify the groups in the GROUP BY clause. Let's look at a more complex PROC SQL query that uses COUNT(*) with grouping. This time, the goal is to find the total number of employees within each job category, using the same table that is used above.

```sql
proc sql;
   select substr(jobcode,1,2)
       label='Job Category',
       count(*) as Count
   from sasuser.payrollmaster
   group by 1;
```

This query defines two new columns in the SELECT clause. The first column, which is labeled **Job Category**, is created by using the SAS function SUBSTR to extract the two-character job category from the existing **JobCode** field. The second column, **Count**, is created by using the COUNT function. The GROUP BY clause specifies that the results are to be grouped by the first defined column (referenced by 1 because the column was not assigned a name).

<table>
<thead>
<tr>
<th>Job Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>9</td>
</tr>
<tr>
<td>FA</td>
<td>34</td>
</tr>
<tr>
<td>ME</td>
<td>29</td>
</tr>
<tr>
<td>NA</td>
<td>8</td>
</tr>
<tr>
<td>PT</td>
<td>20</td>
</tr>
<tr>
<td>SC</td>
<td>7</td>
</tr>
<tr>
<td>TA</td>
<td>41</td>
</tr>
</tbody>
</table>

**CAUTION:**

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

---

### Counting All Non-Missing Values in a Column

Suppose you want to count all of the non-missing values in a specific column instead of in the entire table. To do this, you specify the name of the column as an argument of the COUNT function. For example, the following PROC SQL query counts all non-missing values in the column **JobCode**:

```sql
proc sql;
   select count(JobCode) as Count
   from sasuser.payrollmaster;
```

Because the table has no missing data, you get the same output with this query as you would by using COUNT(*). **JobCode** has a non-missing value for each row in the table. However, if the **JobCode** column contained missing values, this query would produce a lower value of **Count** than the previous query. For example, if **JobCode** contained three missing values, the value of **Count** would be 145.
Counting All Unique Values in a Column

To count all unique values in a column, add the keyword `DISTINCT` before the name of the column that is used as an argument. For example, here is the previous query modified to count only the unique values:

```
proc sql;
    select count(distinct jobcode) as Count
    from sasuser.payrollmaster;
```

This query counts 16 unique values for `JobCode`.

<table>
<thead>
<tr>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

To display the unique `JobCode` values, you can apply the method of eliminating duplicates, which was discussed earlier. The following query lists only the unique values for `JobCode`.

```
proc sql;
    select distinct jobcode
    from sasuser.payrollmaster;
```

There are 16 job codes, so the output contains 16 rows.

<table>
<thead>
<tr>
<th>JobCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK</td>
</tr>
<tr>
<td>FA1</td>
</tr>
<tr>
<td>FA2</td>
</tr>
<tr>
<td>FA3</td>
</tr>
<tr>
<td>ME1</td>
</tr>
<tr>
<td>ME2</td>
</tr>
<tr>
<td>ME3</td>
</tr>
<tr>
<td>NA1</td>
</tr>
<tr>
<td>NA2</td>
</tr>
<tr>
<td>PT1</td>
</tr>
<tr>
<td>PT2</td>
</tr>
<tr>
<td>PT3</td>
</tr>
<tr>
<td>SCP</td>
</tr>
<tr>
<td>TA1</td>
</tr>
<tr>
<td>TA2</td>
</tr>
<tr>
<td>TA3</td>
</tr>
</tbody>
</table>
Selecting Groups by Using the HAVING Clause

You have seen how to use the GROUP BY clause to group data. For example, the following query calculates the average salary within each job-code group, and displays the average for each job code:

```sql
proc sql;
  select jobcode,
       avg(salary) as AvgSalary
  format=dollar11.2
  from sasuser.payrollmaster
  group by jobcode;
```

There are 16 job codes in the table, so the output displays 16 rows.

<table>
<thead>
<tr>
<th>JobCode</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK</td>
<td>$36,111.89</td>
</tr>
<tr>
<td>FA1</td>
<td>$32,255.18</td>
</tr>
<tr>
<td>FA2</td>
<td>$39,181.50</td>
</tr>
<tr>
<td>FA3</td>
<td>$46,107.57</td>
</tr>
<tr>
<td>ME1</td>
<td>$39,900.50</td>
</tr>
<tr>
<td>ME2</td>
<td>$49,807.64</td>
</tr>
<tr>
<td>ME3</td>
<td>$59,374.86</td>
</tr>
<tr>
<td>NA1</td>
<td>$53,845.00</td>
</tr>
<tr>
<td>NA2</td>
<td>$73,336.00</td>
</tr>
<tr>
<td>PT1</td>
<td>$95,071.13</td>
</tr>
<tr>
<td>PT2</td>
<td>$122,253.40</td>
</tr>
<tr>
<td>PT3</td>
<td>$154,706.50</td>
</tr>
<tr>
<td>SCP</td>
<td>$25,632.14</td>
</tr>
<tr>
<td>TA1</td>
<td>$38,809.89</td>
</tr>
<tr>
<td>TA2</td>
<td>$47,004.90</td>
</tr>
<tr>
<td>TA3</td>
<td>$55,551.42</td>
</tr>
</tbody>
</table>

Now, suppose you want to select only a subset of groups for your query output. You can use a HAVING clause, following a GROUP BY clause, to select (or filter) the groups to be displayed. The way a HAVING clause affects groups is similar to the way that a WHERE clause affects individual rows. As in a WHERE clause, the HAVING clause contains an expression that is used to subset the data. Any valid SAS expression can be used. When you use a HAVING clause, PROC SQL displays only the groups that satisfy the HAVING expression.

Note: You can use summary functions in a HAVING clause but not in a WHERE clause, because a HAVING clause is used with groups, but a WHERE clause can only be used with individual rows.

Let's modify the query shown above so that it selects only the JobCode groups with an average salary of more than $56,000. The HAVING clause has been added at the end of the query.
proc sql;
  select jobcode,
       avg(salary) as AvgSalary
  format=dollar11.2
  from sasuser.payrollmaster
  group by jobcode
  having avg(salary) > 56000;

Tip: Alternatively, because the average salary is already calculated in the SELECT clause, the HAVING clause could specify the column alias AvgSalary:

having AvgSalary > 56000

Note that you do not have to specify the keyword CALCULATED in a HAVING clause; you would have to specify it in a WHERE clause.

The query output is shown below. This output is smaller than the previous output, because only a subset of the job-code groups is displayed.

<table>
<thead>
<tr>
<th>JobCode</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME3</td>
<td>$59,374.86</td>
</tr>
<tr>
<td>NA1</td>
<td>$58,845.00</td>
</tr>
<tr>
<td>NA2</td>
<td>$73,336.00</td>
</tr>
<tr>
<td>PT1</td>
<td>$95,071.13</td>
</tr>
<tr>
<td>PT2</td>
<td>$122,253.40</td>
</tr>
<tr>
<td>PT3</td>
<td>$154,706.50</td>
</tr>
</tbody>
</table>

If you omit the GROUP BY clause in a query that contains a HAVING clause, then the HAVING clause and summary functions (if any are specified) treat the entire table as one group. Without a GROUP BY clause, the HAVING clause in the example shown above calculates the average salary for the table as a whole (all jobs in the company), not for each group (each job code). The output contains either all the rows in the table (if the average salary for the entire table is greater than $56,000) or none of the rows in the table (if the average salary for the entire table is less than $56,000).

**Understanding Data Remerging**

Sometimes, when you use a summary function in a SELECT clause or a HAVING clause, PROC SQL must remerge data (make two passes through the table). Remerging requires additional processing time and is often unavoidable. However, there are some situations in which you may be able to modify your query to avoid remerging. Understanding how and when remerging occurs will increase your ability to write efficient queries.

Let’s look at a PROC SQL query that requires remerging. This query calculates each navigator’s salary as a percentage of all navigators’ salaries:

proc sql;
  select empid, salary,
       (salary/sum(salary)) as Percent
  format=percent8.2
  from sasuser.payrollmaster
  where jobcode contains ‘NA’;

When you submit this query, the SAS log displays the following message.
Remerging occurs whenever any of the following conditions exist:

- The values returned by a summary function are used in a calculation.
- The SELECT clause specifies a column that contains a summary function and other column(s) that are not listed in a GROUP BY clause.
- The HAVING clause specifies one or more columns or column expressions that are not included in a subquery or a GROUP BY clause.

During remerging, PROC SQL makes two passes through the table:

1. PROC SQL calculates and returns the value of summary functions. PROC SQL also groups data according to the GROUP BY clause.
2. PROC SQL retrieves any additional columns and rows that it needs to display in the output, and uses the result from the summary function to calculate any arithmetic expressions in which the summary function participates.

Example

Now let's see how PROC SQL remerges data when it processes the following query:

```sql
proc sql;
    select empid, salary,
        (salary/sum(salary)) as Percent
    format=percent8.2
    from sasuser.payrollmaster
    where jobcode contains 'NA';
```

In the first pass, PROC SQL calculates and returns the value of the SUM function (specified in the SELECT clause).

In the second pass, PROC SQL retrieves the additional columns and rows that it needs to display in output (EmpID, Salary) and the rows in which JobCode contains 'NA'. PROC SQL also uses the result from the SUM function to calculate the arithmetic expression (salary/sum(salary)).

CAUTION:

Some implementations of SQL do not support remerging and would consider the preceding example to be in error.△

Tip: You can obtain the same results by using a subquery. Subqueries are discussed later in this chapter.

Subsetting Data by Using Subqueries

Introducing Subqueries

The WHERE and HAVING clauses both subset data based on an expression. In the query examples shown earlier in this chapter, the WHERE and HAVING clauses contained standard SAS expressions. For example, the expression in the following
WHERE clause uses the BETWEEN-AND conditional operator and specifies the **Salary** column as an operand:

```sql
WHERE salary between 70000 and 80000
```

PROC SQL also offers another type of expression that can be used for subsetting in **WHERE** and **HAVING** clauses: a query-expression or **subquery**. A subquery is a query that is nested in, and is part of, another query. A PROC SQL query may contain subqueries at one or more levels.

*Note:* Subqueries are also known as nested queries, inner queries, and sub-selects.

The following PROC SQL query contains a subquery in the **HAVING** clause:

```sql
proc sql;
select jobcode,
    avg(salary) as AvgSalary
format=dollar11.2
from sasuser.payrollmaster
group by jobcode
having avg(salary) >
    (select avg(salary)
     from sasuser.payrollmaster);
```

*Tip:* It is recommended that you enclose a subquery (inner query) in parentheses, as shown here.

A subquery selects one or more rows from a table, then returns single or multiple values to be used by the outer query. The subquery shown above is a **single-value subquery**; it returns a single value, the average salary from the table `Sasuser.Payrollmaster`, to the outer query. A subquery can return values for **multiple rows** but only for a **single column**.

The table that a subquery references can be either the same as or different from the table referenced by the outer query. In the PROC SQL query shown above, the subquery selects data from the same table as the outer query.

---

### Types of Subqueries

There are two types of subqueries.

<table>
<thead>
<tr>
<th>Type of Subquery</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>noncorrelated</td>
<td>a self-contained subquery that <em>executes independently of the outer query</em></td>
</tr>
<tr>
<td>correlated</td>
<td>a dependent subquery that <em>requires one or more values to be passed to it by the outer query before the subquery can return a value to the outer query</em></td>
</tr>
</tbody>
</table>

Both noncorrelated and correlated subqueries can return either single or multiple values to the outer query.

The next few sections provide a more in-depth look at noncorrelated and correlated subqueries, and how they are processed.
Subsetting Data by Using Noncorrelated Subqueries

A noncorrelated subquery is a self-contained subquery that executes independently of the outer query.

Using Single-Value Noncorrelated Subqueries

The simplest type of subquery is a noncorrelated subquery that returns a single value. The following PROC SQL query is the same query that is used in the previous section. This query displays job codes for which the group's average salary exceeds the company's average salary. The HAVING clause contains a noncorrelated subquery.

```sql
proc sql;
  select jobcode,
        avg(salary) as AvgSalary
  format=dollar11.2
  from sasuser.payrollmaster
  group by jobcode
  having avg(salary) >
    (select avg(salary)
     from sasuser.payrollmaster);
```

PROC SQL always evaluates a noncorrelated subquery before the outer query. If a query contains noncorrelated subqueries at more than one level, PROC SQL evaluates the innermost subquery first and works outward, evaluating the outermost query last. In the query shown above, the inner query and outer query are processed as follows:

1. To complete the expression in the HAVING clause, the subquery calculates the average salary for the entire company (all rows in the table), using the AVG summary function with `Salary` as an argument.
2. The subquery returns the value of the average salary to the outer query.
3. The outer query calculates the average salary (in the SELECT clause) for each `JobCode` group (as defined in the GROUP BY clause), and selects only the groups whose average salary is greater than the company's average salary.

The query output is shown here.

<table>
<thead>
<tr>
<th>JobCode</th>
<th>AvgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME3</td>
<td>$59,374.86</td>
</tr>
<tr>
<td>NA1</td>
<td>$59,845.00</td>
</tr>
<tr>
<td>NA2</td>
<td>$73,336.00</td>
</tr>
<tr>
<td>PT1</td>
<td>$95,071.13</td>
</tr>
<tr>
<td>PT2</td>
<td>$122,253.40</td>
</tr>
<tr>
<td>PT3</td>
<td>$154,706.50</td>
</tr>
<tr>
<td>TA3</td>
<td>$55,551.42</td>
</tr>
</tbody>
</table>

This noncorrelated subquery returns only a single value, the average salary for the whole company, to the outer query. Both the subquery and the outer query use the same table as a source.
Using Multiple-Value Noncorrelated Subqueries

Some subqueries are multiple-value subqueries: they return more than one value (row) to the outer query. If your noncorrelated subquery might return a value for more than one row, be sure to use one of the following operators in the WHERE or HAVING clause that can handle multiple values:

- the conditional operator IN
- a comparison operator that is modified by ANY or ALL
- the conditional operator EXISTS.

CAUTION:

If you create a noncorrelated subquery that returns multiple values, but the WHERE or HAVING clause in the outer query contains an operator other than one of the operators that are specified above, the query will fail. An error message is displayed in the SAS log, which indicates that the subquery evaluated to more than one row. For example, if you use the equal (=) operator with a noncorrelated subquery that returns multiple values, the query will fail. The equal operator can handle only a single value.

Let's first look at a query that contains both the conditional operator IN and a noncorrelated subquery that returns multiple values. (The operators ANY, ALL, and EXISTS are presented later in this chapter.)

Example

Suppose you want to send birthday cards to employees who have birthdays coming up. You decide to create a PROC SQL query that will list the names and addresses of all employees who have birthdays in February. This query, unlike the one shown on the previous page, will select data from two different tables:

- employee names and addresses in the table Sasuser.Staffmaster
- employee birth dates in the table Sasuser.Payrollmaster.

In both tables, the employees are identified by their employee identification number (EmpID).

In the following PROC SQL query, the WHERE clause contains the conditional operator IN followed by a noncorrelated subquery:

```sql
proc sql;
  select empid, lastname, firstname, city, state
  from sasuser.staffmaster
  where empid in
    (select empid
     from sasuser.payrollmaster
     where month(dateofbirth)=2);
```

This query is processed as follows:

1. To complete the expression in the WHERE clause of the outer query, the subquery selects the employees whose date of birth is February. Note that the MONTH function is used in the subquery.
2. The subquery then returns the EmpID values of the selected employees to the outer query.
3. The outer query displays data (from the columns identified in the SELECT clause) for the employees identified by the subquery.
The output, shown below, lists the six employees who have February birthdays.

<table>
<thead>
<tr>
<th>EmpID</th>
<th>LastName</th>
<th>FirstName</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1403</td>
<td>BOWDEN</td>
<td>EARL</td>
<td>BRIDGEPORT</td>
<td>CT</td>
</tr>
<tr>
<td>1404</td>
<td>CARTER</td>
<td>DONALD</td>
<td>NEW YORK</td>
<td>NY</td>
</tr>
<tr>
<td>1834</td>
<td>LONG</td>
<td>RUSSELL</td>
<td>NEW YORK</td>
<td>NY</td>
</tr>
<tr>
<td>1103</td>
<td>MCDANIEL</td>
<td>RONDA</td>
<td>NEW YORK</td>
<td>NY</td>
</tr>
<tr>
<td>1420</td>
<td>ROUSE</td>
<td>JEREMY</td>
<td>PATERSON</td>
<td>NJ</td>
</tr>
<tr>
<td>1390</td>
<td>SMART</td>
<td>JONATHAN</td>
<td>NEW YORK</td>
<td>NY</td>
</tr>
</tbody>
</table>

**Using Comparisons with Subqueries**

Sometimes it is helpful to compare a value with a set of values returned by a subquery. When a subquery might return multiple values, you must use one of the conditional operators ANY or ALL to modify a comparison operator in the WHERE or HAVING clause immediately before the subquery. For example, the following WHERE clause contains the less than (\(<\)) comparison operator and the conditional operator ANY:

```sql
where dateofbirth < any <subquery...>
```

**CAUTION:**

If you create a noncorrelated subquery that returns multiple values, and if the WHERE or HAVING clause in the outer query contains a comparison operator that is not modified by ANY or ALL, the query will fail. Δ

When the outer query contains a comparison operator that is modified by ANY or ALL, the outer query compares each value that it retrieves against the value(s) returned by the subquery. All values for which the comparison is true are then included in the query output. 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 ALL is specified, then the comparison is true only if it is true for all values that are returned by the subquery.

**Note:** The operators ANY and ALL can be used with correlated subqueries, but they are usually used only with noncorrelated subqueries. Δ

Let's take a closer look at how the operators ANY or ALL are used.

**Using the ANY Operator**

An outer query that specifies the ANY operator selects values that pass the comparison test with any of the values that are returned by the subquery.

For example, suppose you have an outer query containing the following WHERE clause:

```sql
where dateofbirth < any <subquery...>
```

This WHERE clause specifies that `DateofBirth` (the operand) should be less than `any` (the comparison operator) of the values returned by the subquery.

The following chart shows the effect of using ANY with these common comparison operators: greater than (\(>\)), less than (\(<\)) and equal to (\(=\)).
Comparison Operator with ANY | Outer Query Selects... | Example
--- | --- | ---
\( > \) ANY | values that are greater than any value returned by the subquery | If the subquery returns the values 20, 30, 40, then the outer query selects all values that are \( > 20 \) (the lowest value that was returned by the subquery).

\( < \) ANY | values that are less than any value returned by the subquery | If the subquery returns the values 20, 30, 40, then the outer query selects all values that are \( < 40 \) (the highest value that was returned by the subquery).

\( = \) ANY | values that are equal to any value returned by the subquery | If the subquery returns the values 20, 30, 40, the outer query selects all values that are \( = 20 \) or \( = 30 \) or \( = 40 \).

Tip: Instead of using the ANY operator with a subquery, there are some SAS functions that you can use to achieve the same result with greater efficiency. Instead of \( > \) ANY, use the MIN function in the subquery. Instead of \( < \) ANY, use the MAX function in the subquery.

Example

Suppose you want to identify any flight attendants at level 1 or level 2 who are older than any of the flight attendants at level 3. Job type and level are identified in JobCode; each flight attendant has the job code FA1, FA2, or FA3. The following PROC SQL query accomplishes this task by using a subquery and the ANY operator:

```sql
proc sql;
   select empid, jobcode, dateofbirth
   from sasuser.payrollmaster
   where jobcode in ('FA1','FA2')
   and dateofbirth < any
   (select dateofbirth
    from sasuser.payrollmaster
    where jobcode='FA3');
```

Here is what happens when this query is processed:

1. The subquery returns the birth dates of all level-3 flight attendants.
2. The outer query selects only those level-1 and level-2 flight attendants whose birth date is \( < \) any of the dates returned by the subquery.

Note that both the outer query and subquery use the same table.

Note: Internally, SAS represents a date value as the number of days from January 1, 1960, to the given date. For example, the SAS date for 17 October 1991 is 11612. Representing dates as the number of days from a reference date makes it easy for the computer to store them and perform calendar calculations. These numbers are not meaningful to users, however, so several different formats are available for displaying dates and datetime values in most of the commonly used notations.

Below are the query results.
Using the ALL Operator

Tip: Using the ANY operator to solve this problem results in a large number of calculations, which increases processing time. For this example, it would be more efficient to use the MAX function in the subquery. The alternative WHERE clause follows:

```sql
where jobcode in ('FA1','FA2')
    and dateofbirth <
        (select max(dateofbirth)
             from [...]
```

For more information about the MAX function, see the SAS documentation.

Using the ALL Operator

An outer query that specifies the ALL operator selects values that pass the comparison test with all of the values that are returned by the subquery.

The following chart shows the effect of using ALL with these common comparison operators: greater than (>\) and less than (<\).

<table>
<thead>
<tr>
<th>Comparison Operator with ALL</th>
<th>Sample Values Returned by Subquery</th>
<th>Signifies...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; ALL</td>
<td>(20, 30, 40)</td>
<td>&gt; 40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(greater than the highest number in the list)</td>
</tr>
<tr>
<td>&lt; ALL</td>
<td>(20, 30, 40)</td>
<td>&lt; 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(less than the lowest number in the list)</td>
</tr>
</tbody>
</table>
Example

Let’s take the previous query example and substitute ALL for ANY. The following query identifies level-1 and level-2 flight attendants who are older than all of the level-3 flight attendants:

```
proc sql;
  select empid, jobcode, dateofbirth
  from sasuser.payrollmaster
  where jobcode in ('FA1','FA2')
    and dateofbirth < all
    (select dateofbirth
     from sasuser.payrollmaster
     where jobcode='FA3');
```

Here is what happens when this query is processed:

1. The subquery returns the birth dates of all level-3 flight attendants.
2. The outer query selects only those level-1 and level-2 flight attendants whose birth date is less than all of the dates returned by the subquery.

The query results, below, show that only two level-1 or level-2 flight attendants are older than all of the level-3 flight attendants.

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>DateOfBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1124</td>
<td>FA1</td>
<td>13JUL1956</td>
</tr>
<tr>
<td>1415</td>
<td>FA2</td>
<td>12MAR1956</td>
</tr>
</tbody>
</table>

Tip: For this example, it would be more efficient to solve this problem using the MIN function in the subquery instead of the ALL operator. The alternative WHERE clause follows:

```
where jobcode in ('FA1','FA2')
  and dateofbirth <
    (select min(dateofbirth)
     from [...])
```

For more information about the MIN function, see the SAS documentation.

Subsetting Data by Using Correlated Subqueries

Correlated subqueries cannot be evaluated independently, but depend on the values passed to them by the outer query for their results. Correlated subqueries are evaluated for each row in the outer query and, therefore, tend to require more processing time than noncorrelated subqueries.

Note: Usually, a PROC SQL join is a more efficient alternative to a correlated subquery. You should already be familiar with basic PROC SQL joins.

Example

Let’s look at an example of a PROC SQL query that contains a correlated subquery. The following query displays the names of all navigators who are also managers.
correlated subquery is shown in bold. The WHERE clause in the subquery lists the column `Staffmaster.EmpID`, which is the column that the outer query must pass to the correlated subquery.

```sql
proc sql;
    select lastname, firstname
    from sasuser.staffmaster
    where 'NA'=
        (select jobcategory
            from sasuser.supervisors
            where staffmaster.empid = supervisors.empid);
```

**Note:** When a column appears in more than one table, the column name is preceded with the table name or alias to avoid ambiguity. In this example, `EmpID` appears in both tables, so the appropriate table name is specified in front of each reference to that column.

The output from this query is shown below. There are three navigators who are also managers.

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>FERNANDEZ</td>
<td>KATRINA</td>
</tr>
<tr>
<td>NEWKIRK</td>
<td>WILLIAM</td>
</tr>
<tr>
<td>RIVERS</td>
<td>SIMON</td>
</tr>
</tbody>
</table>

**Using the EXISTS and NOT EXISTS Conditional Operators**

In the WHERE clause or in the HAVING clause of an outer query, you can use the EXISTS or NOT EXISTS conditional operator to test for the existence or non-existence of a set of values returned by a subquery.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Is true if...</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXISTS</td>
<td>the subquery returns at least one row</td>
</tr>
<tr>
<td>NOT EXISTS</td>
<td>the subquery returns no data</td>
</tr>
</tbody>
</table>

**Note:** The operators EXISTS and NOT EXISTS can be used with both correlated and noncorrelated subqueries.

**Example: Correlated Subquery with NOT EXISTS**

Let's look at a sample PROC SQL query that includes the NOT EXISTS conditional operator. Suppose you are working with the following tables:

- `Sasuser.Flightattendants` contains the names and employee ID numbers of all flight attendants.
- `Sasuser.Flightschedule` contains one row for each crew member assigned to a flight for each date.
Performing Advanced Queries Using PROC SQL

As shown in the diagram below, the intersection of these two tables contains data for all flight attendants who have been scheduled to work.

![Diagram of flight attendants scheduled to work compared to employees scheduled to work]

Now suppose you want to list by name the flight attendants who have not been scheduled; that is, you want to identify the data in the area highlighted below.

![Diagram showing flight attendants not scheduled to work]

The following PROC SQL query accomplishes this task by using a correlated subquery and the NOT EXISTS operator:

```sql
proc sql;
select lastname, firstname
from sasuser.flightattendants
where not exists
 (select *
  from sasuser.flightschedule
  where flightattendants.empid=
    flightschedule.empid);
```

The output is shown below.

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATTERSON</td>
<td>RENEE</td>
</tr>
<tr>
<td>VEGA</td>
<td>FRANKLIN</td>
</tr>
</tbody>
</table>

Validating Query Syntax

When you are building a PROC SQL query, you might find it more efficient to check your query without actually executing it. To verify the syntax and the existence of
columns and tables that are referenced in the query without executing the query, use either of the following:

- the NOEXEC option in the PROC SQL statement
- the VALIDATE keyword before a SELECT statement.

Let’s look at how you specify the NOEXEC option and the VALIDATE keyword, and examine the minor differences between them.

---

**Using the NOEXEC Option**

The NOEXEC option is specified in the following PROC SQL statement:

```sql
proc sql noexec;
    select empid, jobcode, salary
    from sasuser.payrollmaster
    where jobcode contains 'NA'
    order by salary;
```

If the query is valid and all referenced columns and tables exist, the SAS log displays the following message.

<table>
<thead>
<tr>
<th>Table 2.5</th>
<th>SAS Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTE:</strong> Statement not executed due to NOEXEC option.</td>
<td></td>
</tr>
</tbody>
</table>

Or, if there are any errors in the query, SAS displays the standard error messages in the log.

When you invoke the NOEXEC option, SAS checks the syntax of all queries in that PROC SQL step for accuracy but does not execute them.

---

**Using the VALIDATE Keyword**

You specify the VALIDATE keyword just before a SELECT statement; it is not used with any other PROC SQL statement.

Let’s modify the preceding PROC SQL query by using the VALIDATE keyword instead of the NOEXEC option:

```sql
proc sql;
    validate
    select empid, jobcode, salary
    from sasuser.payrollmaster
    where jobcode contains 'NA'
    order by salary;
```

*Note:* Note that the VALIDATE keyword is *not* followed by a semicolon.

If the query is valid, the SAS log displays the following message.

<table>
<thead>
<tr>
<th>Table 2.6</th>
<th>SAS Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTE:</strong> PROC SQL statement has valid syntax.</td>
<td></td>
</tr>
</tbody>
</table>

If there are errors in the query, SAS displays the standard error messages in the log.

The main difference between the VALIDATE keyword and the NOEXEC option is that the VALIDATE keyword only affects the SELECT statement that immediately
follows it, whereas the NOEXEC option applies to all queries in the PROC SQL step. If you are working with a PROC SQL query that contains multiple SELECT statements, the VALIDATE keyword must be specified before each SELECT statement that you want to check.

### Additional Features

In addition to the SELECT statement, PROC SQL supports the following statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Use to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER TABLE expression;</td>
<td>add, drop, and modify columns in a table</td>
</tr>
<tr>
<td>CREATE expression;</td>
<td>build new tables, views, or indexes</td>
</tr>
<tr>
<td>DELETE expression;</td>
<td>eliminate unwanted rows from a table or view</td>
</tr>
<tr>
<td>DESCRIBE expression;</td>
<td>display table and view attributes</td>
</tr>
<tr>
<td>DROP expression;</td>
<td>eliminate entire tables, views, or indexes</td>
</tr>
<tr>
<td>INSERT expression;</td>
<td>add rows of data to tables or views</td>
</tr>
<tr>
<td>RESET &lt;option(s)&gt;</td>
<td>add to or change PROC SQL options without re-invoking the procedure</td>
</tr>
<tr>
<td>UPDATE expression;</td>
<td>modify data values in existing rows of a table or view</td>
</tr>
</tbody>
</table>

You can learn more about these PROC SQL statements in the following chapters:

- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
- Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125
- Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159
- Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221
- Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243
- Chapter 8, “Managing Processing Using PROC SQL,” on page 261.
This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

### Text Summary

#### Viewing SELECT Statement Syntax

The SELECT statement and its subordinate clauses are the building blocks that you use to construct all PROC SQL queries.

#### Displaying All Columns

To display all columns in the order in which they are stored in the table, use an asterisk (*) in the SELECT clause. To write the expanded list of columns to the SAS log, use the FEEDBACK option in the PROC SQL statement.

#### Limiting the Number of Rows Displayed

To limit the number of rows that PROC SQL displays as output, use the OUTOBS=n option in the PROC SQL statement.

#### Eliminating Duplicate Rows from Output

To eliminate duplicate rows from your query results, use the keyword DISTINCT in the SELECT clause.

#### Subsetting Rows by Using Conditional Operators

In a PROC SQL query, use the WHERE clause with any valid SAS expression to subset data. The SAS expression can contain one or more operators, including the following conditional operators:

- the BETWEEN-AND operator selects within an inclusive range of values
- the CONTAINS or ? operator selects a character string
- the IN operator selects from a list of fixed values
- the IS MISSING or IS NULL operator selects missing values
- the LIKE operator selects a pattern
- the sounds-like (=*) operator selects a spelling variation

#### Subsetting Rows by Using Calculated Values

It is important to understand how PROC SQL processes calculated columns. When you use a column alias in the WHERE or the HAVING clause to refer to a calculated value, you must also use the keyword CALCULATED along with the alias.
**Enhancing Query Output**

You can enhance PROC SQL query output by using SAS enhancements such as column formats and labels, titles and footnotes, and character constraints.

**Summarizing and Grouping Data**

PROC SQL calculates summary functions and outputs results differently, depending on a combination of factors:
- whether the summary function specifies one or more multiple columns as arguments
- whether the query contains a GROUP BY clause
- if the summary function is specified in a SELECT clause, whether there are additional columns listed that are outside the summary function
- whether the WHERE clause, if there is one, contains only columns that are specified in the SELECT clause.

To count non-missing values, use the COUNT summary function.
To select the groups to be displayed, use a HAVING clause following a GROUP BY clause.
When you use a summary function in a SELECT clause or a HAVING clause, in some situations, PROC SQL must remerge data. When PROC SQL remerges data, it makes two passes through the data, and this requires additional processing time.

**Subsetting Data by Using Subqueries**

In the WHERE or the HAVING clause of a PROC SQL query, you can use a subquery to subset data. A subquery is a query that is nested in, and is part of, another query. Subqueries can return values from a single row or multiple rows to the outer query but can return values only from a single column.

**Subsetting Data by Using Noncorrelated Subqueries**

Noncorrelated subqueries execute independently of the outer query. You can use noncorrelated subqueries that return a single value or multiple values. To further qualify a comparison specified in a WHERE or a HAVING clause, you can use the conditional operators ANY and ALL immediately before a noncorrelated (or correlated) subquery.

**Subsetting Data by Using Correlated Subqueries**

Correlated subqueries cannot be evaluated independently because their results are dependant on the values returned by the outer query. In the WHERE or the HAVING clause of an outer query, you can use the EXISTS and NOT EXISTS conditional operators to test for the existence or non-existence of a set of values returned by the subquery.

**Validating Query Syntax**

To check the validity of the query syntax without actually executing the query, use the NOEXEC option or the VALIDATE keyword.

**Additional Features**

PROC SQL supports many statements in addition to the SELECT statement.
Syntax

PROC SQL OUTOBS=n;
    SELECT column-1,... column-n
    FROM table-1 | view-1... table-n | view-n
    <WHERE expression>
    <GROUP BY column-1,... column-n>
    <HAVING expression>
    <ORDER BY column-1,... column-n>
QUIT;

BETWEEN value-1 AND value-2
sql-expression CONTAINS sql-expression
column IN (constant-1 <,...constant-n>)
column IS MISSING
column IS NULL
column LIKE ‘pattern’
sql-expression =* sql-expression

Sample Programs

Displaying all Columns in Output and an Expanded Column List in the SAS Log

proc sql feedback;
    select *
    from sasuser.staffchanges;
quit;

Eliminating Duplicate Rows from Output

proc sql;
    select distinct flightnumber, destination
    from sasuser.internationalflights
    order by 1;
quit;

Subsetting Rows by Using Calculated Values

proc sql outobs=10;
    validate
    select flightnumber,
        date label="Flight Date", destination,
        boarded + transferred + nonrevenue
    as Total
    from sasuser.marchflights
    where calculated total between 100 and 150;
quit;
**Subsetting Data by Using a Noncorrelated Subquery**

```sql
proc sql noexec;
  select jobcode,
       avg(salary) as AvgSalary
  format=dollar11.2
  from sasuser.payrollmaster
  group by jobcode
  having avg(salary) >
       (select avg(salary)
        from sasuser.payrollmaster);
quit;
```

**Subsetting Data by Using a Correlated Subquery**

```sql
proc sql;
  title 'Frequent Flyers Who Are Not Employees';
  select count(*) as Count
  from sasuser.frequentflyers
  where not exists
    (select *
     from sasuser.staffmaster
     where name=
       trim(lastname)||', '||firstname);
quit;
```

**Points to Remember**

- When you use summary functions, look for missing values. If a table contains missing values, your results might not be what you expect. Many summary functions ignore missing values when performing calculations, and PROC SQL treats missing values in a column as a single group.
- When you create complex queries, it is helpful to use the NOEXEC option or the VALIDATE statement to validate your query without executing it.

**Quiz**

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which PROC SQL query will remove duplicate values of `MemberType` from the query output, so that only the unique values are listed?
   
a. `proc sql nodup;
      select membertype
      from sasuser.frequentflyers;
   
b. `proc sql;
      select distinct(membertype)
        as MemberType
      from sasuser.frequentflyers;
   
c. `proc sql;
      select unique membertype
      from sasuser.frequentflyers
      group by membertype;`
d  proc sql;
    select distinct membertype
    from sasuser.frequentflyers;

2  Which of the following will cause PROC SQL to list rows that have no data in the Address column?
   a  WHERE address is missing
   b  WHERE address not exists
   c  WHERE address is null
   d  both a and c

3  You are creating a PROC SQL query that will list all employees who have spent (or overspent) their allotted 120 hours of vacation for the current year. The hours that each employee used are stored in the existing column Spent. Your query defines a new column, Balance, to calculate each employee’s balance of vacation hours.
   Which query will produce the report that you want?
   a  proc sql;
      select name, spent,
          120-spent as calculated Balance
      from Company.Absences
      where balance <= 0;
   b  proc sql;
      select name, spent,
          120-spent as Balance
      from Company.Absences
      where calculated balance <= 0;
   c  proc sql;
      select name, spent,
          120-spent as Balance
      from Company.Absences
      where balance <= 0;
   d  proc sql;
      select name, spent,
          120-spent as calculated Balance
      from Company.Absences
      where calculated balance <= 0;

4  Consider this PROC SQL query:
   proc sql;
      select flightnumber,
          count(*) as Flights,
          avg( boarded )
          label="Average Boarded"
          format=3.
      from sasuser.internationalflights
      group by flightnumber
      having avg( boarded ) > 150;
The table `Sasuser.Internationalflights` contains 201 rows, 7 unique values of `FlightNumber`, 115 unique values of `Boarded`, and 4 different flight numbers that have an average value of `Boarded` that is greater than 150. How many rows of output will the query generate?

- **a** 150
- **b** 7
- **c** 4
- **d** 1

5 You are writing a PROC SQL query that will display the names of all library cardholders who work as volunteers for the library, and the number of books that each volunteer currently has checked out. You will use one or both of the following tables:

- **Library.Circulation** lists the name and contact information for all library cardholders, and the number of books that each cardholder currently has checked out.
- **Library.Volunteers** lists the name and contact information for all library volunteers.

Assume that the values of `Name` are unique in both tables.

Which of the following PROC SQL queries will produce your report?

- **a** proc sql;
  select name, checkedout
  from library.circulation
  where * in
  (select *
   from library.volunteers);

- **b** proc sql;
  select name, checkedout
  from library.circulation
  where name in
  (select name
   from library.volunteers);

- **c** proc sql;
  select name
  from library.volunteers
  where name, checkedout in
  (select name, checkedout
   from library.circulation);

- **d** proc sql;
  select name, checkedout
  from library.circulation
  where name in
  (select name
   from library.volunteers);

6 By definition, a noncorrelated subquery is a nested query that

- **a** returns a single value to the outer query.
- **b** contains at least one summary function.
- **c** executes independently of the outer query.
- **d** requires only a single value to be passed to it by the outer query.
7 Which statement about the following PROC SQL query is false?

```sql
proc sql;
validate
   select name label='Country',
   rate label='Literacy Rate'
from world.literacy
where 'Asia' =
   (select continent
    from world.continents
    where literacy.name =
      continents.country)
order by 2;
```

a. The query syntax is not valid.
b. The outer query must pass values to the subquery before the subquery can return values to the outer query.
c. PROC SQL will not execute this query when it is submitted.
d. After the query is submitted, the SAS log will indicate whether the query has valid syntax.

8 Consider the following PROC SQL query:

```sql
proc sql;
   select lastname, firstname,
   total, since
from charity.donors
where not exists
   (select lastname
    from charity.current
    where donors.lastname =
      current.lastname);
```

The query references two tables:
- **Charity.Donors** lists name and contact information for all donors who have made contributions since the charity was founded. The table also contains these two columns: **Total**, which shows the total dollars given by each donor, and **Since**, which stores the first year in which each donor gave money.
- **Charity.Current** lists the names of all donors who have made contributions in the current year, and the total dollars each has given this year (**YearTotal**).

Assume that the values of **LastName** are unique in both tables.

The output of this query displays

a. all donors whose rows do not contain any missing values.
b. all donors who made a contribution in the current year.
c. all donors who did not make a contribution in the current year.
d. all donors whose current year's donation in **Charity.Current** has not yet been added to **Total** in **Charity.Donors**.

9 Which statement about data remerging is true?

a. When PROC SQL remerges data, it combines data from two tables.
b. By using data remerging, PROC SQL can avoid making two passes through the data.
c. When PROC SQL remerges data, it displays a related message in the SAS log.
d. PROC SQL does not attempt to remerge data unless a subquery is used.
A public library has several categories of books. Each book in the library is assigned to only one category. The table `Library.Inventory` contains one row for each book in the library. The `Checkouts` column indicates the number of times that each book has been checked out.

You want to display only the categories that have an average circulation (number of checkouts) that is less than 2500. Does the following PROC SQL query produce the results that you want?

```
proc sql;
title 'Categories with Average Circulation';
title2 'Less Than 2500';
select category,
    avg(checkouts) as AvgCheckouts
from library.inventory
having avg(checkouts) < 2500
order by 1;
```

a No. This query will not run because a HAVING clause cannot contain a summary function.
b No. This query will not run because the HAVING clause must include the CALCULATED keyword before the summary function.
c No. Because there is no GROUP BY clause, the HAVING clause treats the entire table as one group.
d Yes.
CHAPTER 3

Combining Tables Horizontally Using PROC SQL

Overview 80
  Introduction 80
  Objectives 80
  Prerequisites 81
Understanding Joins 81
Generating a Cartesian Product 81
Using Inner Joins 83
  Introducing Inner Join Syntax 83
  Example 84
  Understanding How Joins Are Processed 85
  Eliminating Duplicate Columns 85
  Renaming a Column by Using a Column Alias 86
  Joining Tables That Have Rows with Matching Values 87
  Specifying a Table Alias 87
  Example: Complex PROC SQL Inner Join 88
  Example: PROC SQL Inner Join with Summary Functions 90
Using Outer Joins 91
  Introducing Types of Outer Joins 91
  Using a Left Outer Join 93
  Using a Right Outer Join 94
  Using a Full Outer Join 94
  Example: Outer Join 95
Creating an Inner Join with Outer Join-Style Syntax 97
Comparing SQL Joins and DATA Step Match-Merges 97
  When All of the Values Match 98
  When Only Some of the Values Match 99
  When Only Some of the Values Match: Using the COALESCE Function 100
Understanding the Advantages of PROC SQL Joins 101
Using In-Line Views 102
  Introducing In-Line Views 102
  Referencing an In-Line View with Other Views or Tables 103
  Referencing Multiple Tables in an In-Line View 103
  Assigning an Alias to an In-Line View 104
  Example: Query That Contains an In-Line View 104
Joining Multiple Tables and Views 106
  Example: Complex Query That Combines Four Tables 106
Example: Technique 1 (PROC SQL Subqueries, Joins, and In-Line Views) 107
  Query 1: Identify the crew for the Copenhagen (CPH) flight 107
  Query 2: Find the states and job categories of the crew members 108
  Query 3: Find the employee numbers of the crew supervisors 108
  Query 4: Find the names of the supervisors 109
Introduction

When you need to select data from multiple tables and combine the tables horizontally (side by side), PROC SQL can be an efficient alternative to other SAS procedures or the DATA step. You can use a PROC SQL join to combine tables horizontally:

```
proc sql;
  select * from a, b
  where a.x=b.x;
```

A PROC SQL join is a query that specifies multiple tables and/or views to be combined and, typically, specifies the conditions on which rows are matched and returned in the result set.

You should already be familiar with the basics of using PROC SQL to join tables. In this chapter, you will take a more in-depth look at joining tables.

Objectives

In this chapter, you learn to

- combine tables horizontally to produce a Cartesian product
- combine tables horizontally using inner and outer joins
- use a table alias in a PROC SQL query
- use an in-line view in a PROC SQL query
Prerequisites

Before you begin this chapter, you should complete the following chapters:
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3

Understanding Joins

Joins combine tables horizontally (side by side) by combining rows. The tables being joined are not required to have the same number of rows or columns.

Note: You can use a join to combine views as well as tables. Most of the following references to tables are also applicable to views; any exceptions are noted. In-line views are introduced later in this chapter. For more information about PROC SQL views, see Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243.

When you use a PROC SQL query to join tables, you must decide how you want the rows from the various tables to be combined. There are two main types of joins, as shown below.

<table>
<thead>
<tr>
<th>Type of Join</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inner join</strong></td>
<td>Only the rows that match across all table(s)</td>
</tr>
<tr>
<td><strong>Outer join</strong></td>
<td>Rows that match across tables (as in the inner join) plus nonmatching rows from one or more tables</td>
</tr>
</tbody>
</table>

When any type of join is processed, PROC SQL starts by generating a Cartesian product, which contains all possible combinations of rows from all tables. Let’s learn more about how a Cartesian product is generated.

Generating a Cartesian Product

The most basic type of join combines data from two tables that are specified in the FROM clause of a SELECT statement. When you specify multiple tables in the FROM clause but do not include a WHERE statement to subset data, PROC SQL returns the Cartesian product of the tables. In a Cartesian product, each row in the first table is combined with every row in the second table. Below is an example of this type of query, which joins the tables One and Two.
proc sql;
select *
from one, two;

The output shown above displays all possible combinations of each row in table *One* with all rows in table *Two*. Note that each table has a column named *x*, and both of these columns appear in the output. A Cartesian product includes all columns from the source tables; columns that have common names are *not* overlaid.

In most cases, generating all possible combinations of rows from multiple tables does *not* yield useful results, so a Cartesian product is rarely the query outcome that you want. For example, in the Cartesian product of two tables that contain employee information, each row of output might contain information about two different employees. Usually, you want your join to return only a subset of rows from the tables.

The size of a Cartesian product can also be problematic. The number of rows in a Cartesian product is equal to the product of the number of rows in the contributing tables.

The tables *One* and *Two*, used in the preceding example, contain three rows each, as shown below.

<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>x</em></td>
<td><em>a</em></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
</tbody>
</table>

The number of rows in the Cartesian product of tables *One* and *Two* is calculated as follows:

\[ 3 \times 3 = 9 \text{ rows} \]

Joining small tables such as *One* and *Two* results in a relatively small Cartesian product. However, the Cartesian product of large tables can be huge and can require a large amount of system resources for processing.

For example, joining two tables of 1,000 rows each results in output of the following size:

\[ 1,000 \times 1,000 = 1,000,000 \text{ rows} \]
When you run a query that involves a Cartesian product that cannot be optimized, PROC SQL writes the following warning message to the SAS log.

Table 3.1  SAS Log

| NOTE: The execution of this query involves performing one or more Cartesian product joins that cannot be optimized. |

Although you will not often choose to create a query that returns a Cartesian product, it is important to understand how a Cartesian product is built. In all types of joins, PROC SQL generates a Cartesian product first, and then eliminates rows that do not meet any subsetting criteria that you have specified.

Note: In many cases, PROC SQL can optimize the processing of a join, thereby minimizing the resources that are required to generate a Cartesian product.

---

**Using Inner Joins**

**Introducing Inner Join Syntax**

An inner join combines and displays only the rows from the first table that match rows from the second table, based on the matching criteria (also known as join conditions) that are specified in the WHERE clause. A join condition is an expression that specifies the column(s) on which the tables are to be joined.

The following diagram illustrates an inner join of two tables. The shaded area of overlap represents the matching rows (the subset of rows) that the inner join returns as output.

![Diagram of Inner Join](image)

Note: An inner join is sometimes called a conventional join.

Inner join syntax builds on the syntax of the simplest type of join that was shown earlier. In an inner join, a WHERE clause is added to restrict the rows of the Cartesian product that will be displayed in output.
General form, SELECT statement for inner join:

```
SELECT column-1,...,column-n
FROM table-1 | view-1, table-2 | view-2,...,table-n | view-n
WHERE join-condition(s)
    <AND other subsetting condition(s)>
    <other clauses>
```

where

- `join-condition(s)` refers to one or more expressions that specify the column or columns on which the tables are to be joined.
- `other subsetting condition(s)` refers to optional expressions that are used to subset rows in the query results.
- `<other clauses>` refers to optional PROC SQL clauses.

**Note:** You can combine a maximum of 32 tables in a single inner join. If the join involves views (either in-line views or PROC SQL views), it is the number of tables that underlie the views, not the number of views themselves, that counts towards the limit of 32. In-line views are covered later in this chapter and PROC SQL views are discussed in Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243.

---

**Example**

When a WHERE clause is added to the PROC SQL query shown earlier, only a subset of rows is included in output. The modified query, tables, and output are shown below:

```
proc sql;
    select *
    from one, two
    where one.x = two.x;
```

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

<table>
<thead>
<tr>
<th>X</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
</tr>
<tr>
<td>5</td>
<td>v</td>
</tr>
</tbody>
</table>

Because of the WHERE clause, this inner join does not display all rows from the Cartesian product (all possible combinations of rows from both tables) but only a subset of rows. The WHERE clause expression (join condition) specifies that the result set should include only rows whose values of column `X` in the table `One` are equal to values in column `X` of the table `Two`. Only one row from `One` and one row from `Two` have matching values of `X`. Those two rows are combined into one row of output.

**Note:** PROC SQL will not perform a join unless the columns that are compared in the join condition (in this example, `One.X` and `Two.X`) have the same data type. However, the two columns are not required to have the same name. For example, the
join condition shown in the following WHERE statement is valid if ID and EmpID have the same data type:

   where table1.id = table2.empid

   △

Note: The join condition that is specified in the WHERE clause often contains the equal (=) operator, but the expression may contain one or more other operators instead. An inner join that matches rows based on the equal (=) operator, in which the value of a column or expression in one table must be equivalent to the value of a column or expression in another table, is called an equijoin. △

Let’s take a closer look at how PROC SQL processes this inner join.

---

**Understanding How Joins Are Processed**

Understanding how PROC SQL processes inner and outer joins will help you to understand which output is generated by each type of join. Conceptually, PROC SQL follows these steps to process a join:

- builds a Cartesian product of rows from the indicated tables
- evaluates each row in the Cartesian product, based on the join conditions specified in the WHERE clause (along with any other subsetting conditions), and removes any rows that do not meet the specified conditions
- if summary functions are specified, summarizes the applicable rows
- returns the rows that are to be displayed in output.

Note: The PROC SQL query optimizer follows a more complex process than the conceptual approach described here, by breaking the Cartesian product into smaller pieces. For each query, the optimizer selects the most efficient processing method for the specific situation. △

By default, PROC SQL joins do not overlay columns with the same name. Instead, the output displays all columns that have the same name. To avoid having columns with the same name in the output from an inner or outer join, you can eliminate or rename the duplicate columns.

Tip: You can also use the COALESCE function with an inner or outer join to overlay columns with the same name. The COALESCE function is discussed, along with outer joins, later in this chapter.

---

**Eliminating Duplicate Columns**

Let’s return to the sample PROC SQL query that uses an inner join to combine the tables One and Two:

```proc sql;
   select * from one, two
   where one.x = two.x;
```

---

<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

---
The two tables have a column with an identical name (X). Because the SELECT clause in the query shown above contains an asterisk, the output displays all columns from both tables.

To eliminate a duplicate column, you can specify just one of the duplicate columns in the SELECT statement. The SELECT statement in the preceding PROC SQL query can be modified as follows:

```sql
proc sql;
  select one.x, a, b
  from one, two
  where one.x = two.x;
```

Here, the SELECT clause specifies that only the column X from table One will be included in output. The output, which now displays only one column X, is shown below.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

*Note:* In an inner equijoin, like the one shown here, it does not matter which of the same-named columns is listed in the SELECT statement. The SELECT statement in this example could have specified Two.X instead of One.X.

Another way to eliminate the duplicate X column in the preceding example is shown below:

```sql
proc sql;
  select one.*, b
  from one, two
  where one.x = two.x;
```

By using the asterisk (*) to select all columns from table One, and only B from table Two, this query generates the same output as the preceding version.

---

### Renaming a Column by Using a Column Alias

If you are working with several tables that have a column with a common name but slightly different data, you might want both columns to appear in output. To avoid the confusion of displaying two different columns with the same name, you can rename one of the duplicate columns by specifying a column alias in the SELECT statement. For example, you could modify the SELECT statement of the sample query as follows:

```sql
proc sql;
  select one.x as ID, two.x, a, b
  from one, two
  where one.x = two.x;
```

The output of the modified query is shown here.

<table>
<thead>
<tr>
<th>ID</th>
<th>X</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>b</td>
<td>x</td>
</tr>
</tbody>
</table>

Now that the column One.X has been renamed to ID, the output clearly indicates that ID and X are two different columns.
Joining Tables That Have Rows with Matching Values

Let's see what happens when you join two tables in which multiple rows have duplicate values of the column on which the tables are being joined. Each of the tables Three and Four has multiple rows that contain the value 2 for column X. The following PROC SQL inner join matches rows from the two tables based on the common column X:

```
proc sql;
select *
  from three, four
where three.x=four.x;
```

The output shows how this inner join handles the duplicate values of X.

```
<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>X</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b1</td>
<td>2</td>
<td>x1</td>
</tr>
<tr>
<td>2</td>
<td>b1</td>
<td>2</td>
<td>x2</td>
</tr>
<tr>
<td>2</td>
<td>b2</td>
<td>2</td>
<td>x1</td>
</tr>
<tr>
<td>2</td>
<td>b2</td>
<td>2</td>
<td>x2</td>
</tr>
</tbody>
</table>
```

All possible combinations of the duplicate rows are displayed. There are no matches on any other values of X, so no other rows are displayed in output.

*Note:* A DATA step match-merge would output only two rows, because it processes data sequentially from top to bottom. Later in this chapter, there is a comparison of PROC SQL joins and DATA step match-merges.

Specifying a Table Alias

To enable PROC SQL to distinguish between same-named columns from different tables, you use qualified column names. To create a qualified column name, you prefix the column name with its table name. For example, the following PROC SQL inner join contains several qualified column names (shown highlighted):

```
proc sql;
title 'Employee Names and Job Codes';
select staffmaster.empid, lastname, firstname, jobcode
  from sasuser.staffmaster, sasuser.payrollmaster
where staffmaster.empid=payrollmaster.empid;
```

It can be difficult to read PROC SQL code that contains lengthy qualified column names. In addition, typing (and retyping) long table names can be time-consuming. Fortunately, you can use a temporary, alternate name for any or all tables in any PROC SQL query. This temporary name, which is called a *table alias*, is specified after the
table name in the FROM clause. The keyword AS is often used, although its use is optional.

The following modified PROC SQL query specifies table aliases in the FROM clause, and then uses the table aliases to qualify column names in the SELECT and WHERE clauses:

```sql
proc sql;
  title 'Employee Names and Job Codes';
  select s.empid, lastname, firstname, jobcode
  from sasuser.staffmaster as s,
       sasuser.payrollmaster as p
  where s.empid=p.empid;
```

In this query, the optional keyword AS is used to define the table aliases in the FROM clause. The FROM clause would be equally valid with AS omitted, as shown below:

```sql
from sasuser.staffmaster s,
     sasuser.payrollmaster p
```

**Note:** While using table aliases will help you to work more efficiently, the use of table aliases does not cause SAS to execute the query more quickly.

Table aliases are usually optional. However, there are two situations that require their use, as shown below.

<table>
<thead>
<tr>
<th>Table aliases are required when...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>a table is joined to itself (called a self-join or reflexive join)</td>
<td>from airline.staffmaster as s1, airline.staffmaster as s2</td>
</tr>
<tr>
<td>you need to reference columns from same-named tables in different libraries</td>
<td>from airline.flightdelays as af, work.flightdelays as wf where af.delay &gt; wf.delay</td>
</tr>
</tbody>
</table>

So far, you've seen relatively simple examples of inner joins. However, as in any other PROC SQL query, inner joins can include more advanced components, such as
- titles and footers
- functions and expressions in the SELECT clause
- multiple conditions in the WHERE clause
- an ORDER BY clause for sorting
- summary functions with grouping.

Let's look at a few examples of more complex inner joins.

**Example: Complex PROC SQL Inner Join**

Suppose you want to display the names (first initial and last name), job codes, and ages of all company employees who live in New York. You also want the results to be sorted by job code and age.

The data that you need is stored in the two tables listed below.
Of the three columns that you want to display, **JobCode** is the only column that already exists in the tables. The other two columns will need to be created from existing columns.

The PROC SQL query shown here uses an inner join to generate the output that you want:

```sql
proc sql outobs=15;
title 'New York Employees';
select substr(firstname,1,1) || '.' || lastname
   as Name,
   jobcode,
   int((today() - dateofbirth)/365.25)
   as Age
from sasuser.payrollmaster as p,
    sasuser.staffmaster as s
where p.empid =
    s.empid
    and state='NY'
order by 2, 3;
```

---

<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th><strong>JobCode</strong></th>
<th><strong>Age</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>R. LONG</td>
<td>BCK</td>
<td>32</td>
</tr>
<tr>
<td>T. BURNETTE</td>
<td>BCK</td>
<td>36</td>
</tr>
<tr>
<td>J. MARKS</td>
<td>BCK</td>
<td>37</td>
</tr>
<tr>
<td>N. JONES</td>
<td>BCK</td>
<td>38</td>
</tr>
<tr>
<td>R. VANDEUSEN</td>
<td>BCK</td>
<td>44</td>
</tr>
<tr>
<td>J. PEARSON</td>
<td>BCK</td>
<td>45</td>
</tr>
<tr>
<td>L. GORDON</td>
<td>BCK</td>
<td>45</td>
</tr>
<tr>
<td>C. PEARCE</td>
<td>FA1</td>
<td>32</td>
</tr>
<tr>
<td>D. WOOD</td>
<td>FA1</td>
<td>33</td>
</tr>
<tr>
<td>C. RICHARDS</td>
<td>FA1</td>
<td>35</td>
</tr>
<tr>
<td>R. MC DANIEL</td>
<td>FA1</td>
<td>36</td>
</tr>
<tr>
<td>L. JONES</td>
<td>FA1</td>
<td>36</td>
</tr>
<tr>
<td>A. PARKER</td>
<td>FA1</td>
<td>40</td>
</tr>
<tr>
<td>D. FIELDS</td>
<td>FA1</td>
<td>46</td>
</tr>
<tr>
<td>R. PATTERTON</td>
<td>FA2</td>
<td>35</td>
</tr>
</tbody>
</table>
The SELECT clause, shown below, specifies the new column Name, the existing column JobCode, and the new column Age:

```
select substr(firstname,1,1) || ' ' || lastname
    as Name,
    jobcode,
    int((today() - dateofbirth)/365.25) 
    as Age
```

To create the two new columns, the SELECT clause uses functions and expressions as follows:

- To create Name, the SUBSTR function extracts the first initial from FirstName. Then the concatenation operator combines the first initial with a period, a space, and then the contents of the LastName column. Finally, the keyword AS names the new column.

- To calculate Age, the INT function returns the integer portion of the result of the calculation. In the expression that is used as an argument of the INT function, the employee’s birth date (DateOfBirth) is subtracted from today’s date (returned by the TODAY function), and the difference is divided by the number of days in a year (365.25).

The WHERE clause contains two expressions linked by the logical operator AND:

```
where p.empid = 
    s.empid
    and state='NY'
```

This query only outputs rows that have matching values of EmpID and rows in which the value of State is NY. You don’t need to prefix the column name State with a table name, because State occurs in only one of the tables.

---

### Example: PROC SQL Inner Join with Summary Functions

You can also summarize and group data in a PROC SQL join. To illustrate, let’s modify the previous PROC SQL inner join so that the output displays the following summarized columns for New York employees in each job code: number of employees and average age. The modified query is shown below:

```
proc sql outobs=15;
title 'Avg Age of New York Employees';
select jobcode,
    count(p.empid) as Employees,
    avg(int((today() - dateofbirth)/365.25))
    format=4.1 as AvgAge
from sasuser.payrollmaster as p,
    sasuser.staffmaster as s
where p.empid = 
    s.empid
    and state='NY'
group by jobcode
order by jobcode;
```

To create two new columns, the SELECT clause uses summary functions as follows:

- To create Employees, the COUNT function is used with p.EmpID (Payrollmaster.EmpID) as its argument.
To create \texttt{AvgAge}, the \texttt{AVG} function is used with an expression as its argument. As described in the previous example, the expression uses the \texttt{INT} function to calculate each employee’s age.

The output of this modified query is shown below.

\begin{center}
\textbf{Avg Age of New York Employees}
\end{center}

\begin{table}[h!]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{JobCode} & \textbf{Employees} & \textbf{AvgAge} \\
\hline
BCK & 7 & 39.6 \\
FA1 & 7 & 36.9 \\
FA2 & 9 & 40.4 \\
FA3 & 4 & 42.5 \\
ME1 & 5 & 36.0 \\
ME2 & 9 & 43.9 \\
ME3 & 2 & 44.0 \\
NA1 & 1 & 34.0 \\
NA2 & 1 & 44.0 \\
PT1 & 5 & 35.6 \\
PT2 & 5 & 49.8 \\
PT3 & 2 & 56.5 \\
SCP & 6 & 40.3 \\
TA1 & 5 & 39.4 \\
TA2 & 12 & 38.7 \\
\hline
\end{tabular}
\end{table}

\section*{Using Outer Joins}

\subsection*{Introducing Types of Outer Joins}

An outer join combines and displays all rows that \textit{match} across tables, based on the specified matching criteria (also known as join conditions), \textit{plus} some or all of the rows that do \textit{not} match. You can think of an outer join as an augmentation of an inner join: an outer join returns all rows generated by an inner join, plus additional (nonmatching) rows.
## Type of Outer Join

<table>
<thead>
<tr>
<th>Type of Outer Join</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>All matching rows plus nonmatching rows from the first table specified in the FROM clause (the left table)</td>
</tr>
<tr>
<td>Right</td>
<td>All matching rows plus nonmatching rows from the second table specified in the FROM clause (the right table)</td>
</tr>
<tr>
<td>Full</td>
<td>All matching rows plus nonmatching rows in both tables</td>
</tr>
</tbody>
</table>

The syntax of an outer join is shown below.

**General form, SELECT statement for outer join:**

```sql
SELECT column-1,...,column-n
FROM table-1 | view-1
LEFT JOIN | RIGHT JOIN | FULL JOIN
           table-2 | view-2
ON join-condition(s)
<other clauses>;
```

where

- **LEFT JOIN, RIGHT JOIN, FULL JOIN**
  - are keywords that specify the type of outer join.
- **ON**
  - specifies `join-condition(s)`, which are expression(s) that specify the column or columns on which the tables are to be joined.
- `<other clauses>`
  - refers to optional PROC SQL clauses.

**Note:** To further subset the rows in the query output, you can follow the ON clause with a WHERE clause. The WHERE clause subsets the individual detail rows before the outer join is performed. The ON clause then specifies how the remaining rows are to be selected for output. △

**Note:** You can perform an outer join on only two tables or views at a time. Views are covered later in this chapter. △

Let's see how each type of outer join works.
Using a Left Outer Join

A left outer join retrieves all rows that match across tables, based on the specified matching criteria (join conditions), plus nonmatching rows from the left table (the first table specified in the FROM clause).

Suppose you are using the following PROC SQL left join to combine the two tables One and Two. The join condition is stated in the expression following the ON keyword. The two tables and the three rows of output are shown below:

```
proc sql;
select *
  from one
left join
two
on one.x=two.x;
```

<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>X</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

In each row of output, the first two columns correspond to table One (the left table) and the last two columns correspond to table Two (the right table).

Because this is a left join, all rows (both matching and nonmatching) from table One (the left table) are included in the output (the first two columns). Rows from table Two (the right table) are displayed in the output (the last two columns) only if they match a row from table One.

In this example, the second row of output is the only row in which the row from table One matched a row from table Two, based on the matching criteria (join conditions) specified in the ON clause. In the first and third rows of output, the row from table One had no matching row in table Two.

**Note:** In all three types of outer joins (left, right, and full), the columns in the result (combined) row that are from the unmatched row are set to missing values.

To eliminate one of the duplicate columns (in this case, x) in any outer join, as shown earlier with an inner join, you can modify the SELECT clause to list the specific columns that will be displayed. Here, the SELECT clause from the preceding query has been modified to remove the duplicate x column:

```
proc sql;
select one.x, a, b
  from one
left join
two
on one.x=two.x;
```

<table>
<thead>
<tr>
<th>One</th>
<th>Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>
Using a Right Outer Join

A right outer join retrieves all rows that match across tables, based on the specified matching criteria (join conditions), plus nonmatching rows from the right table (the second table specified in the FROM clause).

Let’s see what happens when you use a right join to combine the two tables used in the previous example. The following PROC SQL query uses a right join to combine rows from One and Two, based on the join conditions specified in the ON clause:

```
proc sql;
  select *
  from one
  right join
  two
  on one.x=two.x;
```

In each row of output, the first two columns correspond to table One (the left table) and the last two columns correspond to table Two (the right table).

Because this is a right join, all rows (both matching and nonmatching) from table Two (the right table) are included in the output (the last two columns). Rows from table One (the left table) are displayed in the output (the first two columns) only if they match a row from table Two.

In this example, there is only one row in table One that matches a value of X in table Two, and these two matching rows combine to form the first row of output. In the remaining rows of output, there is no match and the columns corresponding to table One are set to missing values.

Using a Full Outer Join

A full outer join retrieves both matching rows and nonmatching rows from both tables.

Let’s combine the same two tables again, this time using a full join. The PROC SQL query, the tables, and the output are shown below:

```
proc sql;
  select *
  from one
  full join
  two
  on one.x=two.x;
```

In each row of output, the first two columns correspond to table One (the left table) and the last two columns correspond to table Two (the right table).

Because this is a full join, all rows (both matching and nonmatching) from both tables are included in the output (the last two columns). Rows from table One (the left table) are displayed in the output (the first two columns) only if they match a row from table Two.

In this example, there is only one row in table One that matches a value of X in table Two, and these two matching rows combine to form the first row of output. In the remaining rows of output, there is no match and the columns corresponding to table One are set to missing values.
Combining Tables Horizontally Using PROC SQL

Example: Outer Join

Now that you have seen how the three types of outer joins work, let’s look at a realistic situation requiring the use of an outer join.

Suppose you want to list all of an airline’s flights that were scheduled for March, along with corresponding delay information (if it exists). Each flight is identified by both a flight date and a flight number. Your output should display the following data: flight date, flight number, destination, and length of delay in minutes.

The data that you need is stored in the two tables shown below. The applicable columns from each table are identified.

<table>
<thead>
<tr>
<th>Table</th>
<th>Relevant Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasuser.Marchflights</td>
<td>Date, FlightNumber, Destination</td>
</tr>
<tr>
<td>Sasuser.Flightdelays</td>
<td>Date, FlightNumber, Destination, Delay</td>
</tr>
</tbody>
</table>

Your output should include the columns that are listed above and all of the following rows:
- rows that have matching values of Date and FlightNumber across the two tables
- rows from Sasuser.Marchflights that have no matching row in Sasuser.Flightdelays.

To generate the output that you want, the following PROC SQL query uses a left outer join, with Sasuser.Marchflights specified as the left (first) table.

```sql
proc sql outobs=20;
title ‘All March Flights’;
select m.date,
      m.flightnumber
   label=’Flight Number’,
      m.destination;
```

Because this is a full join, all rows (both matching and nonmatching) from both tables are included in the output. There is only one match between table One and table Two, so only one row of output displays values in all columns. All remaining rows of output contain only values from table One or table Two, with the remaining columns set to missing values.
Example: Outer Join

```
label='Left',
f.destination
label='Right',
delay
label='Delay in Minutes'
from sasuser.marchflights as m
left join
sasuser.flightdelays as f
on m.date=f.date
and m.flightnumber=
    f.flightnumber
order by delay;
```

Notice the following:

- The SELECT clause eliminates the duplicate Date and FlightNumber columns by specifying their source as Sasuser.Marchflights. However, the SELECT clause list specifies the Destination columns from both tables and assigns a table alias to each to distinguish between them.

- The ON clause contains two join conditions, which match the tables on the two columns Date and FlightNumber.

The query output is shown below.

**All March Flights**

<table>
<thead>
<tr>
<th>Date</th>
<th>Flight Number</th>
<th>Left</th>
<th>Right Delay in Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16MAR2000</td>
<td>622</td>
<td>FRA</td>
<td></td>
</tr>
<tr>
<td>03MAR2000</td>
<td>416</td>
<td>WAS</td>
<td></td>
</tr>
<tr>
<td>17MAR2000</td>
<td>182</td>
<td>YYZ</td>
<td></td>
</tr>
<tr>
<td>14MAR2000</td>
<td>21</td>
<td>CDG</td>
<td></td>
</tr>
<tr>
<td>11MAR2000</td>
<td>260</td>
<td>WAS</td>
<td></td>
</tr>
<tr>
<td>08MAR2000</td>
<td>182</td>
<td>YYZ</td>
<td></td>
</tr>
<tr>
<td>08MAR2000</td>
<td>132</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11MAR2000</td>
<td>202</td>
<td>ORD</td>
<td></td>
</tr>
<tr>
<td>29MAR2000</td>
<td>829</td>
<td>WAS</td>
<td></td>
</tr>
<tr>
<td>25MAR2000</td>
<td>872</td>
<td>LAX</td>
<td></td>
</tr>
<tr>
<td>22MAR2000</td>
<td>183</td>
<td>WAS</td>
<td></td>
</tr>
<tr>
<td>27MAR2000</td>
<td>982</td>
<td>DFW</td>
<td></td>
</tr>
<tr>
<td>25MAR2000</td>
<td>629</td>
<td>WAS</td>
<td>WAS</td>
</tr>
<tr>
<td>18MAR2000</td>
<td>219</td>
<td>LHR</td>
<td>LHR</td>
</tr>
<tr>
<td>09MAR2000</td>
<td>829</td>
<td>LHR</td>
<td>LHR</td>
</tr>
<tr>
<td>02MAR2000</td>
<td>387</td>
<td>CPH</td>
<td>CPH</td>
</tr>
<tr>
<td>14MAR2000</td>
<td>829</td>
<td>WAS</td>
<td>WAS</td>
</tr>
<tr>
<td>07MAR2000</td>
<td>523</td>
<td>ORD</td>
<td>ORD</td>
</tr>
<tr>
<td>10MAR2000</td>
<td>523</td>
<td>ORD</td>
<td>ORD</td>
</tr>
<tr>
<td>15MAR2000</td>
<td>219</td>
<td>LHR</td>
<td>LHR</td>
</tr>
</tbody>
</table>
The first 12 rows of output display rows from *Sasuser.Marchflights* (the left table) that have no matching rows in *Sasuser.Flightdelays*. Therefore, in these 12 rows, the last 2 columns are set to missing values.

*Note:* The same results could be generated by using a right outer join, with *Sasuser.Marchflights* specified as the right (second) table.

---

### Creating an Inner Join with Outer Join-Style Syntax

If you want to use a consistent syntax for all joins, you can write an inner join using the same style of syntax that is used for an outer join.

General form, SELECT statement for inner join (alternate syntax):

```
SELECT column-1,...,column-n
FROM table-1 | view-1
INNER JOIN
    table-2 | view-2
ON join-condition(s)
<other clauses>;
```

where

- **INNER JOIN** is a keyword.
- **ON** specifies *join-condition(s)*, which are expression(s) that specify the column or columns on which the tables are to be joined.
- **<other clauses>** refers to optional PROC SQL clauses.

*Note:* An inner join that uses this syntax can be performed on only two tables or views at a time. When an inner join uses the syntax presented earlier, up to 32 tables or views can be combined at once. In-line views are covered later in this chapter.

---

### Comparing SQL Joins and DATA Step Match-Merges

You should be familiar with the use of the DATA step to merge data sets. DATA step match-merges and PROC SQL joins can produce the same results. However, there are important differences between these two techniques. For example, a join does not require that you sort the data first; a DATA step match-merge requires that the data be sorted.

Let’s compare the use of SQL joins and DATA step match-merges in the following situations:

- when *all* of the values of the selected variable (column) match
- when *only some* of the values of the selected variable (column) match.
When All of the Values Match

When all of the values of the BY variable match, you can use a PROC SQL inner join to produce the same results as a DATA step match-merge.

Suppose you want to combine the tables One and Two, as shown below.

These two tables have the column X in common, and all values of X in each row match across the two tables. Both tables are already sorted by X.

The following DATA step match-merge (followed by a PROC PRINT step) and the PROC SQL inner join produce identical reports.

**DATA Step Match-Merge**
```
data merged;
    merge one two;
    by x;
run;
```
```
proc print data=merged noobs;
    title 'Table Merged';
run;
```

**PROC SQL Inner Join**
```
proc sql;
    title 'Table Merged';
    select one.x, a, b
    from one, two
    where one.x = two.x
    order by x;
run;
```

Note: The DATA step match-merge creates a data set whereas the PROC SQL inner join, as shown here, creates only a report as output. To make these two programs...
When Only Some of the Values Match

When only some of the values of the BY variable match, you can use a PROC SQL full outer join to produce the same result as a DATA step match-merge. Unlike the DATA step match-merge, however, a PROC SQL outer join does not overlay the two common columns by default. To overlay common columns, you must use the COALESCE function in the PROC SQL full outer join.

Note: The COALESCE function can also be used with inner join operators.

First, let’s see what happens when you use a PROC SQL full outer join without the COALESCE function. Suppose you want to combine the tables Three and Four. These two tables have the column X in common, but most of the values of X do not match across tables. Both tables are already sorted by X. The following DATA step match-merge (followed by a PROC PRINT step) and the PROC SQL full outer join combine these tables, but do not generate the same output. COALESCE function can also be used with inner join operators.

**DATA Step Match-Merge**
```bash
data merged;
  merge three four;
  by x;
run;
PROC SQL Full Outer Join
```  
```bash
proc sql;
title 'Table Merged';
select three.x, a, b
  from three
  full join four
  on three.x = four.x
order by x;
run;
```
When Only Some of the Values Match: Using the COALESCE Function

When you add the COALESCE function to the SELECT clause of the PROC SQL outer join, the PROC SQL outer join can produce the same result as a DATA step match-merge.

General form, COALESCE function in a basic SELECT clause:

```
SELECT COALESCE (column-1,...column-n) <AS> column-alias
<,...column-1,...column-n>
```

where

column-1 through column-n
are the names of two or more columns to be overlaid. The COALESCE function requires that all arguments have the same data type.

column-alias
is a temporary name for the coalesced column. The alias cannot contain any spaces. If another clause in the SELECT statement, such as an ORDER BY clause, references the coalesced column, a column alias must be specified.

The COALESCE function overlays the specified columns by
Checking the value of each column in the order in which the columns are listed
returning the first value that is a SAS nonmissing value.

**Note:** If all returned values are missing, COALESCE returns a missing value.

Below is the SELECT clause from the PROC SQL full outer join shown in the previous section, with the COALESCE function added:

```sql
select coalesce(three.x, four.x) as x, a, b
```

This SELECT clause overlays the X columns from tables *Three* and *Four*. The column alias X is assigned to the coalesced column.

When the COALESCE function is added to the preceding PROC SQL full outer join, the DATA step match-merge (with PROC PRINT step) and the PROC SQL full outer join will combine rows in the same way. The two programs, the tables, and the output are shown below.

### DATA Step Match-Merge
```sas
data merged;
  merge three four;
  by x;
run;
proc print data=merged noobs;
title 'Table Merged';
run;
```

### PROC SQL Full Outer Join
```sql
proc sql;
title 'Table Merged';
select coalesce(three.x, four.x) as x, a, b
  from three
  full join four
  on three.x = four.x;
```

<table>
<thead>
<tr>
<th>Three</th>
<th>X</th>
<th>A</th>
<th>X</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>a</td>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>b</td>
<td>3</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>d</td>
<td>5</td>
<td>v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Merged</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

### Understanding the Advantages of PROC SQL Joins

DATA step match-merges and PROC SQL joins both have advantages and disadvantages. Here are some of the main advantages of PROC SQL joins.
Advantage | Example
---|---
PROC SQL joins do not require sorted or indexed tables. | `proc sql;` `select table1.x, a, b` `from table1` `full join` `table2` `on table1.x = table2.x;` `where table-1 is sorted by column X and table-2 is not` 

PROC SQL joins do not require that the columns in join expressions have the same name. | `proc sql;` `select table1.x, lastname, status` `from table1, table2` `where table1.id = table2.custnum;` 

PROC SQL joins can use comparison operators other than the equal sign (=). | `proc sql;` `select a.itemnumber, cost, price` `from table1 as a,` `table2 as b` `where a.itemnumber = b.itemnumber` `and a.cost>b.price;` 

Note: Join performance can be substantially improved when the tables are indexed on the column(s) on which the tables are being joined. You can learn more about indexing in Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221. 

You can learn more about the comparative advantages and disadvantages of DATA step match-merges and PROC SQL joins in Chapter 15, “Combining Data Horizontally,” on page 513.

Using In-Line Views

Introducing In-Line Views

Sometimes, you might want to specify an in-line view rather than a table as the source of data for a PROC SQL query. An in-line view is a nested query that is specified in the outer query’s **FROM clause**. (You should already be familiar with a subquery, which is a nested query that is specified in a WHERE clause.) An in-line view selects data from one or more tables to produce a temporary (or virtual) table that the outer query then uses to select data for output.

For example, the following FROM clause specifies an in-line view:

```sql
from (select flightnumber, date, boarded/passegercapacity*100 as pctfull format=4.1 label='Percent Full' from sasuser.marchflights)
```

This in-line view selects two existing columns (FlightNumber and Date) and defines the new column PctFull based on the table Sasuser.Marchflights.
Unlike a table, an in-line view exists only during query execution. Because it is temporary, an in-line view can be referenced only in the query in which it is defined. In addition, an in-line view can be assigned an alias but not a permanent name.

**Note:** In a FROM clause, you can also specify a PROC SQL view, which is a query that has been created (using the CREATE statement) and stored. You can learn more about creating PROC SQL views in Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243.

**CAUTION:**
Unlike other queries, an in-line view cannot contain an ORDER BY clause.

There are two potential advantages to using an in-line view rather than a table in a PROC SQL query:
- The complexity of the code is usually reduced, so that the code is easier to write and understand.
- In some cases, PROC SQL might be able to process the code more efficiently.

### Referencing an In-Line View with Other Views or Tables

The preceding FROM clause is from a simple PROC SQL query that references just one data source: the in-line view. However, a PROC SQL query can join multiple tables and in-line views. For example, the FROM clause shown below specifies both a table (`Sasuser.Flightschedule`) and an in-line view.

```sql
from sasuser.marchflights,
    (select flightnumber, date,
         boarded/passengercapacity*100
         as pctfull
         format=4.1 label=’Percent Full’
    from sasuser.marchflights)
```

In other words, you can base an in-line view on a join.

**Note:** Remember that each table that is referenced in an in-line view counts toward the 32-table limit for an inner join.
Assigning an Alias to an In-Line View

You can assign an alias to an in-line view just as you can to a table. In the following example, the alias $f$ has been added in the first FROM clause to reference the table `Sasuser.Flightschedule` and the alias $m$ has been added in the second FROM clause to reference the in-line view. After the main FROM clause, a WHERE clause that uses both of the aliases has been added.

```
from sasuser.flightschedule as f,
    (select flightnumber, date
     boarded/passengercapacity*100
     as pctfull
     format=4.1 label='Percent Full'
     from sasuser.marchflights) as m
where m.flightnumber=f.flightnumber
    and m.date=f.date
```

Example: Query That Contains an In-Line View

Suppose you want to identify the air travel destinations that experienced the worst delays in March. You would like your output to show all of the following data:

- destination
- average delay
- maximum delay
- probability of delay.

Your PROC SQL query uses an in-line view (shown highlighted) to calculate all of the new columns except for the last one:

```
proc sql;
title "Flight Destinations and Delays";
select destination,
    average format=3.0 label='Average Delay',
    max format=3.0 label='Maximum Delay',
    late/(late+early) as prob format=5.2
    label='Probability of Delay'
from (select destination,
    avg(delay) as average,
    max(delay) as max,
    sum(delay > 0) as late,
    sum(delay <= 0) as early
    from sasuser.flightdelays
    group by destination)
order by average;
```

Let’s look at each clause of the outer query, starting with the FROM clause, because PROC SQL evaluates the FROM clause before the SELECT clause.

The `FROM clause` specifies an in-line view rather than a table. The in-line view (nested query) specifies the following columns that are in the table `Sasuser.Flightdelays` or are based on a column in that table:

- the existing column `Destination`
- the new column `Average`
- the new column `Max`
Combining Tables Horizontally Using PROC SQL

Example: Query That Contains an In-Line View

The columns **Average**, **Max**, **Late**, and **Early** are all calculated by using summary functions.

In the calculation for the columns **Late** and **Early**, a Boolean expression is used as the argument for the summary function. A Boolean function resolves either to 1 (true) or 0 (false). For example, **Late** is calculated by taking the sum of the Boolean expression `delay > 0`. For every value of **Delay** that is greater than 0, the Boolean expression resolves to 1; values that are equal to or less than 0 resolve to 0. The SUM function adds all values of **Delay** to indicate the number of delays that occurred for each destination.

The in-line view concludes with the clause **group by destination**, specifying that the in-line view data should be grouped and summarized by the values of **Destination**.

If you submitted this in-line view (nested query) as a separate query, it would generate the following output.

Next, let’s look at the outer query’s SELECT and ORDER BY clauses (shown highlighted):

```
proc sql;
title "Flight Destinations and Delays";
select destination,
    average format=3.0 label='Average Delay',
    max format=3.0 label='Maximum Delay',
    late/(late+early) as prob format=5.2 label='Probability of Delay'
from (select destination,
    avg(delay) as average,
    max(delay) as max,
    sum(delay > 0) as late,
    sum(delay <= 0) as early
    from sasuser.flightdelays
    group by destination)
order by average;
```

The outer query’s **SELECT clause** specifies columns as follows:

- **Destination** is an existing column in the table.
- **Average** and **Max** are calculated in the in-line view, and are assigned labels in this **SELECT clause**.

<table>
<thead>
<tr>
<th>Destination</th>
<th>average</th>
<th>max</th>
<th>late</th>
<th>early</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDG</td>
<td>9.12308</td>
<td>39</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>CPH</td>
<td>5.74074</td>
<td>26</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>DFW</td>
<td>2.72131</td>
<td>20</td>
<td>38</td>
<td>23</td>
</tr>
<tr>
<td>FRA</td>
<td>5.06153</td>
<td>34</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>LAX</td>
<td>4.86667</td>
<td>27</td>
<td>82</td>
<td>41</td>
</tr>
<tr>
<td>LHR</td>
<td>5.55172</td>
<td>30</td>
<td>39</td>
<td>18</td>
</tr>
<tr>
<td>ORD</td>
<td>3.03260</td>
<td>19</td>
<td>51</td>
<td>41</td>
</tr>
<tr>
<td>WAS</td>
<td>0.63576</td>
<td>15</td>
<td>76</td>
<td>75</td>
</tr>
<tr>
<td>YYY</td>
<td>2.01667</td>
<td>14</td>
<td>36</td>
<td>24</td>
</tr>
</tbody>
</table>
Prob (with the label “Probability of Delay”) is calculated in this SELECT clause by using two columns that were calculated in the in-line view: Late and Early. The outer query’s SELECT clause can refer to the calculated columns Late and Early without using the keyword CALCULATED because PROC SQL evaluates the in-line view (the outer query’s FROM clause) first.

The outer query’s last clause is an ORDER BY clause. The output will be sorted by the values of Average.

This PROC SQL query generates the following output.

Flight Destinations and Delays

<table>
<thead>
<tr>
<th>Destination</th>
<th>Average Delay</th>
<th>Maximum Delay</th>
<th>Probability of Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAS</td>
<td>1</td>
<td>15</td>
<td>0.50</td>
</tr>
<tr>
<td>YYZ</td>
<td>2</td>
<td>14</td>
<td>0.80</td>
</tr>
<tr>
<td>DFW</td>
<td>3</td>
<td>20</td>
<td>0.82</td>
</tr>
<tr>
<td>ORD</td>
<td>3</td>
<td>19</td>
<td>0.55</td>
</tr>
<tr>
<td>LAX</td>
<td>5</td>
<td>27</td>
<td>0.87</td>
</tr>
<tr>
<td>LHR</td>
<td>6</td>
<td>30</td>
<td>0.87</td>
</tr>
<tr>
<td>CPH</td>
<td>6</td>
<td>26</td>
<td>0.59</td>
</tr>
<tr>
<td>FRA</td>
<td>6</td>
<td>34</td>
<td>0.54</td>
</tr>
<tr>
<td>CDG</td>
<td>9</td>
<td>39</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Later in this chapter, you will see a PROC SQL query that combines multiple tables and uses an in-line view.

Joining Multiple Tables and Views

So far, this chapter has presented PROC SQL queries that combine only two tables horizontally. However, there might be situations in which you have to create complex queries to combine more than two tables. Let’s look at an example of a complex query that combines four different tables.

Example: Complex Query That Combines Four Tables

Suppose you want to list the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000. To solve this problem, you will need to use the following four tables.

<table>
<thead>
<tr>
<th>Table</th>
<th>Relevant Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasuser:Flightschedule</td>
<td>identifies the crew who flew to Copenhagen on March 4, 2000</td>
</tr>
<tr>
<td>Sasuser:Staffmaster</td>
<td>identifies the names and states of residence for the employees</td>
</tr>
</tbody>
</table>
Example: Technique 1 (PROC SQL Subqueries, Joins, and In-Line Views)

<table>
<thead>
<tr>
<th>Table</th>
<th>Relevant Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasuser.Payrollmaster</td>
<td>identifies the job categories for the employees</td>
</tr>
<tr>
<td>Sasuser.Supervisors</td>
<td>identifies the employees who are supervisors</td>
</tr>
</tbody>
</table>

Note: Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

This problem can be handled in a number of different ways. Let's examine and compare three different techniques:

- **Technique 1**: using PROC SQL subqueries, joins, and in-line views
- **Technique 2**: using a multi-way join that combines four different tables and a reflexive join (joining a table with itself)
- **Technique 3**: using traditional SAS programming (a series of PROC SORT and DATA steps, followed by a PROC PRINT step).

Example: Technique 1 (PROC SQL Subqueries, Joins, and In-Line Views)

<table>
<thead>
<tr>
<th>Task</th>
<th>List the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Sasuser.Flightschedule (EmpID, Date, Destination)</td>
</tr>
<tr>
<td></td>
<td>Sasuser.Staffmaster (EmpID, FirstName, LastName, State)</td>
</tr>
<tr>
<td></td>
<td>Sasuser.Payrollmaster (EmpID, JobCode)</td>
</tr>
<tr>
<td></td>
<td>Sasuser.Supervisors (EmpID, State, JobCategory)</td>
</tr>
</tbody>
</table>

Note: Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

Completing the stated task requires a complex query that includes several subqueries, joins, and an in-line view. To make the task more manageable, let’s build the complex query piece-by-piece in four steps:

1. Identify the crew for the Copenhagen flight.
2. Find the states and job categories of the crew members that were returned by the first query.
3. Find the employee numbers of the crew supervisors, based on the states and job categories that were returned by the second query.
4. Find the names of the supervisors, based on the employee numbers that were returned by the third query.

Note that at each of the four steps, a new piece of the final query will be added. The final query will include the four separate pieces.

**Query 1: Identify the crew for the Copenhagen (CPH) flight**

This query lists the employee ID numbers of all six crew members on the Copenhagen flight:
Query 2: Find the states and job categories of the crew members

Query 1 becomes a subquery and returns the employee ID numbers of the six Copenhagen crew members to the outer query, Query 2. (Query 2 is shown in bold.) Query 2 uses an inner join to combine two tables. Query 2 selects the job category (by using the SUBSTR function to extract the job category from JobCode) and state for each of the six crew members.

```sql
proc sql;
select substr(JobCode,1,2) as JobCategory,
       state
from sasuser.staffmaster as s,
sasuser.payrollmaster as p
where s.empid=p.empid and s.empid in
  (select empid
   from sasuser.flightschedule
   where date='04mar2000'd
        and destination='CPH');
```

Query 3: Find the employee numbers of the crew supervisors

Query 2 becomes an in-line view within Query 3, and the alias c has been assigned to the in-line view. (Query 3 is shown in bold.) Query 2 returns to Query 3 the job category and state for each crew member. Query 3 selects the employee ID numbers for supervisors whose job category and state match the job category and state of a crew member.

Note: Sasuser.Supervisors specifies the label Supervisor ID for the EmpID column, and this label appears in the output.

```sql
proc sql;
select empid
from sasuser.supervisors as m,
     (select substr(JobCode,1,2) as JobCategory,
          state
       from sasuser.staffmaster as s,
sasuser.payrollmaster as p
       where s.empid=p.empid and s.empid in
         (select empid
          from sasuser.flightschedule
          where date='04mar2000'd
               and destination='CPH')) as c
where m.jobcategory=c.jobcategory
     and m.state=c.state;
```
Note that two rows contain the same value of **EmpID**: 1983. This duplication indicates that two different crew members have the same manager. In all, there are five supervisors for the six crew members of the Copenhagen flight.

**Query 4: Find the names of the supervisors**

Query 3 becomes a subquery within Query 4. (Query 4 is shown highlighted.) Query 3 returns to Query 4 the employee numbers (supervisor ID's) for the supervisors of the Copenhagen crew. Query 4 selects the names of the supervisors.

```
proc sql;
select firstname, lastname
   from sasuser.staffmaster
where empid in
   (select empid
    from sasuser.supervisors as m,
    (select substr(jobcode,1,2)
     as JobCategory,
     state
    from sasuser.staffmaster as s,
    sasuser.payrollmaster as p
    where s.empid=p.empid
    and s.empid in
    (select empid
     from sasuser.flightschedule
     where date='04mar2000'd
    and destination='CPH'))
    as c
    where m.jobcategory=c.jobcategory
    and m.state=c.state);
```

Note that the output has five rows, one for each supervisor. The duplicate name of a supervisor has been eliminated.

Technique 1 produces a PROC SQL query that includes
- four SELECT statements
- four tables, each read separately.

This program is not optimized and, in addition, includes complex code that is likely to take a long time to write.

---

**Example: Technique 2 (PROC SQL Multi-way Join with Reflexive Join)**

<table>
<thead>
<tr>
<th>Task</th>
<th>List the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td><em>Sasuser.Flightschedule</em> (<strong>EmpID, Date, Destination</strong>)</td>
</tr>
<tr>
<td></td>
<td><em>Sasuser.Staffmaster</em> (<strong>EmpID, FirstName, LastName, State</strong>)</td>
</tr>
<tr>
<td></td>
<td><em>Sasuser.Payrollmaster</em> (<strong>EmpID, JobCode</strong>)</td>
</tr>
<tr>
<td></td>
<td><em>Sasuser.Supervisors</em> (<strong>EmpID, State, JobCategory</strong>)</td>
</tr>
</tbody>
</table>

*Note:* Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.
You can also solve this problem by using a multi-way join with a reflexive join (joining a table to itself). The code is shown below:

```sql
proc sql;
select distinct e.firstname, e.lastname
from sasuser.flightschedule as a,
 sasuser.staffmaster as b,
 sasuser.payrollmaster as c,
 sasuser.supervisors as d,
 sasuser.staffmaster as e
where a.date='04mar2000'd and
 a.destination='CPH' and
 a.empid=b.empid and
 a.empid=c.empid and
 d.jobcategory=substr(c.jobcode,1,2)
and d.state=b.state
and d.empid=e.empid;
```

Example: Technique 3 (Traditional SAS Programming)

<table>
<thead>
<tr>
<th>Task</th>
<th>List the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Sasuser.Flightschedule (EmpID, Date, Destination)</td>
</tr>
<tr>
<td></td>
<td>Sasuser.Staffmaster (EmpID, FirstName, LastName, State)</td>
</tr>
<tr>
<td></td>
<td>Sasuser.Payrollmaster (EmpID, JobCode)</td>
</tr>
<tr>
<td></td>
<td>Sasuser.Supervisors (EmpID, State, JobCategory)</td>
</tr>
</tbody>
</table>

Note: Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

For comparison, look at the traditional SAS programming that can be used to solve this problem. The code is shown below, followed by the output.

```sas
/* Find the crew for the flight. */

proc sort data=sasuser.flightschedule (drop=flightnumber)
   out=crew (keep=empid);
   where destination='CPH' and date='04MAR2000'd;
   by empid;
run;
```
/* Find the State and job code for the crew. */

proc sort data=sasuser.payrollmaster
   (keep=empid jobcode)
   out=payroll;
   by empid;
run;

proc sort data=sasuser.staffmaster
   (keep=empid state firstname lastname)
   out=staff;
   by empid;
run;

data st_cat (keep=state jobcategory);
   merge crew (in=c)
      staff
      payroll;
   by empid;
   if c;
   jobcategory=substr(jobcode,1,2);
run;

/* Find the supervisor IDs. */

proc sort
   data=st_cat;
   by jobcategory state;
run;

proc sort data=sasuser.supervisors
   out=superv;
   by jobcategory state;
run;

data super (keep=empid);
   merge st_cat(in=s)
      superv;
   by jobcategory state;
   if s;
run;

/* Find the names of the supervisors. */

proc sort data=super;
   by empid;
run;

data names(drop=empid);
   merge super (in=super)
      staff (keep=empid firstname lastname);
   by empid;
   if super;
run;
   proc print data=names noobs uniform;
run;

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARSHBURN</td>
<td>JASPER</td>
</tr>
<tr>
<td>DENNIS</td>
<td>ROGER</td>
</tr>
<tr>
<td>RIVERS</td>
<td>SIMON</td>
</tr>
<tr>
<td>YOUNG</td>
<td>DEBORAH</td>
</tr>
<tr>
<td>DEAN</td>
<td>SHARON</td>
</tr>
<tr>
<td>DEAN</td>
<td>SHARON</td>
</tr>
</tbody>
</table>

This output is not identical to the output of the PROC SQL approaches (Techniques 1 and 2). The SQL queries eliminated the duplicate names that are seen here. When you use Technique 3, you can eliminate duplicates by adding the NODUPKEY option to the last PROC SORT statement, as shown below:

   proc sort data=super nodupkey;

   Based on a mainframe benchmark in batch mode, the SQL queries use less CPU time, but more I/O operations, than this non-SQL program.
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Understanding Joins

A PROC SQL join is a query that combines tables horizontally (side by side) by combining rows. The two main types of joins are inner joins and outer joins.

Generating a Cartesian Product

When you specify multiple tables in the FROM clause but do not include a WHERE statement to subset data, PROC SQL returns the Cartesian product of the tables. In a Cartesian product, each row in the first table is combined with every row in the second table. In all types of joins, PROC SQL generates a Cartesian product first, and then eliminates rows that do not meet any subsetting criteria that you have specified.

Using Inner Joins

An inner join combines and displays the rows from the first table that match rows from the second table, based on the matching criteria (also known as join conditions) that are specified in the WHERE clause. When the tables that are being joined contain a column with a common name, you might want to eliminate the duplicate column from results or specify a column alias to rename one of the duplicate columns. To refer to tables in an inner join, or in any PROC SQL step, you can specify a temporary name called a table alias.

Using Outer Joins

An outer join combines and displays all rows that match across tables, based on the specified matching criteria (also known as join conditions), plus some or all of the rows that do not match. There are three types of outer joins:

- A left outer join retrieves all rows that match across tables, based on the specified matching criteria (join conditions), plus nonmatching rows from the left table (the first table specified in the FROM clause).
- A right outer join retrieves all rows that match across tables, based on the specified matching criteria (join conditions), plus nonmatching rows from the right table (the second table specified in the FROM clause).
- A full outer join retrieves both matching rows and nonmatching rows from both tables.

Creating an Inner Join with Outer Join-Style Syntax

If you want to use a consistent syntax for all joins, you can write an inner join using the same style of syntax as used for an outer join.
Comparing SQL Joins and DATA Step Match-Merges

DATA step match-merges and PROC SQL joins can produce the same results, although there are important differences between these two techniques.

- When all the values of the BY variable (column) match and there are no duplicate BY variables, you can use a PROC SQL inner join.
- When only some of the values of the BY variable match, you can use a PROC SQL full outer join. To overlay common columns, you must use the COALESCE function with the PROC SQL join.

Using In-Line Views

An in-line view is a subquery that appears in a FROM clause. An in-line view selects data from one or more tables to produce a temporary (or virtual) table that the outer query uses to select data for output. You can reference an in-line view with other views or tables, reference multiple tables in an in-line view, and assign an alias to an in-line view.

Joining Multiple Tables and Views

When you perform a complex query that combines more than two tables or views, you might need to choose between several different techniques.

Syntax

PROC SQL;

```
SELECT column-1,...,column-n
FROM table-1 | view-1, table-2 | view-2,...,table-n | view-n>
WHERE join-condition(s)
   <AND other subsetting condition(s)>
   <other clauses>;
```

```
SELECT column-1,...,column-n
FROM table-1 | view-1
    LEFT JOIN | RIGHT JOIN | FULL JOIN
    table-2 | view-2
    ON join-condition(s)
   <other clauses>;
```

```
SELECT column-1,...,column-n
FROM table-1 | view-1
    INNER JOIN
    table-2 | view-2
    ON join-condition(s)
   <other clauses>;
```

```
SELECT COALESCE (column-1,...,column-n)
   <AS column-alias>column-1,...,column-n>
FROM table-1 | view-1
```
LEFT JOIN | RIGHT JOIN | FULL JOIN

\[
\text{table-2} \mid \text{view-2} \\
\text{ON} \ \text{join-condition(s)} \\
<\text{other clauses}>;
\]

QUIT;

Sample Programs

Combining Tables by Using an Inner Join

```sql
proc sql outobs=15;
  title 'New York Employees';
  select substr(firstname,1,1) || ' ' || lastname
    as Name,
    jobcode,
    int((today() - dateofbirth)/365.25)
    as Age
  from sasuser.payrollmaster as p,
  sasuser.staffmaster as s
  where p.empid =
    s.empid
  and state='NY'
  order by 2, 3;
quit;
```

Combining Tables by Using a Left Outer Join

```sql
proc sql outobs=20;
  title 'All March Flights';
  select m.date,
    m.flightnumber
    label='Flight Number',
    m.destination
    label='Left',
    f.destination
    label='Right',
    delay
    label='Delay in Minutes'
  from sasuser.marchflights as m
  left join
    sasuser.flightdelays as f
  on m.date=f.date
  and m.flightnumber=
    f.flightnumber
  order by delay;
quit;
```

Overlaying Common Columns in a Full Outer Join

```sql
proc sql;
  select coalesce(p.empid, s.empid)
```
Joining Tables by Using a Subquery and an In-Line View

```sql
proc sql;
select empid
from sasuser.supervisors as m,
(select substr(jobcode,1,2) as JobCategory,
state
from sasuser.staffmaster as s,
sasuser.payrollmaster as p
where s.empid=p.empid and s.empid in
(select empid
from sasuser.flightschedule
where date='04mar2000'd
and destination='CPH')) as c
where m.jobcategory=c.jobcategory
and m.state=c.state;
quit;
```

Points to Remember

- In most cases, generating all possible combinations of rows from multiple tables does not yield useful results, so a Cartesian product is rarely the query outcome that you want.
- You can combine a maximum of 32 tables in a single inner join. (If the join involves views, it is the number of tables that underlie the views, not the number of views, that counts towards the limit of 32.) An outer join can be performed on only two tables or views at a time.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. A Cartesian product is returned when
   a. join conditions are not specified in a PROC SQL join.
   b. join conditions are not specified in a PROC SQL set operation.
   c. more than two tables are specified in a PROC SQL join.
   d. the keyword ALL is used with the OUTER UNION operator.

2. Given the PROC SQL query and tables shown below, which output is generated?
proc sql;
    select *
    from store1,
    store2
    where store1.wk=
    store2.wk;

<table>
<thead>
<tr>
<th>Wk</th>
<th>Sales</th>
<th>Wk</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$515.07</td>
<td>1</td>
<td>$1368.99</td>
</tr>
<tr>
<td>2</td>
<td>$772.29</td>
<td>2</td>
<td>$1506.23</td>
</tr>
<tr>
<td>3</td>
<td>$888.88</td>
<td>3</td>
<td>$1200.57</td>
</tr>
<tr>
<td>4</td>
<td>$1000.01</td>
<td>4</td>
<td>$1784.11</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
<td>$43.00</td>
</tr>
</tbody>
</table>

(a) Store1

(b) Store2

(c)
<table>
<thead>
<tr>
<th>Wk</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$515.07</td>
</tr>
<tr>
<td>2</td>
<td>$772.29</td>
</tr>
<tr>
<td>3</td>
<td>$888.88</td>
</tr>
<tr>
<td>4</td>
<td>$1000.01</td>
</tr>
<tr>
<td>1</td>
<td>$1368.99</td>
</tr>
<tr>
<td>2</td>
<td>$1506.23</td>
</tr>
<tr>
<td>3</td>
<td>$1200.57</td>
</tr>
<tr>
<td>4</td>
<td>$1784.11</td>
</tr>
</tbody>
</table>
Given the PROC SQL query and tables shown below, which output is generated?

```sql
proc sql;
    select s.*, bonus
    from bonus as b
    right join
    salary as s
    on b.id=
    s.id;
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Salary</td>
<td>Bonus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>--------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>70000</td>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>80000</td>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>978</td>
<td>55000</td>
<td>3500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Salary</td>
<td>Bonus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>--------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>70000</td>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>80000</td>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>744</td>
<td>55000</td>
<td>3500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Salary</td>
<td>Bonus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>--------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>70000</td>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>80000</td>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>744</td>
<td>55000</td>
<td>3500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Salary</td>
<td>Bonus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>--------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>70000</td>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>80000</td>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>978</td>
<td>55000</td>
<td>3500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d
4 Which PROC SQL query produces the same output as the query shown here?

```sql
proc sql;
    select a.*,
            duration
    from groupa as a,
            groupb as b
    where a.obs=b.obs;
```

**Note:** Assume that the table Groupa contains the columns Obs and Med. Groupb contains the columns Obs and Duration.

a proc sql;
    select a.obs label='Obs',
            med,
            b.obs label='Obs',
            duration
    from groupa as a, groupb as b
    where a.obs=b.obs;

b proc sql;
    select coalesce(a.obs, b.obs)
            label='Obs', med, duration
    from groupa as a
    full join
    groupb as b
    on a.obs=b.obs;

c proc sql;
    select a.*, duration
    from groupa as a
    left join
    groupb as b
    on a.obs=b.obs;

d proc sql;
    select a.*, duration
    from groupa as a
    inner join
    groupb as b
    on a.obs=b.obs;

5 Which output will the following PROC SQL query generate?

```sql
proc sql;
    select * 
    from table1 
    left join 
    table2 
    on table1.g3= 
    table2.g3;
```
6 In order for PROC SQL to perform an inner join,
   a the tables being joined must contain the same number of columns.
   b the tables must be sorted before they are joined.
   c the columns that are specified in a join condition in the WHERE clause must have the same data type.
   d the columns that are specified in a join condition in the WHERE clause must have the same name.

7 Which statement about in-line views is false?
   a Once defined, an in-line view can be referenced in any PROC SQL query in the current SAS session.
   b An in-line view can be assigned a table alias but not a permanent name.
   c In-line views can be combined with tables in PROC SQL joins and set operations.
   d This PROC SQL query contains an in-line view that uses valid syntax:

```sql
proc sql;
  select name, numvisits
  from (select name, sum(checkin) as numvisits
        from facility as f, members as m
        where area='POOL' and
        f.id=m.id
        group by name)
```
where numvisits<=10
order by 1;

8 Which PROC SQL query will generate the same output as the DATA step match-merge and PRINT step shown below?

```
data merged;
  merge table1 table2;
  by g3;
run;

proc print data=merged
  noobs;
  title 'Merged';
run;
```

<table>
<thead>
<tr>
<th></th>
<th>Table1</th>
<th>Table2</th>
<th>Merged</th>
</tr>
</thead>
<tbody>
<tr>
<td>G3</td>
<td>Z</td>
<td>G3</td>
<td>B</td>
</tr>
<tr>
<td>46</td>
<td>UI</td>
<td>46</td>
<td>BC</td>
</tr>
<tr>
<td>47</td>
<td>BA</td>
<td>85</td>
<td>FL</td>
</tr>
<tr>
<td>69</td>
<td>FL</td>
<td>99</td>
<td>BA</td>
</tr>
<tr>
<td>48</td>
<td>UI</td>
<td>48</td>
<td>BC</td>
</tr>
<tr>
<td>47</td>
<td>BA</td>
<td>85</td>
<td>FL</td>
</tr>
<tr>
<td>88</td>
<td>FL</td>
<td>99</td>
<td>BA</td>
</tr>
</tbody>
</table>

a proc sql;
  title 'Merged';
  select a.g3, z, r
  from table1 as a
  full join
  table2 as b
  on a.g3 = b.g3
  order by 1;

b proc sql;
  title 'Merged';
  select a.g3, z, r
  from table1 as a
  full join corr
  table2 as b
  on a.g3 = b.g3
  order by 1;

c proc sql;
  title 'Merged';
  select coalesce(a.g3, b.g3)
    label='G3', z, r
  from table1 as a
  full join
  table2 as b
  on a.g3 = b.g3
  order by 1;

d proc sql;
  title 'Merged';
  select g3, z, r
  from table1 as a
  full join
  table2 as b
  on a.g3 = b.g3
  order by 1;
9 A PROC SQL inner join can combine
   a a maximum of 2 tables or in-line views, but multiple joins can be chained together.
   b a maximum of 32 tables or 2 in-line views.
   c a maximum of 32 tables, which includes any tables referenced by an in-line view.
   d a maximum of 2 tables and 32 columns.

10 Which statement about the use of table aliases is false?
   a Table aliases must be used when referencing identical table names from different libraries.
   b Table aliases can be referenced by using the keyword AS.
   c Table aliases (or full table names) must be used when referencing a column name that is the same in two or more tables.
   d Table aliases must be used when using summary functions.
Overview

Introduction

Objectives

Prerequisites

Understanding Set Operations

Example

Processing a Single Set Operation

Using Multiple Set Operators

Example

Processing Multiple Set Operations

Introducing Set Operators

Processing Unique vs. Duplicate Rows

Combining and Overlaying Columns

Modifying Results by Using Keywords

Using the EXCEPT Set Operator

Using the EXCEPT Operator Alone

Using the Keyword ALL with the EXCEPT Operator

Using the Keyword CORR with the EXCEPT Operator

Using the Keywords ALL and CORR with the EXCEPT Operator

Example: EXCEPT Operator

Example: EXCEPT Operator in an In-Line View

Using the INTERSECT Set Operator

Using the INTERSECT Operator Alone

Using the Keyword ALL with the INTERSECT Operator

Using the Keyword CORR with the INTERSECT Operator

Using the Keywords ALL and CORR with the INTERSECT Operator

Example: INTERSECT Operator

Using the UNION Set Operator

Using the UNION Operator Alone

Using the Keyword ALL with the UNION Operator

Using the Keyword CORR with the UNION Operator

Using the Keywords ALL and CORR with the UNION Operator

Example: UNION Operator

Example: UNION Operator and Summary Functions

Using the OUTER UNION Set Operator

Using the OUTER UNION Operator Alone

Using the Keyword CORR with the OUTER UNION Operator

Example: OUTER UNION Operator

Comparing Outer Unions and Other SAS Techniques

Program 1: PROC SQL OUTER UNION Set Operation with CORR

Program 2: DATA Step, SET Statement, and PROC PRINT Step
Overview

Introduction

Suppose you are generating a report based on data from a health clinic. You want to display the results of individual patient stress tests taken in 1998 (which are stored in table A), followed by the results from stress tests taken in 1999 (which are stored in table B). Instead of combining the table rows horizontally, as you would in a PROC SQL join, you want to combine the table rows vertically (one on top of the other).

When you need to select data from multiple tables and combine the tables vertically, PROC SQL can be an efficient alternative to using other SAS procedures or the DATA step. In a PROC SQL set operation, you use one of four set operators (EXCEPT, INTERSECT, UNION, and OUTER UNION) to combine tables (and views) vertically by combining the results of two queries:

```sql
proc sql;
  select *
    from a
  set-operator
  select *
    from b;
```

Each set operator combines the query results in a different way.

In this chapter, you will learn how to use the various set operators, with or without the optional keywords ALL and CORR (CORRESPONDING), to combine the results of multiple queries.

Note: In this chapter, the references to tables are also applicable to views, unless otherwise noted.

Objectives

In this chapter, you learn to
combine the results of multiple PROC SQL queries in different ways by using the set operators EXCEPT, INTERSECT, UNION, and OUTER UNION

- modify the results of a PROC SQL set operation by using the keywords ALL and CORR (CORRESPONDING)

- compare PROC SQL outer unions with other SAS techniques.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79.

Understanding Set Operations

A set operation is a SELECT statement that contains

- two groups of query clauses (each group beginning with a SELECT clause)
- a set operator
- optionally, one or both of the keywords ALL and CORR (CORRESPONDING).

General form, SELECT statement containing one set operation:

```sql
SELECT column-1<, ... column-n>
FROM table-1 | view-1<, ... table-n | view-n>
<optional query clauses>
set-operator <ALL> <CORR>
SELECT column-1<, ... column-n>
FROM table-1 | view-1<, ... table-n | view-n>
<optional query clauses>;
```

where

- `SELECT` specifies the column(s) that will appear in the result.
- `FROM` specifies the table(s) or view(s) to be queried.
- `optional query clauses` are used to refine the query further and include the clauses WHERE, GROUP BY, HAVING, and ORDER BY.
- the `set-operator` is one of the following: EXCEPT|INTERSECT|UNION|OUTER UNION.
- the optional keywords ALL and CORR (CORRESPONDING) further modify the set operation.

The SELECT statement contains one semicolon, which is placed after the last group of query clauses.
Example

In the following PROC SQL step, the SELECT statement contains one set operation. The set operation uses the set operator UNION to combine the result of a query on the table `Sasuser.Stress98` with the result of a query on the table `Sasuser.Stress99`.

```sql
proc sql;
  select *
    from sasuser.stress98
  union
  select *
    from sasuser.stress99;
```

You will learn the details about using each set operator later in this chapter.

Processing a Single Set Operation

PROC SQL evaluates a SELECT statement with one set operation as follows:
1. Each query is evaluated to produce an intermediate (internal) result table.
2. Each intermediate result table then becomes an operand linked with a set operator to form an expression, for example, `Table1 UNION Table2`.
3. PROC SQL evaluates the entire expression to produce a single output result set.

Using Multiple Set Operators

A single SELECT statement can contain more than one set operation. Each additional set operation includes a set operator and a group of query clauses, as shown in the following example:

```sql
proc sql;
  select *
    from table1
  set-operator
  select *
    from table2
  set-operator
  select *
    from table3;
```

This SELECT statement uses two set operators to link together three queries. Regardless of the number of set operations in a SELECT statement, the statement contains only one semicolon, which is placed after the last group of query clauses.

Example

The following PROC SQL step contains two set operators (both are OUTER UNION) that combine three queries:

```sql
proc sql;
  select *
    from sasuser.mechanicslevel1
  outer union
  select *
```
from sasuser.mechanicslevel2
outer union
select *
from sasuser.mechanicslevel3;

Processing Multiple Set Operations
When PROC SQL evaluates a SELECT statement that contains multiple set operations, an additional processing step (step 3 below) is required:

1. Each query is evaluated to produce an intermediate (internal) result table.
2. Each intermediate result table then becomes an operand linked with a set operator to form an expression, for example, `Table1 UNION Table2`.
3. If the set operation contains more than two queries, then the result from the first two queries (enclosed in parentheses in the following examples) becomes an operand for the next set operator and operand. For example:
   - with two set operators: `(Table1 UNION Table2) EXCEPT Table3`
   - with three set operators: `((Table1 UNION Table2) EXCEPT Table3) INTERSECT Table4`

4. PROC SQL evaluates the entire expression to produce a single output result set.

Note: When processing set operators, PROC SQL follows a default order of precedence, unless this order is overridden by parentheses in the expression(s). By default, INTERSECT is evaluated first. OUTER UNION, UNION, and EXCEPT all have the same level of precedence.

Introducing Set Operators
Each of the four set operators EXCEPT, INTERSECT, UNION, and OUTER UNION selects rows and handles columns in a different way, as described below.

Note: In the following chart, Table 1 is the table that is referenced in the first query and Table 2 is the table that is referenced in the second query.

<table>
<thead>
<tr>
<th>Set Operator</th>
<th>Treatment of Rows</th>
<th>Treatment of Columns</th>
<th>Example</th>
</tr>
</thead>
</table>
| EXCEPT       | Selects unique rows from the first table that are not found in the second table. | Overlays columns based on their position in the SELECT clause without regard to the individual column names. | proc sql;
   select *
   from table1
   except
   select *
   from table2; |
| INTERSECT    | Selects unique rows that are common to both tables. | Overlays columns based on their position in the SELECT clause without regard to the individual column names. | proc sql;
   select *
   from table1
   intersect
   select *
   from table2; |
Processing Unique vs. Duplicate Rows

When processing a set operation that displays only unique rows (a set operation that contains the set operator EXCEPT, INTERSECT, or UNION), PROC SQL makes two passes through the data, by default:

1. PROC SQL eliminates duplicate (nonunique) rows in the tables.
2. PROC SQL selects the rows that meet the criteria and, where requested, overlays columns.

For set operations that display both unique and duplicate rows, only one pass through the data (step 2 above) is required.

Combining and Overlaying Columns

You can use a set operation to combine tables that have different numbers of columns and rows or that have columns in a different order.

Three of the four set operators (EXCEPT, INTERSECT, and UNION) combine columns by overlaying them. (The set operator OUTER UNION does not overlay columns.)

By default, the set operators EXCEPT, INTERSECT, and UNION overlay columns based on the relative position of the columns in the SELECT clause. Column names are ignored. You control how PROC SQL maps columns in one table to columns in another table by specifying the columns in the appropriate order in the SELECT clause. The first column specified in the first query’s SELECT clause and the first column specified in the second query’s SELECT clause are overlaid, and so on.
When columns are overlaid, PROC SQL uses the column name from the first table (the table referenced in the first query). If there is no column name in the first table, the column name from the second table is used. When the SELECT clause contains an asterisk (*) instead of a list of column names, the set operation combines the tables (and, if applicable, overlays columns) based on the positions of the columns in the tables.

For example, the following set operation uses the set operator EXCEPT, so columns are overlaid. The SELECT clause in each query uses an asterisk (*), so the columns are overlaid based on their positions in the tables. The first column in table One (X) is overlaid on the first column in table Two (X), and so on.

```
proc sql;
  select *
    from one
  except
  select *
    from two;
```

In order to be overlaid, columns in the same relative position in the two SELECT clauses must have the *same data type*. If they do not, PROC SQL generates a warning message in the SAS log and stops executing. For example, in the tables shown above, if the column One.X had a different data type than column Two.X, the SAS log would display the following error message.

```
Table 4.1  SAS Log

ERROR: Column 1 from the first contributor of EXCEPT
       is not the same type as its counterpart from the second.
```

Next, let’s see how to use the keywords ALL and CORR to modify the default action of the set operators.

## Modifying Results by Using Keywords

To modify the behavior of set operators, you can use either or both of the keywords ALL and CORR immediately following the set operator:

```
proc sql;
  select *
    from table1
  set-operator <all> <corr>
  select *
    from table2;
```

The use of each keyword is described below.
Chapter 4

Using the EXCEPT Set Operator

The set operator EXCEPT does both of the following:

- selects unique rows from the first table (the table specified in the first query) that are not found in the second table (the table specified in the second query)
- overlays columns.

Table 1

Table 2

Let’s see how EXCEPT works when used alone and with the keywords ALL and CORR.

Using the EXCEPT Operator Alone

Suppose you want to display the unique rows in table One that are not found in table Two. The PROC SQL set operation that includes the EXCEPT operator, the tables One and Two, and the output of the set operation are shown below:
Combining Tables Vertically Using PROC SQL  △ Using the EXCEPT Operator Alone  133

```
proc sql;
  select *  
    from one
except
  select *  
    from two;
```

The set operator EXCEPT overlays columns by their position. In this output, the following columns are overlaid:

- the first columns, **One.X** and **Two.X**, both of which are numeric
- the second columns, **One.A** and **Two.B**, both of which are character.

The column names from table **One** are used, so the second column of output is named **A** rather than **B**.

Let’s take a closer look at this example to see exactly how PROC SQL selects rows from table **One** to display in output.

*In the first pass*, PROC SQL eliminates any duplicate rows from the tables. As shown below, there is one duplicate row: in table **One**, the second row is a duplicate of the first row. All remaining rows in table **One** are still candidates in PROC SQL’s selection process.

```
proc sql;
  select *  
    from one
except
  select *  
    from two;
```

*In the second pass*, PROC SQL identifies any rows in table **One** for which there is a matching row in table **Two** and eliminates them. There is one matching row in the two tables, as shown below, which is eliminated.
Using the Keyword ALL with the EXCEPT Operator

To select all rows in the first table (both unique and duplicate) that do not have a matching row in the second table, add the keyword ALL after the EXCEPT set operator. The modified PROC SQL set operation, the tables One and Two, and the output are shown below:

```
proc sql;
  select *
    from one
  except all
  select *
    from two;
```

The output now contains six rows. PROC SQL has again eliminated the one row in table One (the fifth row) that has a matching row in table Two (the fourth row). Remember that when the keyword ALL is used with the EXCEPT operator, PROC SQL
Combining Tables Vertically Using PROC SQL

Using the Keywords ALL and CORR with the EXCEPT Operator

Using the Keyword CORR with the EXCEPT Operator

To display both of the following, add the keyword CORR after the set operator.

- only columns that have the same name
- all unique rows in the first table that do not appear in the second table.

The modified PROC SQL set operation, the tables One and Two, and the output are shown below:

```
proc sql;
select * from one
except corr
select * from two;
```

X is the only column that has the same name in both tables, so X is the only column that PROC SQL examines and displays in the output.

In the first pass, PROC SQL eliminates the second and third rows of table One from the output because they are not unique within the table; they contain values of X that duplicate the value of X in the first row of table One. In the second pass, PROC SQL eliminates the first, fourth, and fifth rows of table One because each contains a value of X that matches a value of X in a row of table Two. The output displays the two remaining rows in table One, the rows that are unique in table One and that do not have a row in table Two that has a matching value of X.

Using the Keywords ALL and CORR with the EXCEPT Operator

If the keywords ALL and CORR are used together, the EXCEPT operator will display all unique and duplicate rows in the first table that do not appear in the second table, and will overlay and display only columns that have the same name. The modified PROC SQL set operation, the tables One and Two, and the output are shown below:
Example: EXCEPT Operator

Let’s see how the EXCEPT operator is used to solve a realistic business problem. Suppose you want to display the names of all new employees of a company. There is no table that contains information for only the new employees, so you will have to use data from the following two tables.

<table>
<thead>
<tr>
<th>Table</th>
<th>Relevant Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasuser.Staffchanges</td>
<td>FirstName, LastName</td>
</tr>
<tr>
<td>Sasuser.Staffmaster</td>
<td>FirstName, LastName</td>
</tr>
</tbody>
</table>

The relationship between these two tables is shown in the diagram below:

![Diagram of tables Sasuser.Staffchanges and Sasuser.Staffmaster]

Once again, PROC SQL looks at and displays only the column that has the same name in the two tables: x. Because the ALL keyword is used, PROC SQL does not eliminate any duplicate rows in table One. Therefore, the second and third rows in table One, which are duplicates of the first row in table One, appear in the output. PROC SQL does eliminate the first, fourth, and fifth rows in table One from the output because for each one of these three rows there is a corresponding row in table Two that has a matching value of x.

As this example shows, when the ALL keyword is used with the EXCEPT operator, a row in table One cannot be eliminated from the output unless it has a separate matching row in table Two. Table One contains three rows in which the value of x is 1, but table Two contains only one row in which the value of x is 1. That one row in table Two causes the first of the three rows in table One that have a matching value of x to be eliminated from the output. However, table Two does not have two additional rows in which the value of x is 1, so the other two rows in table One are not eliminated, and do appear in the output.
Combining Tables Vertically Using PROC SQL

Example: EXCEPT Operator in an In-Line View

The intersection of these two tables includes information for all existing employees who have had changes in job code or salary. The shaded portion, the portion of Sasuser.Staffchanges that does not overlap with Sasuser.Staffmaster, includes information for the people that you want: new employees.

To separate the new employees from the existing employees in Sasuser.Staffchanges, you create a set operation that displays all rows from the first table (Sasuser.Staffchanges) that do not exist in the second table (Sasuser.Staffmaster). The following PROC SQL step solves the problem:

```
proc sql;
  select firstname, lastname
  from sasuser.staffchanges
  except all
  select firstname, lastname
  from sasuser.staffmaster;
```

This PROC SQL set operation includes the operator EXCEPT and the keyword ALL. Although you do not want the output to contain duplicate rows, you already know that there are no duplicates in these two tables. Therefore, ALL is specified to prevent PROC SQL from making an extra pass through the data, which speeds up the processing of this query.

PROC SQL compares only the columns that are specified in the SELECT clauses, and these columns are compared in the order in which they are specified. The output displays the first and last names of the two new employees.

```
<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMY</td>
<td>BRIDESTON</td>
</tr>
<tr>
<td>JIM</td>
<td>POWELL</td>
</tr>
</tbody>
</table>
```

Note: In a set operation that uses the EXCEPT operator, the order in which the tables are listed in the SELECT statement makes a difference. If the tables in this example were listed in the opposite order, the output would display all existing employees who have had no changes in salary or job code.

Example: EXCEPT Operator in an In-Line View

Let’s look at a variation of the preceding set operation. Suppose you want to display the number of existing employees who have had no changes in salary or job code. Once again, the query uses the following tables and columns.

<table>
<thead>
<tr>
<th>Table</th>
<th>Relevant Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasuser.Staffchanges lists information for all new employees and existing employees who have had a change in salary or job code</td>
<td>FirstName, LastName</td>
</tr>
<tr>
<td>Sasuser.Staffmaster lists information for all existing employees</td>
<td>FirstName, LastName</td>
</tr>
</tbody>
</table>

The following PROC SQL query solves this problem:

```
proc sql;
  select count(*) label='No. of Persons'
```
Using the INTERSECT Set Operator

The set operator INTERSECT does both of the following:
- selects unique rows that are common to both tables
- overlays columns.

Let's see how INTERSECT works when used alone and with the keywords ALL and CORR.

Using the INTERSECT Operator Alone

The INTERSECT operator compares and overlays columns in the same way as the EXCEPT operator, by column position instead of column name. However, INTERSECT selects rows differently and displays in output the unique rows that are common to both tables. The following PROC SQL set operation uses the INTERSECT operator to combine the tables One and Two, which were introduced previously:
Combining Tables Vertically Using PROC SQL

Using the Keyword CORR with the INTERSECT Operator

Tables *One* and *Two* have only one unique row in common and this row is displayed in the output. (This is the same row that was eliminated in the earlier example that contained the EXCEPT operator.)

**Using the Keyword ALL with the INTERSECT Operator**

Adding the keyword *ALL* to the preceding PROC SQL query prevents PROC SQL from making an extra pass through the data. If there were any rows common to tables *One* and *Two* that were duplicates of other common rows, they would also be included in output. However, as you have seen, there is only one common row in these tables. The modified PROC SQL query, the tables *One* and *Two*, and the output are shown below:

To display the unique rows that are common to the two tables based on the column name instead of the column position, add the *CORR* keyword to the PROC SQL set operation. The modified query, the tables *One* and *Two*, and the output are shown below:
Using the Keywords ALL and CORR with the INTERSECT Operator

If the keywords ALL and CORR are used together, the INTERSECT operator will display all unique and nonunique (duplicate) rows that are common to the two tables, based on columns that have the same name. The modified query, the tables One and Two, and the output are shown below:

```
proc sql;
  select *
  from one
  intersect all corr
  select *
  from two;
```

PROC SQL examines and displays only the column with the same name, \( x \). There are three common rows across the two tables, which are highlighted above, and these are the three rows that are displayed in the output.

Note that each of the tables contains at least one other row that duplicates a value of \( x \) in one of the common rows. For example, in the second and third rows in table One, the value of \( x \) is 1, as in one of the common rows. However, in order to be considered a
Combining Tables Vertically Using PROC SQL

Example: INTERSECT Operator

Now that you have seen how the INTERSECT set operator works with very small tables, let's use INTERSECT in a realistic business problem. Suppose you want to display the names of the existing employees who have changed their salary or job code. (This query is the opposite of the query that you solved with the EXCEPT operator.) Once again, you will use the following tables.

<table>
<thead>
<tr>
<th>Table</th>
<th>Relevant Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasuser.Staffchanges</td>
<td>lists information for all new employees and existing employees who have had a change in salary or job code</td>
</tr>
<tr>
<td>Sasuser.Staffmaster</td>
<td>lists information for all existing employees</td>
</tr>
</tbody>
</table>

The relationship between these two tables is shown in the diagram below:

As shown in the earlier example with EXCEPT, the intersection of these two tables includes information for all existing employees who have had changes in job code or salary. It is the intersection of these two tables, shaded above, that you want to display.

To display the unique rows that are common to both tables, you use a PROC SQL set operation that contains INTERSECT. It is known that these tables contain no duplicates, so ALL is used to speed up query processing. The PROC SQL set operation is shown below:

```
proc sql;
    select firstname, lastname
    from sasuser.staffchanges
    intersect all
    select firstname, lastname
    from sasuser.staffmaster;
```

Note: In this PROC SQL step, which contains just one INTERSECT set operator, the order in which you list the tables in the SELECT statement does not make a difference. However, in a more complex PROC SQL step that contains multiple stacked INTERSECT set operators, it is important to think through the table order carefully, depending on when you want the non-matches to be eliminated.

The output shows that there are four existing employees who have changed their salary or job code.
Using the UNION Set Operator

The set operator UNION does both of the following:
- selects *unique rows* from both tables together
- overlays columns.

Let’s see how UNION works when used alone and with the keywords ALL and CORR.

### Using the UNION Operator Alone

To display all rows from the tables One and Two that are *unique* in the combined set of rows from both tables, use a PROC SQL set operation that includes the UNION operator:

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

With the UNION operator, PROC SQL first concatenates and sorts the rows from the two tables, then eliminates any duplicate rows. In this example, two rows are
eliminated: the second row in table One is a duplicate of the first row, and the fourth row in table Two matches the fifth row in table One. All remaining rows, the unique rows, are included in the output. The columns are overlaid by position.

**Using the Keyword ALL with the UNION Operator**

When the keyword ALL is added to the UNION operator, the output displays all rows from both tables, both unique and duplicate. The modified PROC SQL set operation, the tables One and Two, and the new output are shown below:

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

When the ALL keyword is used, PROC SQL does not remove duplicates or sort the rows. The output now includes the two duplicate rows that were eliminated in the previous example: the second row in table One and the fourth row in table Two. Note that the rows are in a different order in this output than they were in the output from the previous set operation.

**Using the Keyword CORR with the UNION Operator**

To display all rows from the tables One and Two that are unique in the combined set of rows from both tables, based on columns that have the same name rather than the same position, add the keyword CORR after the set operator. The modified query, the tables One and Two, and the output are shown below:
proc sql;
select *
from one
union corr
select *
from two;

X is the only column name that is common to both tables, so X is the only column that PROC SQL examines and displays in the output. In the combined set of rows from the two tables, there are duplicates of the values 1, 2, and 3, and these duplicate rows are eliminated from the output. The output displays the six unique values of X.

---

Using the Keywords ALL and CORR with the UNION Operator

If the keywords ALL and CORR are used together, the UNION operator will display all rows in the two tables both unique and duplicate, based on the columns that have the same name. In this example, the output displays all 12 values for X, the one column that has the same name in both tables.

proc sql;
select *
from one
union all corr
select *
from two;
Example: UNION Operator

Let’s see how the UNION operator can be used to solve a realistic business problem. Suppose you are generating a report based on data from a health clinic. You want to display the results of individual patient stress tests taken in 1998, followed by the results from stress tests taken in 1999. To do this, you will use the UNION operator to combine the tables Sasuser.Stress98 and Sasuser.Stress99. These two tables are similar in structure:

- both tables contain nine columns that have the same names
- each row contains data for an individual patient.

You are not sure whether the tables contain duplicate records, but you do not want duplicates in your output. Because the tables have the same column structure, you can overlay the columns by position and the CORR keyword is not necessary. The PROC SQL set operation and output are shown below:

```sql
proc sql;
select *
  from sasuser.stress98
union
select *
  from sasuser.stress99;
```

Tip: If you can determine that these tables have no duplicate records, you could add the keyword ALL to speed up processing by avoiding an extra pass through the data.

Example: UNION Operator and Summary Functions

Let’s look at another realistic business problem, to see how summary functions can be used with a set operator (in this case, UNION). Suppose you want to display the following summarized data for members of a frequent-flyer program: total points earned, total points used, and total miles traveled. All three values can be calculated from columns in the table Sasuser.Frequentflyers by using summary functions.

You might wonder why set operations are needed when only one table is involved. If you wanted to display the three summarized values horizontally, in three separate columns, you could solve the problem without a set operation, using the following simple SELECT statement:
Using the OUTER UNION Set Operator

The set operator OUTER UNION *concatenates* the results of the queries by...
Combining Tables Vertically Using PROC SQL

- selecting all rows (both unique and nonunique) from both tables
- not overlaying columns.

Table 1

Table 2

Let’s see how OUTER UNION works when used alone and with the keyword CORR. The ALL keyword is not used with OUTER UNION because this operator’s default action is to include all rows in output.

### Using the OUTER UNION Operator Alone

Suppose you want to display all rows from both of the tables One and Two, without overlaying columns. The PROC SQL set operation that includes the OUTER UNION operator, the two tables, and the output are shown below:

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

In the output, the columns have not been overlaid. Instead, all four columns from both tables are displayed. Each row of output contains missing values in the two columns that correspond to the other table.

### Using the Keyword CORR with the OUTER UNION Operator

The output from the preceding set operation contains two columns with the same name. To overlay the columns with a common name, add the CORR keyword to the set operation:
**Example: OUTER UNION Operator**

There are many business situations that require two or more tables to be concatenated. For example, suppose you want to display the employee numbers, job codes, and salaries of all mechanics working for an airline. The mechanic job has three levels and there is a separate table containing data for the mechanics at each level: `Sasuser.Mechanicslevel1`, `Sasuser.Mechanicslevel2`, and `Sasuser.Mechanicslevel3`. These tables all contain the same three columns.

The following PROC SQL step uses two OUTER UNION operators to concatenate the tables, and the CORR keyword to overlay the columns that have common names:

```sql
proc sql;
   select *
      from one
   outer union corr
      select *
      from two;
```

The output from the modified set operation contains only three columns, because the two columns named `X` are overlaid.
Combining Tables Vertically Using PROC SQL

Program 1: PROC SQL OUTER UNION Set Operation with CORR

```sas
proc sql;
   create table three as
      select * from one
         outer union corr
         select * from two;
quit;
```

Program 2: DATA Step, SET Statement, and PROC PRINT Step

```sas
data three;
   set one two;
run;
proc print data=three noobs;
run;
```
These two programs create the same table as output, as shown below.

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th></th>
<th>X</th>
<th>B</th>
<th></th>
<th>X</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td>1</td>
<td>x</td>
<td></td>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td>2</td>
<td>y</td>
<td></td>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td></td>
<td>3</td>
<td>z</td>
<td></td>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td></td>
<td>3</td>
<td>v</td>
<td></td>
<td>2</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>v</td>
<td></td>
<td>5</td>
<td>w</td>
<td></td>
<td>3</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>o</td>
<td></td>
<td>6</td>
<td>g</td>
<td></td>
<td>4</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
<td>2</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td></td>
<td>3</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>v</td>
<td></td>
<td>3</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>w</td>
<td></td>
<td>5</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When tables have a same-named column, the PROC SQL outer union will *not* produce the same output unless the keyword CORR is also used. CORR causes the same-named columns (in this example, the two columns named X) to be overlaid; without CORR, the OUTER UNION operator will include both of the same-named columns in the result set. The DATA step program will generate only one column X.

The two concatenation techniques shown above also vary in efficiency. A PROC SQL set operation generally requires more computer resources but may be more convenient and flexible than the DATA step equivalent.
Text Summary

Understanding Set Operations

A set operation combines tables or views vertically (one on top of the other) by combining the results of two queries. A set operation is a SELECT statement that contains

- two groups of query clauses (each group beginning with a SELECT clause)
- one of the set operators EXCEPT, INTERSECT, UNION, and OUTER UNION
- optionally, one or both of the keywords ALL and CORR (CORRESPONDING) as modifiers.

A single SELECT statement can contain multiple set operations.

When processing a set operation that displays only unique rows (a set operation that contains the set operator EXCEPT, INTERSECT, or UNION), PROC SQL makes two passes through the data, by default. For set operations that display both unique and duplicate rows, only one pass through the data is required.

For the set operators EXCEPT, INTERSECT, and UNION, columns are overlaid based on the relative position of the columns in the SELECT clause rather than by column name. In order to be overlaid, columns in the same relative position in the two SELECT clauses must have the same data type.

One or both keywords can be used to modify the default action of a set operator.

Using the EXCEPT Set Operator

The set operator EXCEPT selects unique rows from the first table (the table specified in the first query) that are not found in the second table (the table specified in the second query) and overlays columns. This set operation can be modified by using either or both of the keywords ALL and CORR.

Using the INTERSECT Set Operator

The set operator INTERSECT selects unique rows that are common to both tables and overlays columns. This set operation can be modified by using either or both of the keywords ALL and CORR.

Using the UNION Set Operator

The set operator UNION selects unique rows from both tables together and overlays columns. This set operation can be modified by using either or both of the keywords ALL and CORR.
Using the OUTER UNION Set Operator

The set operator OUTER UNION concatenates the results of two queries by selecting all rows (both unique and nonunique) from both tables and not overlaying columns. This set operation can be modified by using the keyword CORR.

Comparing Outer Unions and Other SAS Techniques

A PROC SQL set operation that uses the OUTER UNION set operator is not the only way to concatenate tables in SAS. Other SAS techniques can be used, such as a program that consists of a DATA step, a SET statement, and a PROC PRINT step.

Syntax

PROC SQL;
    SELECT column-1<, ... column-n>
    FROM table-1 | view-1<, ... table-n | view-n>
    <optional query clauses>
    set-operator <ALL> <CORR>
    SELECT column-1<, ... column-n>
    FROM table-1 | view-1<, ... table-n | view-n>
    <optional query clauses>;
QUIT;

Sample Program

proc sql;
    select firstname, lastname
    from sasuser.staffchanges
    intersect all
    select firstname, lastname
    from sasuser.staffmaster;
quit;

Points to Remember

- Regardless of the number of set operations in a SELECT statement, the statement contains only one semicolon, which is placed after the last group of query clauses.
- In order to be overlaid, columns must have the same data type.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which statement is false with respect to a set operation that uses the EXCEPT, UNION, or INTERSECT set operator without a keyword?
   a. Column names in the result set are determined by the first table.
   b. To be overlaid, columns must be of the same data type.
   c. To be overlaid, columns must have the same name.
Combining Tables Vertically Using PROC SQL

2 The keyword ALL cannot be used with which of the following set operators?
   a) EXCEPT
   b) INTERSECT
   c) UNION
   d) OUTER UNION

3 Which PROC SQL step combines the tables Summer and Winter to produce the output displayed below?

```
<table>
<thead>
<tr>
<th>Month</th>
<th>Temp</th>
<th>Precip</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>78</td>
<td>0.05</td>
</tr>
<tr>
<td>8</td>
<td>85</td>
<td>0.04</td>
</tr>
<tr>
<td>9</td>
<td>83</td>
<td>0.15</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Mo</th>
<th>Temp</th>
<th>Precip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
<td>0.15</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>0.17</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>0.20</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Month</th>
<th>Temp</th>
<th>Precip</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>78</td>
<td>0.05</td>
</tr>
<tr>
<td>8</td>
<td>65</td>
<td>0.04</td>
</tr>
<tr>
<td>9</td>
<td>83</td>
<td>0.15</td>
</tr>
</tbody>
</table>
```

a) proc sql;
   select *
     from summer
     intersect all
     select *
     from winter;

b) proc sql;
   select *
     from summer
     outer union
     select *
     from winter;

c) proc sql;
   select *
     from summer
     union corr
     select *
     from winter;

d) proc sql;
   select *
     from summer
     union
     select *
     from winter;
4 Which PROC SQL step combines tables but does not overlay any columns?

a proc sql;
       select *
       from groupa
       outer union
       select *
       from groupb;

b proc sql;
       select *
       from groupa as a
       outer union corr
       select *
       from groupb as b;

c proc sql;
       select coalesce(a.obs, b.obs)
           label='Obs', med, duration
       from groupa as a
       full join
       groupb as b
       on a.obs=b.obs;

d proc sql;
       select *
       from groupa as a
       intersect
       select *
       from groupb as b;

5 Which statement is false regarding the keyword CORRESPONDING?

a It cannot be used with the keyword ALL.
b It overlays columns by name, not by position.
c When used in EXCEPT, INTERSECT, and UNION set operations, it removes any columns not found in both tables.
d When used in OUTER UNION set operations, it causes same-named columns to be overlaid.

6 Which PROC SQL step generates the following output from the tables *Dogs* and *Pets*?

<table>
<thead>
<tr>
<th>Dogs</th>
<th>Pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Price</td>
</tr>
<tr>
<td>FIFI</td>
<td>$101</td>
</tr>
<tr>
<td>GEORGE</td>
<td>$75</td>
</tr>
<tr>
<td>SPARKY</td>
<td>$136</td>
</tr>
<tr>
<td>TRUFFLE</td>
<td>$250</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a proc sql;
    select name, price
    from pets
    except all
    select *
    from dogs;

b proc sql;
    select name, price
    from pets
    except
    select *
    from dogs;

c proc sql;
    select name, price
    from pets
    except corr all
    select *
    from dogs;

d proc sql;
    select *
    from dogs
    except corr
    select name, price
    from pets;

7 The PROG1 and PROG2 tables list students who took the PROG1 and PROG2 courses, respectively. Which PROC SQL step will give you the names of the students who took only the PROG1 class?

\[
\begin{array}{|c|c|}
\hline
\text{FName} & \text{LName} \\
\hline
\text{Pete} & \text{Henry} \\
\hline
\text{Mary} & \text{Johnson} \\
\hline
\text{Alex} & \text{Kinsley} \\
\hline
\text{Dori} & \text{O'Neil} \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{FName} & \text{LName} \\
\hline
\text{Clara} & \text{Addams} \\
\hline
\text{Pete} & \text{Henry} \\
\hline
\text{Dori} & \text{O'Neil} \\
\hline
\text{Cindy} & \text{Phillips} \\
\hline
\text{Mandi} & \text{Young} \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{FName} & \text{LName} \\
\hline
\text{Alex} & \text{Kinsley} \\
\hline
\text{Mary} & \text{Johnson} \\
\hline
\end{array}
\]
8 Which PROC SQL step will return the names of all the students who took PROG1, PROG2, or both classes?

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pete</td>
<td>Henry</td>
</tr>
<tr>
<td>Mary</td>
<td>Johnson</td>
</tr>
<tr>
<td>Alex</td>
<td>Kinsley</td>
</tr>
<tr>
<td>Dorl</td>
<td>O'Neil</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clara</td>
<td>Addams</td>
</tr>
<tr>
<td>Pete</td>
<td>Henry</td>
</tr>
<tr>
<td>Dorl</td>
<td>O'Neil</td>
</tr>
<tr>
<td>Cindy</td>
<td>Phillips</td>
</tr>
<tr>
<td>Mandi</td>
<td>Young</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>Kinsley</td>
</tr>
<tr>
<td>Cindy</td>
<td>Phillips</td>
</tr>
<tr>
<td>Clara</td>
<td>Addams</td>
</tr>
<tr>
<td>Dorl</td>
<td>O'Neil</td>
</tr>
<tr>
<td>Mandi</td>
<td>Young</td>
</tr>
<tr>
<td>Mary</td>
<td>Johnson</td>
</tr>
<tr>
<td>Pete</td>
<td>Henry</td>
</tr>
</tbody>
</table>

a proc sql;
select fname, lname
from prog1
intersect
select fname, lname
from prog2;

b proc sql;
select fname, lname
from prog1
outer union corr
select fname, lname
from prog2;

c proc sql;
select fname, lname
from prog1
union
select fname, lname
from prog2;

d proc sql;
select fname, lname
from prog1
except corr
select fname, lname
from prog2;
9 Which PROC SQL step will return the names of all the students who took both the PROG1 and PROG2 classes?

- a proc sql;
  select fname, lname
  from prog1
  union
  select fname, lname
  from prog2;

- b proc sql;
  select fname, lname
  from prog1
  except corr
  select fname, lname
  from prog2;

- c proc sql;
  select fname, lname
  from prog1
  intersect all
  select fname, lname
  from prog2;

- d proc sql;
  select fname, lname
  from prog1
  union corr
  select fname, lname
  from prog2;

10 Which PROC SQL step will generate the same results as the following DATA step?

data allstudents;
  set prog1 prog2;
  by lname;
run;
proc print noobs;
run;
a proc sql;
  select fname, lname
  from prog1
  outer union corr
  select fname, lname
  from prog2
  order by lname;

b proc sql;
  select fname, lname
  from prog1
  union
  select fname, lname
  from prog2
  order by lname;

c proc sql;
  select fname, lname
  from prog2
  outer union
  select fname, lname
  from prog1
  order by lname;

d proc sql;
  select fname, lname
  from prog2
  union corr
  select fname, lname
  from prog1
  order by lname;
# Chapter 5

Creating and Managing Tables Using PROC SQL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>161</td>
</tr>
<tr>
<td>Overview</td>
<td>161</td>
</tr>
<tr>
<td>Introduction</td>
<td>161</td>
</tr>
<tr>
<td>Objectives</td>
<td>162</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>162</td>
</tr>
<tr>
<td>Understanding Methods of Creating Tables</td>
<td>162</td>
</tr>
<tr>
<td>Creating an Empty Table by Defining Columns</td>
<td>163</td>
</tr>
<tr>
<td>Example</td>
<td>164</td>
</tr>
<tr>
<td>Specifying Data Types</td>
<td>165</td>
</tr>
<tr>
<td>Specifying Column Widths</td>
<td>166</td>
</tr>
<tr>
<td>Specifying Column Modifiers</td>
<td>167</td>
</tr>
<tr>
<td>Example</td>
<td>167</td>
</tr>
<tr>
<td>Displaying the Structure of a Table</td>
<td>168</td>
</tr>
<tr>
<td>Example</td>
<td>169</td>
</tr>
<tr>
<td>Creating an Empty Table That Is Like Another Table</td>
<td>169</td>
</tr>
<tr>
<td>Example</td>
<td>169</td>
</tr>
<tr>
<td>Specifying a Subset of Columns from a Table</td>
<td>170</td>
</tr>
<tr>
<td>Example</td>
<td>171</td>
</tr>
<tr>
<td>Creating a Table from a Query Result</td>
<td>172</td>
</tr>
<tr>
<td>Example</td>
<td>172</td>
</tr>
<tr>
<td>Copying a Table</td>
<td>173</td>
</tr>
<tr>
<td>Example</td>
<td>174</td>
</tr>
<tr>
<td>Inserting Rows of Data into a Table</td>
<td>174</td>
</tr>
<tr>
<td>Inserting Rows by Using the SET Clause</td>
<td>175</td>
</tr>
<tr>
<td>Example</td>
<td>176</td>
</tr>
<tr>
<td>Inserting Rows by Using the VALUES Clause</td>
<td>177</td>
</tr>
<tr>
<td>Example</td>
<td>179</td>
</tr>
<tr>
<td>Inserting Rows from a Query Result</td>
<td>180</td>
</tr>
<tr>
<td>Example</td>
<td>181</td>
</tr>
<tr>
<td>Creating a Table That Has Integrity Constraints</td>
<td>182</td>
</tr>
<tr>
<td>General Integrity Constraints</td>
<td>183</td>
</tr>
<tr>
<td>Referential Integrity Constraints</td>
<td>183</td>
</tr>
<tr>
<td>Creating a Constraint in a Column Specification</td>
<td>184</td>
</tr>
<tr>
<td>Example</td>
<td>186</td>
</tr>
<tr>
<td>Creating a Constraint by Using a Constraint Specification</td>
<td>186</td>
</tr>
<tr>
<td>Example</td>
<td>189</td>
</tr>
<tr>
<td>Handling Errors in Row Insertions</td>
<td>190</td>
</tr>
<tr>
<td>Example</td>
<td>190</td>
</tr>
<tr>
<td>Using the UNDO POLICY= Option to Control UNDO Processing</td>
<td>191</td>
</tr>
<tr>
<td>Example</td>
<td>192</td>
</tr>
<tr>
<td>Displaying Integrity Constraints for a Table</td>
<td>193</td>
</tr>
<tr>
<td>Example</td>
<td>193</td>
</tr>
</tbody>
</table>
Updating Values in Existing Table Rows 194

Updating Rows by Using the Same Expression 195
Example 195
Updating Rows by Using Different Expressions 197
Example 198

How PROC SQL Updates Rows Based on a CASE Expression 199
How the Case Operand Works 199

Updating Rows by Using the CASE Expression without a Case Operand 199
Example 199
Updating Rows by Using the CASE Expression with a Case Operand 201
Example 201

Using the CASE Expression in the SELECT Statement 202
Example 202

Deleting Rows in a Table 202
Example 203

Altering Columns in a Table 204

Adding Columns to a Table 205
Example 205
Dropping Columns from a Table 206
Example 206
Modifying Columns in a Table 207
Example 208

Adding, Dropping, and Modifying Columns in a Single Statement 209
Example 209

Dropping Tables 210
Example 210

Summary 211

Text Summary 211
Understanding Methods of Creating Tables 211
Creating an Empty Table by Defining Columns 211
Displaying the Structure of a Table 211
Creating an Empty Table That Is Like Another Table 211
Creating a Table from a Query Result 211
Inserting Rows of Data into a Table 211
Creating a Table That Has Integrity Constraints 212
Handling Errors in Row Insertions 212
Displaying Integrity Constraints for a Table 212
Updating Values in Existing Table Rows 212
Deleting Rows in a Table 212
Altering Columns in a Table 212
Dropping Tables 212

Syntax 213

Sample Programs 214
Creating an Empty Table by Defining Columns 214
Creating an Empty Table That Is Like Another Table 214
Creating a Table from a Query Result 214
Displaying the Structure of a Table 215
Inserting Rows into a Table by Specifying Column Names and Values 215
Inserting Rows into a Table by Specifying Lists of Values 215
Inserting Rows into a Table from a Query Result 215
Creating a Table That Has Integrity Constraints 215
Displaying Integrity Constraints for a Table 216
Updating Rows in a Table Based on an Expression 216
Updating Rows in a Table by Using a CASE Expression 216
Overview

Introduction

By using PROC SQL, you can create, modify, and drop (delete) tables quickly and efficiently. Many PROC SQL statements are quite versatile, enabling you to perform the same action in several different ways. For example, there are three methods of creating a table by using the CREATE TABLE statement:

- creating an empty table (a table without rows) by defining columns
- creating an empty table that has the same columns and attributes as another table
- creating a table from a query result.

The following PROC SQL step uses the CREATE TABLE statement to create an empty table by defining columns, and uses the DESCRIBE TABLE statement to display information about the table’s structure in the SAS log:

```sql
proc sql;
create table work.discount
(Destination char(3),
 BeginDate num Format=date9.,
 EndDate num format=date9.,
 Discount num);
describe table work.discount;
```

<table>
<thead>
<tr>
<th>Table 5.1</th>
<th>SAS Log</th>
</tr>
</thead>
</table>
| 1 proc sql;
  2 create table work.discount
  3 (Destination char(3),
    4 BeginDate num Format=date9.,
    5 EndDate num format=date9.,
    6 Discount num);
  describe table work.discount;

NOTE: Table WORK.DISCOUNT created, with 0 rows and 4 columns.
NOTE: SQL table WORK.DISCOUNT was created like:

```sql
create table WORK.DISCOUNT( bufsize=4096 )
( Destination char(3),
  BeginDate num Format=DATE9.,
  EndDate num format=DATE9.,
  Discount num )
```

In this chapter, you will learn to create and manage tables by using the PROC SQL statements shown above, and many others.
Objectives

In this chapter, you learn to
- create a table that has no rows by specifying columns and values
- create a table that has no rows by copying the columns and attributes from an existing table
- create a table that has rows, based on a query result
- display the structure of a table in the SAS log
- insert rows into a table by listing values
- insert rows into a table by specifying column-value pairs
- insert rows into a table, based on a query result
- create a table that has integrity constraints
- set the UNDO_POLICY option to control how PROC SQL handles errors in table insertions and updates
- display the integrity constraints of a table in the SAS log
- update values in existing rows of a table by using one expression and by using conditional processing with multiple expressions
- delete rows in a table
- add, modify, or drop (delete) columns in a table
- drop (delete) entire tables.

Prerequisites

Before beginning this chapter, you should complete the following chapters:
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79.

Understanding Methods of Creating Tables

You can use PROC SQL to create a table in three ways. The CREATE TABLE statement is used for all three methods, although the syntax of the statement varies for each method.
Creating an Empty Table by Defining Columns

Sometimes you want to create a new table that is unlike any existing table. In this case, you need to define all of the columns and attributes that the table will have. To accomplish this, use the `CREATE TABLE` statement that includes `column specifications` for the columns that you want to include. This statement creates a table without rows (an empty table).

Note: In addition, integrity constraints can be specified in the `CREATE TABLE` statement. Integrity constraints are discussed later in this chapter.
General form, basic CREATE TABLE statement with column specifications:

```
CREATE TABLE table-name
    (column-specification-1<,
     ...column-specification-n>);
```

where

- `table-name` specifies the name of the table to be created.
- `column-specification` specifies a column to be included in the table, and consists of
  
  - `column-definition <column-constraint-1<, ...column-constraint-n>>`<MESSAGE='message-string'<MSGTYPE=message-type>>
  
  where
  
  - `column-definition` consists of the following:
    
    - `column-name data-type<(column-width)><column-modifier-1<...column-modifier-n>>`
    
    - `column-name` specifies the name of the column. The column name is stored in the table in the same case that is used in `column-name`.
    
    - `data-type` is enclosed in parentheses and specifies one of the following: CHARACTER (or CHAR) | VARCHAR | INTEGER (or INT) | SMALLINT | DECIMAL (or DEC) | NUMERIC (or NUM) | FLOAT | REAL | DOUBLE PRECISION | DATE.
    
    - `column-width` which is enclosed in parentheses, is an integer that specifies the width of the column. (PROC SQL processes this value only for the CHARACTER and VARCHAR data types.)
    
    - `column-modifier` is one of the following: INFORMAT= | FORMAT= | LABEL= . More than one column-modifier may be specified.
    
    - `column-constraint` specifies an integrity constraint.
    
    - `MESSAGE=` and `MSGTYPE=` specify an error message that is related to an integrity constraint. (Integrity constraints, the column-constraint, `MESSAGE=` and `MSGTYPE=` are not elaborated here, but are discussed in detail later in this chapter.)

**Note:** The entire set of `column-specifications` must be enclosed in parentheses. Multiple `column-specifications` must be separated by commas. Elements within a `column-specification` must be separated by spaces.

**Example**

Suppose you want to create the temporary table `Work.Discount`, which contains data about discounts that are offered by an airline. There is no existing table that contains the four columns (and column attributes) that you would like to include: `Destination`,
**BeginDate, EndDate, and Discount.** You use the following PROC SQL step to create the table, based on column definitions that you specify:

```sql
proc sql;
create table work.discount
    (Destination char(3),
     BeginDate num format=date9.,
     EndDate num format=date9.,
     Discount num);
```

The SAS log confirms that the table has been created.

<table>
<thead>
<tr>
<th>Table 5.3 SAS Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTE:</strong> Table WORK.DISCOUNT created, with 0 rows and 4 columns.</td>
</tr>
</tbody>
</table>

*Tip:* In this example and all other examples in this chapter, you are instructed to save your data to a temporary table (in the library **Work**) that will be deleted at the end of the SAS session. To save the table permanently in a different library, use the appropriate libref instead of the libref **Work** in the CREATE TABLE clause.

In the next few sections, you will learn more about specifying data types and column modifiers in a column specification.

*Note:* You will learn to insert rows of data in a table later in this chapter.

---

**Specifying Data Types**

When you create a table by defining columns, you must specify a *data type* for each column, following the column name:

```
column-name  data-type (<column-width>) <column-modifier-1>
<...column-modifier-n>>
```

For example, the following PROC SQL step (shown also in the previous section) defines four columns: one character column (**Destination**) and three numeric columns (**BeginDate, EndDate, and Discount**):

```sql
proc sql;
create table work.discount
    (Destination char(3),
     BeginDate num format=date9.,
     EndDate num format=date9.,
     Discount num);
```

SAS tables use two data types: *numeric* and *character*. However, PROC SQL supports additional data types (many, but not all, of the data types that SQL-based databases support). Therefore, in the CREATE TABLE statement, you can specify any of 10 different data types. When the table is created, PROC SQL converts the supported data types that are not SAS data types to either numeric or character format.
Table 5.4  Character Data Types Supported by PROC SQL

<table>
<thead>
<tr>
<th>Specified Data Type</th>
<th>SAS Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER (or CHAR)</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>CHARACTER</td>
</tr>
</tbody>
</table>

Table 5.5  Numeric Data Types Supported by PROC SQL

<table>
<thead>
<tr>
<th>Specified Data Type</th>
<th>Description</th>
<th>SAS Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC (or NUM)</td>
<td>floating-point</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>DECIMAL (or DEC)</td>
<td>floating-point</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>FLOAT</td>
<td>floating-point</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>REAL</td>
<td>floating-point</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>floating-point</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>INTEGER (or INT)</td>
<td>integer</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>integer</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
<td>NUMERIC with a DATE.7 informat and format</td>
</tr>
</tbody>
</table>

The following PROC SQL step is very similar to the previous example. The only difference is that this step specifies three supported data types other than CHAR and NUM: VARCHAR, DATE, and FLOAT.

```sql
proc sql;
create table work.discount2
(Destination varchar(3),
 BeginDate date,
 EndDate date,
 Discount float);
```

PROC SQL will convert these data types to either character or numeric, as indicated in the charts above. Therefore, the table Work.Discount2 (created by this PROC SQL step) and Work.Discount (created by the previous PROC SQL step) will contain identical columns.

By supporting data types other than SAS data types, PROC SQL can save you time. In many cases, you can copy native code from an implementation of SQL that is external to SAS without having to modify the data types.

**Specifying Column Widths**

In SAS, the default column width for both character and numeric columns is 8 bytes. However, character and numeric data values are stored differently:

- **Character data** is stored one character per byte.
- **Numeric data** is stored as floating point numbers in real binary representation, which allows for 16- or 17-digit precision within 8 bytes.

PROC SQL allows you to specify a column width for character columns but not for numeric columns.
Note: PROC SQL allows the WIDTH and NDEC (decimal places) arguments to be included in the column specification for the DECIMAL, NUMERIC, and FLOAT data types. However, PROC SQL ignores this specification and uses the SAS defaults.

In a column specification, the column width follows the data type and is specified as an integer enclosed in parentheses:

    column-name data-type <(column-width)> <column-modifier-1
    <...column-modifier-n>>

In the following PROC SQL step, the first column specification indicates a column width of 3 for the character column Destination:

    proc sql;
    create table work.discount
    (Destination char(3),
     BeginDate num format=date9.,
     EndDate num format=date9.,
     Discount num);

Because the last three columns are numeric, no width is specified and these columns will have the default column width of 8 bytes.

---

**Specifying Column Modifiers**

In the CREATE TABLE statement, a column specification may optionally include one or more of the following SAS column modifiers: INFORMAT=, FORMAT=, and LABEL=. Column modifiers, if used, are specified at the end of the column specification.

    column-name data-type <(column-width)> <...column-modifier-1
    <...column-modifier-n>>

Note: A fourth SAS column modifier, LENGTH=, is not allowed in a CREATE TABLE clause. It may be used in a SELECT clause.

---

**Example**

The following PROC SQL step creates the table Work.Departments by specifying 4 columns. The column modifiers LABEL= and FORMAT= are used to specify additional column attributes.

    proc sql;
    create table work.departments
    (Dept varchar(20) label='Department',
     Code integer label='Dept Code',
     Manager varchar(20),
     AuditDate num format=date9.);

The SAS log verifies that the table was created.
Displaying the Structure of a Table

Sometimes you want to look at the structure (the columns and column attributes) of a table that you created or of a table that was created by someone else. When you create a table, the CREATE TABLE statement writes a message to the SAS log, which indicates the number of rows and columns in the table that was created. However, that message does not contain information about column attributes.

If you are working with an existing table that contains rows of data, you can use a PROC SQL query to generate a report that shows all of the columns in a table. However, the report does not list the column attributes, and a PROC SQL query will not generate output for an empty table.

To display a list of columns and column attributes for one or more tables in the SAS log, regardless of whether the tables contain rows of data, you can use the DESCRIBE TABLE statement in PROC SQL.

General form, DESCRIBE TABLE statement:

```
DESCRIBE TABLE table-name-1<, ... table-name-n>
```

where

- `table-name` specifies the table to be described as one of the following:
  - a one-level name
  - a two-level `libref.table` name
  - a physical pathname that is enclosed in single quotation marks.

The DESCRIBE TABLE statement writes a CREATE TABLE statement that includes column definitions to the SAS log for the specified table, regardless of how the table was originally created. For example, if the DESCRIBE TABLE statement specifies a table that was created with the DATA step, a CREATE TABLE statement will still be displayed.

Note: The DESCRIBE TABLE statement also displays information about any indexes that are defined on a table. You can learn more about using the DESCRIBE TABLE statement to display information about indexes in Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221.

Tip: As an alternative to the DESCRIBE TABLE statement, you can use other SAS procedures, like PROC CONTENTS, to list a table’s columns and column attributes. PROC CONTENTS generates a report instead of writing a message to the SAS log, as the DESCRIBE TABLE statement does. You can learn more about using PROC CONTENTS in Chapter 13, “Creating Samples and Indexes,” on page 451.
Example

Earlier in this chapter, the empty table `Work.Discount` was created by using the CREATE TABLE statement and column specifications shown below:

```sql
proc sql;
create table work.discount
(Destination char(3),
 BeginDate num format=date9.,
 EndDate num format=date9.,
 Discount num);
```

The following DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the table `Work.Discount`:

```sql
proc sql;
describe table work.discount;
```

Table 5.7  SAS Log

```
NOTE: SQL table WORK.DISCOUNT was created like:
create table WORK.DISCOUNT( bufsize=4096 )
  (Destination char(3),
   BeginDate num format=DATE9.,
   EndDate num format=DATE9.,
   Discount num)
```

Creating an Empty Table That Is Like Another Table

Sometimes you want to create a new table that has the same columns and attributes as an existing table, but has no rows. To create an empty table that is like another table, use a CREATE TABLE statement with a LIKE clause.

General form, CREATE TABLE statement with a LIKE clause:

```sql
CREATE TABLE table-name
   LIKE table-1;
```

where

- `table-name` specifies the name of the table to be created.
- `table-1` specifies the table whose columns and attributes will be copied to the new table.

Example

Suppose you want to create a new table, `Work.Flightdelays2`, that contains data about flight delays. You would like the new table to contain the same columns and
attributes as the existing table Sasuser.Flightdelays, but you do not want to include any of the existing data. The following PROC SQL step uses a CREATE TABLE statement and a LIKE clause to create Work.Flightdelays2:

```sql
proc sql;
create table work.flightdelays2
    like sasuser.flightdelays;
```

The following DESCRIBE TABLE statement displays the structure of the empty table Work.Flightdelays2:

```sql
proc sql;
describe table work.flightdelays2;
```

Table 5.8  SAS Log

```sql
NOTE: SQL table WORK.FLIGHTDELAYS2 was created like:

create table WORK.FLIGHTDELAYS2( bufsize=8192 )
    (  
        Date num format=DATE9. informat=DATE9.,  
        FlightNumber char(3),  
        Origin char(3),  
        Destination char(3),  
        DelayCategory char(15),  
        DestinationType char(15),  
        DayOfWeek num,  
        Delay num  
    );
```

Work.Flightdelays2 contains eight columns, as listed.

Specifying a Subset of Columns from a Table

If you want to create an empty table that contains only a specified subset of columns from an existing table, use the SAS data set options DROP= or KEEP=.

General form, DROP= and KEEP= data set options:

```
(DROP | KEEP =column-1<...column-n>)
```

where

- `column` specifies the name of a column to be dropped or kept. Multiple column names must be separated by spaces. The entire option must be enclosed in parentheses.

In the CREATE TABLE statement, the DROP= or KEEP= option can be inserted in either of the following locations:

- between the name of the table that is being created and the LIKE clause (as shown in the following example)
- after the name of the source table, at the end of the LIKE clause.
**Example**

The following PROC SQL step creates the new table `Work.Flightdelays3` that contains a subset of columns from the table `Sasuser.Flightdelays`. The DROP= option is used to specify that all columns except `DelayCategory` and `DestinationType` will be included in the new table.

```sql
proc sql;
  create table work.flightdelays3
    (drop=delaycategory destinationtype)
  like sasuser.flightdelays;
```

For comparison, the results of running the DESCRIBE TABLE statement for the original table and the new table are shown below.

**Table 5.9 SAS Log**

```
NOTE: SQL table WORK.FLIGHTDELAYS was created like:
create table SASUSER.FLIGHTDELAYS( bufsize=8192 )
  (  
    Date num format=DATE9. informat=DATE9.,
    FlightNumber char(3),
    Origin char(3),
    Destination char(3),
    DelayCategory char(15),
    DestinationType char(15),
    DayOfWeek num,
    Delay num
  );
```

**Table 5.10 SAS Log**

```
NOTE: SQL table WORK.FLIGHTDELAYS was created like:
create table WORK.FLIGHTDELAYS3( bufsize=4096 )
  (  
    Date num format=DATE9. informat=DATE9.,
    FlightNumber char(3),
    Origin char(3),
    Destination char(3),
    DayOfWeek num,
    Delay num
  );
```

As these messages show, `Sasuser.Flightdelays` contains the columns `DelayCategory` and `DestinationType`, while `Work.Flightdelays3` does not.

**Note:** In PROC SQL, you can apply most of the SAS data set options, such as DROP= and KEEP=, to tables any time that you specify a table. You can use a more limited set of SAS data set options with PROC SQL views. However, because the DROP= and KEEP= options are SAS options and not part of the ANSI standard for SQL, you can use the DROP= and KEEP= options only with the SAS implementation of SQL.
Creating a Table from a Query Result

Sometimes you want to create a new table that contains both columns and rows that are derived from an existing table or set of tables. In this situation, you can submit one PROC SQL step that does both of the following:

- creates a new table
- populates the table with data from the result of a PROC SQL query.

To create a table from a query result, use a CREATE TABLE statement that includes the keyword AS and the clauses that are used in a query: SELECT, FROM, and any optional clauses, such as ORDER BY.

General form, CREATE TABLE statement with query clauses:

```
CREATE TABLE table-name AS
    SELECT column-1<, ... column-n>
    FROM table-1 | view-1<, ... table-n | view-n>
    <optional query clauses>;
```

where

- `table-name` specifies the name of the table to be created.
- `SELECT` specifies the column(s) that will appear in the table.
- `FROM` specifies the table(s) or view(s) to be queried.
- `optional query clauses` are used to refine the query further and include WHERE, GROUP BY, HAVING, and ORDER BY.

You should be familiar with the use of the CREATE TABLE statement to create a table from a query result. Let’s quickly review the basics of this method.

When a table is created from a query result,

- the new table is populated with data that is derived from one or more tables or views that are referenced in the query’s FROM clause
- the new table contains the columns that are specified in the query’s SELECT clause
- the new table’s columns have the same column attributes (type, length, informat, and format) as those of the selected source columns.

Note: When you are creating a table, if you do not specify a column alias for a calculated column, SAS will assign a column name, such as `_TEMA001`.

When query clauses are used within a CREATE TABLE statement, that query’s automatic report generation is shut off. Only the new table is generated as output.

**Example**

Suppose you want to create a new, temporary table that contains data for ticket agents who are employed by an airline. The data that you need is a subset of the data
contained in two existing tables, *Sasuser.Payrollmaster* and *Sasuser.Staffmaster*. The following PROC SQL step creates the new table *Work.Ticketagents* from the result of a query on the two existing tables. The WHERE clause selects the subset of rows for employees whose *JobCode* contains TA.

```sql
proc sql;
create table work.ticketagents as
    select lastname, firstname,
           jobcode, salary
    from sasuser.payrollmaster,
         sasuser.staffmaster
    where payrollmaster.empid
          = staffmaster.empid
          and jobcode contains 'TA';
.
Note: Because this query lists two tables in the FROM clause and subsets rows based on a WHERE clause, the query is actually a PROC SQL inner join.

The new table *Work.Ticketagents* is not empty; it contains rows of data. Therefore, you can submit a SELECT statement to display *Work.Ticketagents* as a report:

```sql
select *
    from work.ticketagents;
```

The first four rows of *Work.Ticketagents* are shown below.

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>GERALD</td>
<td>TA2</td>
<td>$48,126</td>
</tr>
<tr>
<td>AVERY</td>
<td>JERRY</td>
<td>TA3</td>
<td>$54,361</td>
</tr>
<tr>
<td>BLALOCK</td>
<td>RALPH</td>
<td>TA2</td>
<td>$45,661</td>
</tr>
<tr>
<td>BOSTIC</td>
<td>MARIE</td>
<td>TA3</td>
<td>$54,299</td>
</tr>
</tbody>
</table>

The SAS log also displays a message, indicating that the table has been created.

**Table 5.11 SAS Log**

```
NOTE: Table WORK.TICKETAGENTS created, with 41 rows and 4 columns.
```

### Copying a Table

To copy a table quickly, you can use the CREATE TABLE statement with a query that returns an entire table instead of a subset of columns and rows. The CREATE TABLE statement should contain only the following clauses:

- a SELECT clause that specifies that all columns from the source table should be selected
- a FROM clause that specifies the source table.

**Note:** Remember that the order of rows in a PROC SQL query result cannot be guaranteed, unless you use an ORDER BY clause. Therefore, a CREATE TABLE statement without an ORDER BY clause can create a table that contains the same rows as the original table, but the rows might be in a different order.
Example

The following PROC SQL step creates the new table Work.Supervisors2, which is an exact duplicate of the source table Sasuser.Supervisors:

```
proc sql;
    create table work.supervisors2 as
    select *
    from sasuser.supervisors;
```

The first four rows of the two tables are shown below.

**Figure 5.1** Source Table: Sasuser.Supervisors

<table>
<thead>
<tr>
<th>Supervisor Id</th>
<th>State</th>
<th>Job Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1677</td>
<td>CT</td>
<td>BC</td>
</tr>
<tr>
<td>1634</td>
<td>NY</td>
<td>BC</td>
</tr>
<tr>
<td>1431</td>
<td>CT</td>
<td>FA</td>
</tr>
<tr>
<td>1433</td>
<td>NJ</td>
<td>FA</td>
</tr>
</tbody>
</table>

**Figure 5.2** New Table: Work.Supervisors2

<table>
<thead>
<tr>
<th>Supervisor Id</th>
<th>State</th>
<th>Job Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1677</td>
<td>CT</td>
<td>BC</td>
</tr>
<tr>
<td>1634</td>
<td>NY</td>
<td>BC</td>
</tr>
<tr>
<td>1431</td>
<td>CT</td>
<td>FA</td>
</tr>
<tr>
<td>1433</td>
<td>NJ</td>
<td>FA</td>
</tr>
</tbody>
</table>

Inserting Rows of Data into a Table

After you have created an empty table, you’ll want to insert rows of data. You might also want to insert additional rows of data into tables that already contain data. You can use the `INSERT` statement in three different ways to insert rows of data into existing tables, either empty or populated.

**Note:** You can also use the INSERT statement to insert rows of data in a single table that underlies a PROC SQL view. To learn more about PROC SQL views, see Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243.
Method of Inserting Row | Example
--- | ---
insert values by column name by using the SET clause | proc sql;
insert into work.discount
set destination='LHR',
begindate='01MAR2000'd,
enddate='05MAR2000'd,
discount=.33
set destination='CPH',
begindate='03MAR2000'd,
enddate='10MAR2000'd,
discount=.15;

insert lists of values by using the VALUES clause | proc sql;
insert into work.discount (destination, begindate,enddate,discount)
values ('LHR','01MAR2000'd,'05MAR2000'd,.33)
values ('CPH','03MAR2000'd,'10MAR2000'd,.15);

insert rows that are copied from another table by using a query result | proc sql;
insert into payrollchanges2
select empid,salary,dateofhire
from sasuser.payrollmaster
where empid in ('1919','1350','1401');

In each method, the INSERT statement inserts new rows into the table, then adds data to the rows. To indicate that the rows have been inserted, the SAS log displays a message similar to the following.

Table 5.12  SAS Log

**NOTE: 1 row was inserted into WORK.DISCOUNT.**

Let’s see how to use each of these methods to insert rows of data into a table.

**Inserting Rows by Using the SET Clause**

Sometimes you need to add rows of data to a table, but the data is not currently contained in any table. In this situation, you can use either the SET clause or the VALUES clause in the INSERT statement to specify the data to be added.

The **SET clause** in the **INSERT statement** enables you to insert rows and specify new data to be added to a table. The SET clause specifies column names and values in pairs. PROC SQL reads each column name-value pair and assigns the value to the specified column. A separate SET clause is used for each row to be added to the table.

The syntax of the INSERT statement that contains the SET clause is shown below.
General form, INSERT statement containing the SET clause:

```
INSERT INTO table-name <(target-column-1<, ... target-column-n>>
  SET column-1=value-1<, ... column-n=value-n>
  <... SET column-1=value-1<, ... column-n=value-n>>;
```

where

- `table-name` specifies the name of the table to which rows will be inserted.
- `target-column` specifies the name of a column into which data will be inserted.
- Each `SET` clause specifies one or more values to be inserted in one or more specified columns in a row. Multiple `SET` clauses are not separated by commas.
- `column` specifies the name of a column into which data will be inserted.
- `value` specifies a data value to be inserted into the specified column. Character values must be enclosed in quotation marks.
- Multiple `column=value` pairs in a `SET` clause are separated by commas.

Note: It is optional to include a list of target column names after the table name in the INSERT TABLE statement that includes a SET clause. The list can include the names of all or only a subset of columns in the table. If you specify an optional list of target column names, then you can specify values only for columns that are in the list. You can list target columns in any order, regardless of their position in the table. Any columns that are in the table but not listed are given missing values in the inserted rows. △

Note: Although it is recommended that the SET clause list column-value pairs in order (as they appear in the table or the optional column list), it is not required. △

**Example**

Let’s take another look at the table `Work.Discount`, which was presented in the last topic. `Work.Discount` stores airline discounts for certain flight destinations and time periods in March. By submitting a DESCRIBE TABLE statement, you can see this table’s columns and column attributes.
Creating and Managing Tables Using PROC SQL  △  Inserting Rows by Using the VALUES Clause

Table 5.13  SAS Log

<table>
<thead>
<tr>
<th>NOTE: SQL table WORK.DISCOUNT was created like:</th>
</tr>
</thead>
<tbody>
<tr>
<td>create table WORK.DISCOUNT( bufsize=4096 )</td>
</tr>
<tr>
<td>(</td>
</tr>
<tr>
<td>Destination char(3),</td>
</tr>
<tr>
<td>BeginDate num format=DATE9.,</td>
</tr>
<tr>
<td>EndDate num format=DATE9.,</td>
</tr>
<tr>
<td>Discount num</td>
</tr>
<tr>
<td>);</td>
</tr>
</tbody>
</table>

The following PROC SQL step does both of the following:

- adds two rows of new data to Work.Discount by using an INSERT statement that contains two SET clauses, one for each row
- generates a report that displays Work.Discount, with its two new rows, by using a SELECT statement.

In this situation, you don’t need to include an optional list of column names.

```sql
proc sql;
  insert into work.discount
    set destination='LHR',
        begindate='01MAR2000'd,
        enddate='05MAR2000'd,
        discount=.33
  set destination='CPH',
        begindate='03MAR2000'd,
        enddate='10MAR2000'd,
        discount=.15;
  select *
  from discount;
```

### Inserting Rows by Using the VALUES Clause

The INSERT statement uses the VALUES clause to insert a list of values into a table. Unlike the SET clause, the VALUES clause does not specify a column name for each value, so the values must be listed in the correct order. Values must be specified in the order in which the columns appear in the table or, if an optional column list is specified, in the order that the columns appear in that list.
General form, INSERT statement containing the VALUES clause:

```sql
INSERT INTO table-name <(target-column-1<, ... target-column-n)>
    VALUES (value-1<, ... value-n)>
    <... VALUES (value-1<, ... value-n)>;
```

where

- `table-name` specifies the name of the table to which rows will be inserted.
- `target-column` specifies the name of a column into which data will be inserted.
- Each `VALUES` clause lists the values to be inserted in some or all columns in one row. Multiple `VALUES` clauses are not separated by commas.
- `value` specifies a data value to be added. Character values must be enclosed in quotation marks. Multiple values must be separated by commas. Values must be listed in positional order, either as they appear in the table or, if the optional column list is specified, as they appear in the column list.

**Note:** It is optional to include a list of target column names after the table name in the INSERT TABLE statement that includes a VALUES clause. The list can include the names of all or only a subset of columns in the table. If an optional list of target column names is specified, then only those columns are given values by the statement. Target columns can be listed in any order, regardless of their position in the table. Any columns that are in the table but not listed are given missing values in the inserted rows.

You can use the VALUES clause to insert a value for all or only some of the columns in the table.
Creating and Managing Tables Using PROC SQL

If you want to ... Then ... Example

| Insert a value for all columns in the table | You can omit the optional list of column names in the INSERT statement. | insert into work.newtable
values ('WI','FLUTE',6)
values ('ST','VIOLIN',3); |

PROC SQL
- reads values in the order in which they are specified in the VALUES clause
- inserts the values into columns in the order in which the columns appear in the table.

| Insert a value for only some of the columns in the table | You must include a list of column names in the INSERT statement. | insert into work.newtable
(item,qty)
values ('FLUTE',6)
values ('VIOLIN',3); |

PROC SQL
- reads values in the order in which they are specified in the VALUES clause
- inserts the values into columns in the order in which the columns are specified in the column list.

You must list a value for every column into which PROC SQL will insert values (as specified in either the table or the optional list of column names). To specify that a value is missing, use a space enclosed in single quotation marks for character values and a period for numeric values. For example, the following VALUES clause specifies values to be inserted in three columns; the first two values are missing:

values (' ', ., 45)

In this example, the first value specified is a missing value for a character column, and the second value is a missing value for a numeric column.

Example

Suppose you want to insert two more rows into the table Work.Discount, which stores airline discounts for certain flight destinations and time periods in March. In the previous section, you inserted two rows into Work.Discount by using the SET clause, so the table now looks like the following table.

<table>
<thead>
<tr>
<th>Destination</th>
<th>BeginDate</th>
<th>EndDate</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHR</td>
<td>01MAR2000</td>
<td>05MAR2000</td>
<td>0.33</td>
</tr>
<tr>
<td>CPH</td>
<td>03MAR2000</td>
<td>10MAR2000</td>
<td>0.15</td>
</tr>
</tbody>
</table>
Now, let’s add two more rows, this time by using the VALUES clause. The following PROC SQL step adds two rows of new data to Work.Discount and generates a report that displays the updated table:

```sql
proc sql;
insert into work.discount (destination, begindate, enddate, discount)
values ('ORD','05MAR2000'd,'15MAR2000'd,.25)
values ('YYZ','06MAR2000'd,'20MAR2000'd,.10);
select *
from work.discount;
```

<table>
<thead>
<tr>
<th>Destination</th>
<th>BeginDate</th>
<th>EndDate</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHR</td>
<td>01MAR2000</td>
<td>05MAR2000</td>
<td>0.33</td>
</tr>
<tr>
<td>CPH</td>
<td>03MAR2000</td>
<td>10MAR2000</td>
<td>0.15</td>
</tr>
<tr>
<td>ORD</td>
<td>05MAR2000</td>
<td>15MAR2000</td>
<td>0.25</td>
</tr>
<tr>
<td>YYZ</td>
<td>06MAR2000</td>
<td>20MAR2000</td>
<td>0.1</td>
</tr>
</tbody>
</table>

The two rows that were just inserted by using the VALUES clause are the third and fourth rows above.

You may have noticed that the INSERT statement in this example includes an optional list of column names. In this example, data is being inserted into all columns of the table, and the values are listed in the order in which the columns appear in the table, so it is not strictly necessary to use a column list. However, including the list of column names makes it easier to read the code and understand what the code is doing.

**Inserting Rows from a Query Result**

The fastest way to insert rows of data into a table is to use a query to select existing rows from one or more tables (or views) and to insert the rows into another table. You can insert rows from a query result into either an empty table or a table that already contains rows of data. When you add rows of data to a table that already contains rows, the new rows are added at the end of the table.

To insert rows from a query result, use an `INSERT statement` that includes the clauses that are used in a query: SELECT, FROM, and any optional clauses, such as ORDER BY. Values from the query result are inserted into columns in the order in which the columns appear in the table or, if an optional column list is specified, in the order that the columns appear in that list.
General form, INSERT statement containing query clauses:

\[
\text{INSERT INTO } \text{table-name} \begin{cases} \text{(target-column-1<, ... target-column-n>)} \end{cases} \\
\text{SELECT} \begin{cases} \text{column-1<, ... column-n>} \end{cases} \\
\text{FROM} \begin{cases} \text{table-1 | view-1<, ... table-n | view-n>} \end{cases} \\
<\text{optional query clauses}>; \\
\end{cases}
\]

where

- **table-name** specifies the name of the table to which rows will be inserted.
- **target-column** specifies the name of a column into which data will be inserted.
- **SELECT** specifies the column(s) that will be inserted.
- **FROM** specifies the table(s) or view(s) to be queried.
- **optional query clauses** are used to refine the query further. These include the WHERE, GROUP BY, HAVING, and ORDER BY clauses.

**Note:** It is optional to include a list of target column names after the table name in the INSERT TABLE statement that includes query clauses. The list can include the names of all or only a subset of columns in the table. If an optional list of target column names is specified, then only those columns are given values by the statement. Target columns may be listed in any order, regardless of their position in the table. Any columns that are in the table but not listed are given missing values in the inserted rows.

**Example**

A mechanic at a company has been promoted from level 2 to level 3, and you need to add this employee to **Sasuser.Mechanicslevel3**, a table that lists all level-3 mechanics. To start, let’s create a temporary copy of **Sasuser.Mechanicslevel3** called **Work.Mechanicslevel3_New**, and display the new table in a report:

```sql
proc sql; 
create table work.mechanicslevel3_new as 
select * 
from sasuser.mechanicslevel3;
```

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1499</td>
<td>ME3</td>
<td>$60,225</td>
</tr>
<tr>
<td>1409</td>
<td>ME3</td>
<td>$58,171</td>
</tr>
<tr>
<td>1379</td>
<td>ME3</td>
<td>$59,170</td>
</tr>
<tr>
<td>1521</td>
<td>ME3</td>
<td>$59,136</td>
</tr>
<tr>
<td>1365</td>
<td>ME3</td>
<td>$61,460</td>
</tr>
<tr>
<td>1420</td>
<td>ME3</td>
<td>$60,286</td>
</tr>
<tr>
<td>1682</td>
<td>ME3</td>
<td>$58,153</td>
</tr>
</tbody>
</table>

Next, you insert a row into **Work.Mechanicslevel3_New** for the new level-3 employee, whose **EmpID** is 1653. This employee is currently listed in **Sasuser.Mechanicslevel2**, so
your INSERT statement queries the table Sasuser.Mechanicslevel2. Your PROC SQL step ends with a SELECT statement that outputs the revised table Work.Mechanicslevel3_New to a report.

```sql
proc sql;
  insert into work.mechanicslevel3_new
  select empid, jobcode, salary
  from sasuser.mechanicslevel2
  where empid='1653';
  select *
  from work.mechanicslevel3_new;
```

The row that you have inserted into Work.Mechanicslevel3_New is row 8 above. As you can see, the values for JobCode and Salary for the new employee will have to be changed. Updating existing values in a table is covered later in this chapter.

Note: Although the new row is shown here at the bottom of the table, the order of rows in a PROC SQL query cannot be guaranteed if an ORDER BY clause is not used.

---

### Creating a Table That Has Integrity Constraints

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a column in a table. SAS enforces integrity constraints when values associated with a column are added, updated, or deleted. Integrity constraints help you preserve the validity and consistency of your data.

You can create integrity constraints by using either PROC SQL or PROC DATASETS. PROC DATASETS can assign constraints only to an existing table. PROC SQL can assign constraints either as it creates a new table or as it modifies an existing table. This chapter discusses the use of PROC SQL to create integrity constraints while creating a table.

Note: To learn more about the use of PROC DATASETS to create integrity constraints, see Chapter 18, “Modifying SAS Data Sets and Tracking Changes,” on page 633. For additional information about integrity constraints, see the SAS documentation.

When you place integrity constraints on a table, you specify the type of constraint that you want to create. Each constraint has a different action.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CHECK</strong></td>
<td>Ensures that a specific set or range of values are the only values in a column. It can also check the validity of a value in one column based on a value in another column within the same row.</td>
</tr>
<tr>
<td><strong>NOT NULL</strong></td>
<td>Guarantees that a column has non-missing values in each row.</td>
</tr>
<tr>
<td><strong>UNIQUE</strong></td>
<td>Enforces uniqueness for the values of a column.</td>
</tr>
</tbody>
</table>
Creating and Managing Tables Using PROC SQL

Referential Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMARY KEY</td>
<td>Uniquely defines a row within a table, which can be a single column or a set of columns. A table can have only one PRIMARY KEY. The PRIMARY KEY includes the attributes of the constraints NOT NULL and UNIQUE.</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>Links one or more rows in a table to a specific row in another table by matching a column or set of columns (a FOREIGN KEY) in one table with the PRIMARY KEY in another table. This parent/child relationship limits modifications made to both PRIMARY KEY and FOREIGN KEY constraints. The only acceptable values for a FOREIGN KEY are values of the PRIMARY KEY or missing values.</td>
</tr>
</tbody>
</table>

Note: When you add an integrity constraint to a table that contains data, SAS checks all data values to determine whether they satisfy the constraint before the constraint is added.

You can use integrity constraints in two ways, general and referential. General constraints enable you to restrict the data values accepted for a column in a single table. Referential constraints enable you to link the data values of a column in one table to the data values of columns in another table.

### General Integrity Constraints

General integrity constraints enable you to restrict the values of columns within a single table. The following four integrity constraints can be used as general integrity constraints:

- CHECK
- NOT NULL
- UNIQUE
- PRIMARY KEY.

Note: A PRIMARY KEY constraint is a general integrity constraint if it does not have any FOREIGN KEY constraints referencing it. A PRIMARY KEY used as a general constraint is a shortcut for assigning the constraints NOT NULL and UNIQUE.

### Referential Integrity Constraints

A referential integrity constraint is created when a PRIMARY KEY integrity constraint in one table is referenced by a FOREIGN KEY integrity constraint in another table. There are two steps that must be followed to create a referential integrity constraint:

1. Define a PRIMARY KEY constraint on the first table.
2. Define a FOREIGN KEY constraint on other tables.

Note: Integrity constraints

- follow ANSI standards
- cannot be defined for views
- cannot be defined for historical versions of generation data sets.
To create a table that has integrity constraints, use a CREATE TABLE statement that specifies both columns and constraints. There are two ways to specify integrity constraints in the CREATE TABLE statement:

- in a column specification
- as a separate constraint specification.

You can use either or both of these methods in the same CREATE TABLE statement.

Creating a Constraint in a Column Specification

Earlier in this chapter, you learned how to create a table by using a CREATE TABLE statement that contains column specifications:

```
CREATE TABLE table-name
  (column-specification-1,<, ...column-specification-n>);
```

You also learned that a column specification consists of these elements:

- `column-definition <column-constraint-1<, ... column-constraint-n>>`
- `<MESSAGE='message-string' <MSGTYPE=message-type>>`

The column specifications used in earlier examples contained only the column definition. Now let’s see how to create an integrity constraint with a column, by specifying the optional `column constraint` in the column specification:
General form, column-constraint in a column-specification:

```
column-definition <<column-constraint-1>, ..., column-constraint-n>>
<<MESSAGE='message-string' <MSGTYPE=message-type>>
```

where

column-constraint

is one of the following:

- **CHECK (expression)**
  
  specifies that all rows in the table (which is specified in the CREATE TABLE statement) satisfy the expression, which can be any valid where-expression.

- **DISTINCT**
  
  specifies that the values of the column must be unique within the table. 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.

- **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 column 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. The referential-action specifies the type of action to be performed on all matching FOREIGN KEY values and is one of the following:

  - **CASCADE**
    
    allows PRIMARY KEY data values to be updated, and updates matching values in the FOREIGN KEY to the same values.

    Note: 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.

- **UNIQUE**
  
  specifies that the values of the column must be unique within the table. This constraint is identical to DISTINCT.

Note: The optional MSGTYPE= and MESSAGE= elements will be discussed later in this chapter. △

Just like a column, an integrity constraint must have a unique name within the table. When you create an integrity constraint by specifying a column constraint in a
column specification, then SAS automatically assigns a name to the constraint. The form of the constraint name depends on the type of constraint, as shown below:

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Default Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECK</td>
<td><em>CKxxxx</em></td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td><em>FKxxxx</em></td>
</tr>
<tr>
<td>NOT NULL</td>
<td><em>NMxxxx</em></td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td><em>PKxxxx</em></td>
</tr>
<tr>
<td>UNIQUE</td>
<td><em>UNxxxx</em></td>
</tr>
</tbody>
</table>

Note: xxxx is a counter that begins at 0001.

Now let’s look at an example of a PROC SQL step that creates integrity constraints by specifying one or more column constraints in a column specification.

Example

Suppose you need to create the table Work.Employees to store the identification number, name, gender, and hire date for all employees. In addition, you want to ensure that

- the ID column contains only values that are nonmissing and unique
- the Gender column contains only the values M and F.

The following PROC SQL step creates the table Work.Employees, which contains four columns and integrity constraints for two of those columns:

```sql
proc sql;
create table work.employees
(ID char (5) primary key,
 Name char(10),
 Gender char(1) not null check(gender in ('M','F')),
 HDate date label='Hire Date');
```

In the column specification for ID, the PRIMARY KEY column constraint ensures that the ID column will contain only values that are nonmissing and unique.

The column specification for Gender defines two integrity constraints:

- The NOT NULL column constraint ensures that the values of Gender will be nonmissing values.
- The CHECK column constraint ensures that the values of Gender will satisfy the expression gender in ('M','F').

Next, let’s look at the other method of creating integrity constraints: specifying a constraint specification in the CREATE TABLE statement.

Creating a Constraint by Using a Constraint Specification

Sometimes you might prefer to create integrity constraints outside of column specifications, by specifying individual constraint specifications in the CREATE TABLE statement:

```
CREATE TABLE table-name
```
(column-specification-1<, 
...column-specification-n><, 
constraint-specification-1><, 
...constraint-specification-n>); 

The first specification in the CREATE TABLE statement must be a column specification. However, following the initial column specification in the statement, you can include multiple additional column specifications, constraint specifications, or both. All specifications after the first specification can be listed in any order. The entire list of column specifications and constraint specifications follows the same guidelines that were presented earlier for column specifications:

- The entire set of column specifications and constraint specifications must be enclosed in parentheses.
- Multiple column specifications and constraint specifications must be separated by commas.

There are several important differences between specifying an integrity constraint within a column specification and specifying an integrity constraint by using a separate constraint specification. Using a constraint specification offers the following advantages:

- You can specify a name for the constraint. In fact, you must specify a name, because SAS does not automatically assign one.
- For certain constraint types, you can define a constraint for multiple columns in a single specification.

The syntax of a constraint specification is shown below.
General form, constraint specification:

```
CONSTRAINT constraint-name constraint <MESSAGE='message-string' 
<MSGTYPE=message-type>>
```

where

- **constraint-name** specifies a name for the constraint that is being specified. The name must be a valid SAS name.

**CAUTION:**

The names PRIMARY, FOREIGN, MESSAGE, UNIQUE, DISTINCT, CHECK, and NOT cannot be used as values for constraint-name.

**constraint** is one of the following:

- **CHECK (expression)** specifies that all rows in table-name (which is specified in the CREATE TABLE statement) satisfy the expression, which can be any valid where-expression.

- **DISTINCT (column-1, ... column-n)** specifies that the values of each column must be unique within the table. This constraint is identical to UNIQUE.

- **FOREIGN KEY (column-1, ... column-n)**

  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 column 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. The referential-action specifies the type of action to be performed on all matching FOREIGN KEY values, and is one of the following:

  - **CASCADE** allows PRIMARY KEY data values to be updated, and updates matching values in the FOREIGN KEY to the same values.

    *Note:* 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 the column does not contain a null or missing value, including special missing values.

- **PRIMARY KEY (column-1, ... column-n)** specifies one or more columns as PRIMARY KEY columns, that is, columns that do not contain missing values and whose values are unique.

- **UNIQUE (column-1, ... column-n)** specifies that the values of each column must be unique within the table. This constraint is identical to DISTINCT.
Creating and Managing Tables Using PROC SQL

MESSAGE=
specifies a message-string that specifies the text of an error message that is written to the SAS log when the integrity constraint is not met. The maximum length of message-string is 250 characters.

MSGTYPE=
specifies the message-type, which specifies how the error message is displayed in the SAS log when an integrity constraint is not met. The message-type is one of the following:

- NEWLINE
  the text that is specified for MESSAGE= is displayed in addition to the default error message for that integrity constraint.

- USER
  only the text that is specified for MESSAGE= is displayed.

Elements within a constraint-specification must be separated by spaces.

You might have noticed another difference between the two methods of creating an integrity constraint. When you use a column specification to create a FOREIGN KEY integrity constraint, you use the keyword FOREIGN KEY in addition to the keyword REFERENCES.

Now let’s look at an example of a PROC SQL step that uses column specifications to create integrity constraints on a column.

Example

In an example earlier in this chapter, the table Work.Discount was created to hold data about discounts that are offered by an airline. Suppose you now want to ensure that the table

- holds only discounts that are less than or equal to .5
- does not allow missing values for Destination.

Let’s create a new version of the table Work.Discount, called Work.Discount3, that includes two integrity constraints. One integrity constraint limits the values that can be entered in the Discount column and the other prevents missing values from being entered in the Destination column. The following PROC SQL step creates Work.Discount3 by specifying four columns and two integrity constraints:

```sql
proc sql;
  create table work.discount3
    (Destination char(3),
     BeginDate num Format=date9.,
     EndDate num format=date9.,
     Discount num,
     constraint ok_discount check (discount le .5),
     constraint notnull_dest not null(destination));
```

The CHECK integrity constraint named OK_Discount uses the WHERE expression `discount le .5` to limit the values that can be added to the Discount column.

The NOT NULL integrity constraint named NotNull_Dest prevents missing values from being entered in Destination.
Handling Errors in Row Insertions

When you add rows to a table that has integrity constraints, PROC SQL evaluates the new data to ensure that it meets the conditions that are determined by the integrity constraints. If the new (or modified) data complies with the integrity constraints, the rows are added. However, if you add data that does not comply with the integrity constraints, the rows are not added. To find out whether rows of data have been successfully added, you need to check the SAS log.

Note: PROC SQL also evaluates changes that are made to existing data by using the UPDATE and DELETE statements. These statements are discussed later in this chapter.

Example

In a previous section of this chapter, the following PROC SQL step was used to create the table Work.Discount3 with two integrity constraints, one on the column Discount and the other on the column Destination:

```sql
proc sql;
create table work.discount3
(Destination char(3),
BeginDate num Format=date9.,
EndDate num format=date9.,
Discount num,
constraint ok_discount check (discount le .5),
constraint notnull_dest not null(destination));
```

This table does not yet contain any rows, so let’s add some data. The following INSERT statement uses the VALUES clause to add two rows of data to the table:

```sql
proc sql;
insert into work.discount3
values('CDG','03MAR2000’d,’10MAR2000’d,.15)
values('LHR','10MAR2000’d,’12MAR2000’d,.55);
```

When this PROC SQL step is submitted, the following messages are displayed in the SAS log.

Table 5.14 SAS Log

```
ERROR: Add/Update failed for data set WORK.DISCOUNT3
because data value(s) do not comply with integrity constraint
ok_discount.
NOTE: This insert failed while attempting to add data from
VALUES clause 2 to the data set.
NOTE: Deleting the successful inserts before error noted above
to restore table to a consistent state.
```

The three parts of this message explain what the problem is:

- The error message indicates that this attempt to add rows failed. One or more of the data values for Discount does not comply with the integrity constraint OK_Discount, which specifies that values in the column Discount must be less than or equal to .5.

- The first note indicates that there is a problem with the second VALUES clause. This clause specifies a value of .55 for the column Discount, which does not comply.
CAUTION:
Even if multiple VALUES clauses specify non-compliant data values, the SAS log lists only the first VALUES clause that violates the constraint.

The second note indicates that SAS is “deleting the successful inserts” before the error. Even though all the other specified data is valid, none of the data has been added to the table.

Let’s find out why SAS prevented any of the data from being added to the table.

Using the UNDO_POLICY= Option to Control UNDO Processing

When you use the INSERT or UPDATE statement to add or modify data in a table, you can control how PROC SQL handles updated data if any errors occur during the insertion or update process. You can use the UNDO_POLICY= option in the PROC SQL statement to specify whether PROC SQL will make or undo the changes you submitted up to the point of the error.

You can specify one of the following values for the UNDO_POLICY= option.

<table>
<thead>
<tr>
<th>UNDO_POLICY=Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIRED</td>
<td>PROC SQL performs UNDO processing for INSERT and UPDATE statements. If the UNDO operation cannot be done reliably, PROC SQL does not execute the statement and issues an ERROR message. This is the PROC SQL default.</td>
</tr>
<tr>
<td>NONE</td>
<td>PROC SQL skips records that cannot be inserted or updated, and writes a warning message to the SAS log similar to that written by PROC APPEND. Any data that meets the integrity constraints is added or updated.</td>
</tr>
<tr>
<td>OPTIONAL</td>
<td>PROC SQL performs UNDO processing if it can be done reliably. If the UNDO cannot be done reliably, then no UNDO processing is attempted. This action is a combination of REQUIRED and NONE. If UNDO can be done reliably, then it is done, and PROC SQL proceeds as if UNDO_POLICY=REQUIRED is in effect. Otherwise, it proceeds as if UNDO_POLICY=NONE was specified.</td>
</tr>
</tbody>
</table>

CAUTION:
In the following situations, you cannot reliably attempt the UNDO operation:

- A SAS data set that is accessed through a SAS/SHARE server and opened with CNTLLEV=RECORD can allow other users to update newly inserted records. An error during the insert deletes the record that the other user updated.
- Changes made through a SAS/ACCESS view might not be able to reverse changes made by the INSERT or UPDATE statement without reversing other changes at the same time.

Note: The ANSI standard for SQL includes a ROLLBACK statement that is used for UNDO processing. The ROLLBACK statement is not currently supported in PROC SQL.
Note: When you use the UNDO_POLICY= option, the value that you set remains in effect for the entire PROC SQL statement or until a RESET statement is used to change the option. To learn more about the RESET statement, see Chapter 8, “Managing Processing Using PROC SQL,” on page 261.

Example

In the last example, the INSERT step was used to insert two rows of data into the table Work.Discount3, which has two integrity constraints. Because the UNDO_POLICY= option was not specified in the code, PROC SQL used the default policy, which is UNDO_POLICY=REQUIRED. When PROC SQL encountered a value in the INSERT statement that violated an integrity constraint, none of the new values specified in the INSERT statement were added to the table.

Now let’s see what happens when we submit the same INSERT statement and specify the option UNDO_POLICY=NONE.

The following PROC SQL step creates the table Work.Discount4, which has four columns and two integrity constraints, and inserts the same two rows of data that were inserted in the earlier example. In this case, however, the option UNDO_POLICY=NONE is specified.

```
proc sql undo_policy=none;
  create table work.discount4
    (Destination char(3),
     BeginDate num format=date9.,
     EndDate num format=date9.,
     Discount num,
     constraint ok_discount check (discount le .5),
     constraint notnull_dest not null(destination));
insert into work.discount4
  values('CDG','03MAR2000'd,'10MAR2000'd,.15)
  values('LHR','10MAR2000'd,'12MAR2000'd,.55);
```

As you know, one of the data values for the column Discount violates the specified constraint. When this step is submitted, the SAS log displays the following messages.

```Table 5.15 SAS Log
WARNING: The SQL option UNDO_POLICY=REQUIRED is not in effect.
If an error is detected when processing this INSERT statement,
that error will not cause the entire statement to fail.
ERROR: Add/Update failed for data set WORK.DISCOUNT4 because
data value(s) do not comply with integrity constraint ok_discount.
NOTE: This insert failed while attempting to add data from VALUES
clause 2 to the data set.
NOTE: 2 rows were inserted into WORK.DISCOUNT4 -- of these 1 row
was rejected as an ERROR, leaving 1 row that was inserted
successfully.
```

The four parts of this message explain what the problem is and how PROC SQL will handle UNDO processing:

- The warning tells you that you have specified a setting for the UNDO_POLICY= option that is different from the default (REQUIRED). The warning also explains that, as a result, if an error is detected, the error will not cause the entire INSERT statement to fail.
The error message was also displayed in the earlier example, when the default setting of UNDO_POLICY= was in effect. This message states that the INSERT statement failed and explains why.

The first note was also displayed in the earlier example, when the default setting of UNDO_POLICY= was in effect. This note identifies the first VALUES clause that contains non-compliant data.

The second note tells you that one row (the first row of the two rows that you specified) was inserted successfully into the table.

### Displaying Integrity Constraints for a Table

Sometimes you want to add data to a table but you are not sure what integrity constraints, if any, the table has. To display only the integrity constraints for a specified table, use a **DESCRIBE TABLE CONSTRAINTS statement**. (The DESCRIBE TABLE statement, which is discussed earlier in this chapter, lists both a CREATE TABLE statement and the table’s integrity constraints in the SAS log.)

**Note:** Some versions of SAS display information about integrity constraints in output as well as in the SAS log. △

Sometimes you want to add data to a table but you are not sure what integrity constraints, if any, the table has. To display only the integrity constraints for a specified table, use a **DESCRIBE TABLE CONSTRAINTS statement**. (The DESCRIBE TABLE statement, which is discussed earlier in this chapter, lists both a CREATE TABLE statement in the SAS log and a listing of the table’s integrity constraints in output.)

<table>
<thead>
<tr>
<th>General form, DESCRIBE TABLE CONSTRAINTS statement:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DESCRIBE TABLE CONSTRAINTS</strong> table-name-1[, ... table-name-n];</td>
</tr>
<tr>
<td>where</td>
</tr>
<tr>
<td><strong>table-name</strong></td>
</tr>
<tr>
<td>specifies the table to be described as one of the following:</td>
</tr>
<tr>
<td>□ a one-level name</td>
</tr>
<tr>
<td>□ a two-level libref.table name</td>
</tr>
<tr>
<td>□ a physical pathname that is enclosed in single quotation marks.</td>
</tr>
</tbody>
</table>

### Example

To display only the table constraints for the table Work.Discount4 that was created earlier, you submit the following PROC SQL step:

```sql
proc sql;
    describe table constraints work.discount4;
```
Table 5.16  SAS Log

**NOTE: SQL table WORK.DISCOUNT4 ( bufsize=4096 ) has the following integrity constraint(s):**

<table>
<thead>
<tr>
<th>Integrity Constraint</th>
<th>Type</th>
<th>Variables</th>
<th>Where Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>notnull_dest</td>
<td>Not Null</td>
<td>Destination</td>
<td>Destination</td>
</tr>
<tr>
<td>ok_discount</td>
<td>Check</td>
<td>Discount</td>
<td>Discount &lt;= 0.5</td>
</tr>
</tbody>
</table>

As shown, Work.Discount4 has two integrity constraints: **NotNull_Dest** and **OK_Discount**.

### Updating Values in Existing Table Rows

To modify data values in some or all of the existing rows in a table, you use the **UPDATE statement** in PROC SQL. In the UPDATE statement, for each column whose rows you want to modify, you specify an **expression** that indicates how the values should be modified. For example, the following expression indicates that the values for the column **Units** should be multiplied by 4:

```
units=units*4
```

You can use the UPDATE statement in two main ways.

<table>
<thead>
<tr>
<th>Method of Updating Table</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>update all (or a subset of) rows in a column with the same expression</td>
<td>proc sql; update work.payrollmaster_new set salary=salary*1.05 where jobcode like '__1';</td>
</tr>
<tr>
<td>update different rows in a column with different expressions</td>
<td>proc sql; update work.payrollmaster_new set salary=salary*1.05 case when substr(jobcode,3,1)='1' then 1.05 when substr(jobcode,3,1)='2' then 1.10 when substr(jobcode,3,1)='3' then 1.15 else 1.08 end;</td>
</tr>
</tbody>
</table>
Note: The UPDATE statement does not insert new rows into the table. To insert rows, you must use the INSERT statement.

Note: You can also use the UPDATE statement to update existing values in a table that underlies a PROC SQL view. For details, see Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243.

Let’s look at each of these methods of updating existing rows in a table.

### Updating Rows by Using the Same Expression

To update all (or a subset of) rows in a column with the same expression, use an *UPDATE statement* that contains a *SET clause* and (optionally) a *WHERE clause*.

**General form, basic UPDATE statement for updating table rows:**

```
UPDATE table-name
SET column-1=expression,<,..., column-n=expression>
WHERE expression;
```

*table-name* specifies the name of the table in which values will be updated. 

*SET* specifies one or more pairs of *column* names to be updated, and *expressions* that indicate how each column is to be updated.

*WHERE* is optionally used to specify an *expression* that subsets the rows to be updated.

**CAUTION:**

If you want to update only a subset of rows in the table, you must specify a WHERE clause or all rows of the table will be updated.

### Example

Suppose a company is considering giving all level-1 employees a 5% raise. Employee salaries are stored in the table *Sasuser.Payrollmaster*. You don’t want to update the original table, so you create a temporary copy of *Sasuser.Payrollmaster*, called *Work.Payrollmaster_New*. The following PROC SQL step creates *Work.Payrollmaster_New* based on a query result and generates an output report of the new table:

```
proc sql;
  create table work.payrollmaster_new as
    select * from sasuser.payrollmaster;
    select * from work.payrollmaster_new;
```

The first 10 rows of *Work.Payrollmaster_New*, the table in which you will update salaries, are shown below.
Next, you write a PROC SQL step that updates the specified rows. The UPDATE statement contains both of the following:

- a SET clause that specifies the expression to be used in updating `salary`.
- a WHERE clause that specifies a subset of rows (level-1 employees) to be updated.

```sql
proc sql;
update work.payrollmaster_new
    set salary=salary*1.05
    where jobcode like '__1';
```

Finally, you can use a SELECT statement to display the updated table as a report. The first 10 rows of `Work.Payrollmaster_New`, with updates, are shown below.

```plaintext
<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>16SEP1958</td>
<td>07JUN1986</td>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>$48,126</td>
</tr>
<tr>
<td>19OCT1962</td>
<td>12AUG1988</td>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>$49,151</td>
</tr>
<tr>
<td>08NOV1965</td>
<td>19OCT1988</td>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>$41,677</td>
</tr>
<tr>
<td>04SEP1963</td>
<td>01AUG1988</td>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>$46,040</td>
</tr>
<tr>
<td>16DEC1948</td>
<td>21NOV1983</td>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>$54,351</td>
</tr>
<tr>
<td>29APR1952</td>
<td>11JUN1978</td>
<td>1499</td>
<td>M</td>
<td>ME3</td>
<td>$60,235</td>
</tr>
<tr>
<td>03JUN1960</td>
<td>04OCT1988</td>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>$26,212</td>
</tr>
<tr>
<td>03APR1969</td>
<td>14FEB1979</td>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>$124,048</td>
</tr>
<tr>
<td>20JAN1961</td>
<td>05DEC1988</td>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>$45,661</td>
</tr>
<tr>
<td>26DEC1966</td>
<td>09OCT1987</td>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>$54,299</td>
</tr>
</tbody>
</table>
```

The third row lists data for a level-1 employee, and that person’s salary has been updated.

If you wanted to increase all of the salaries, you would simply remove the WHERE clause from the UPDATE statement:

```sql
proc sql;
update work.payrollmaster_new
    set salary=salary*1.05;
```
Updating Rows by Using Different Expressions

Sometimes you want to use different expressions to modify values for different subsets of rows within a column.

For example, instead of only raising the salary of level-1 employees by 5%, you might also want to raise the salaries of level-2 employees by 10%, and so on, using a different percentage increase for each group of employees.

There are two possible ways to use different expressions to update different subsets of rows.

<table>
<thead>
<tr>
<th>Method of Updating Table</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>use multiple UPDATE statements, one for each subset of rows</td>
<td>proc sql;</td>
</tr>
<tr>
<td></td>
<td>update work.payrollmaster_new</td>
</tr>
<tr>
<td></td>
<td>set salary=salary*1.05</td>
</tr>
<tr>
<td></td>
<td>where substr(jobcode,3,1)='1';</td>
</tr>
<tr>
<td></td>
<td>update work.payrollmaster_new</td>
</tr>
<tr>
<td></td>
<td>set salary=salary*1.10</td>
</tr>
<tr>
<td></td>
<td>where substr(jobcode,3,1)='2';</td>
</tr>
<tr>
<td></td>
<td>update work.payrollmaster_new</td>
</tr>
<tr>
<td></td>
<td>set salary=salary*1.15</td>
</tr>
<tr>
<td></td>
<td>where substr(jobcode,3,1)='3';</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>use a single UPDATE statement that contains a CASE expression</td>
<td>proc sql;</td>
</tr>
<tr>
<td></td>
<td>update work.payrollmaster_new</td>
</tr>
<tr>
<td></td>
<td>set salary=salary*case</td>
</tr>
<tr>
<td></td>
<td>when substr(jobcode,3,1)='1' then 1.05</td>
</tr>
<tr>
<td></td>
<td>when substr(jobcode,3,1)='2' then 1.10</td>
</tr>
<tr>
<td></td>
<td>when substr(jobcode,3,1)='3' then 1.15</td>
</tr>
<tr>
<td></td>
<td>else 1.08</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
</tbody>
</table>

The first method, which requires the use of multiple UPDATE statements, is cumbersome because the SET statement and expression must be repeated in each UPDATE statement. In this example, the first method is inefficient because the table Work.Payrollmaster_New must be read three times.

The second method, which uses conditional processing (the CASE expression), is recommended. Let’s take a look at the second method now.

To update different subsets of rows in a table in different ways, you can incorporate conditional processing by using the CASE expression in the SET clause of an UPDATE statement. The CASE expression selects result values that satisfy specified conditions.
General form, CASE expression:

```sql
CASE <case-operand>
    WHEN when-condition THEN result-expression
    <...WHEN when-condition THEN result-expression>
    <ELSE result-expression>
END;
```

where

- **CASE**
  - performs conditional processing.
- **case-operand**
  - is an optional expression that resolves to a table column whose values are compared to all the **when-conditions**.
- **WHEN**
  - specifies a **when-condition**, a shortened expression that assumes **case-operand** as one of its operands, and that resolves to true or false.
- **THEN**
  - specifies a **result-expression**, an expression that resolves to a value.
- **ELSE**
  - specifies a **result-expression**, which provides an alternate action if none of the **when-conditions** is executed.
- **END**
  - indicates the end of the CASE expression.

**CAUTION:**

Although the ELSE clause is optional, its use is strongly recommended. If you omit the ELSE clause, each row that is not described in one of the WHEN clauses receives a missing value for the column that you are updating.

**Note:** You can also use the CASE expression in the INSERT and SELECT statements.

**Example**

In the following UPDATE statement, the CASE expression contains three WHEN-THEN clauses that specify three different subsets of rows in the table `Work.Insure_New`:

- homeowners that are insured by Acme
- homeowners that are insured by Reliable
- homeowners that are insured by Homelife.

```sql
update work.insure_new
set pctinsured=pctinsured*
    case
        when company='ACME'
        then 1.10
```
PROC SQL updates each specified subset of rows differently, according to the corresponding WHEN-THEN (or ELSE) clause.

---

**How PROC SQL Updates Rows Based on a CASE Expression**

When you specify a CASE expression, PROC SQL updates each row as follows:

1. In the CASE expression, PROC SQL finds the WHEN-THEN clause that contains a condition that the row matches.
2. The CASE expression then returns the result from the matching WHEN-THEN clause to the SET clause. The returned value completes the expression in the SET clause.
3. The SET clause uses the completed expression to update the value of the specified column in the current row.

The use of the CASE expression is efficient because of the way PROC SQL processes the WHEN-THEN clauses. The WHEN-THEN clauses in the CASE expression are evaluated sequentially. When a matching case is found, the THEN expression is evaluated and set, and the remaining WHEN cases are not considered.

---

**How the Case Operand Works**

In the next few sections, you will learn about the use of the CASE expression in the UPDATE statement, without and with the optional case operand:

```
CASE <case-operand>
```

---

**Updating Rows by Using the CASE Expression without a Case Operand**

Let’s look at an example of an UPDATE statement that uses the CASE expression for conditional processing. This example shows the form of the CASE expression that does not include the optional case operand.

---

**Example**

Suppose a company is considering giving raises to all of its employees, with a different percentage for each employee level:

- level-1 employees get a 5% raise
- level-2 employees get a 10% raise
- level-3 employees get a 15% raise.

First, you create the temporary table `Work.Payrollmaster3`, which is a copy of `Sasuser.Payrollmaster`, the table containing the employee salary data. The first 10 rows of `Work.Payrollmaster3` are shown below.
Next, you create a PROC SQL step that updates rows by using an UPDATE statement that contains a SET clause and a CASE expression:

```sql
proc sql;
update work.payrollmaster3
set salary=salary*
    case
        when substr(jobcode,3,1)='1'
            then 1.05
        when substr(jobcode,3,1)='2'
            then 1.10
        when substr(jobcode,3,1)='3'
            then 1.15
        else 1.08
    end;
end;
```

In this example, the CASE expression contains three WHEN clauses, one for each subset of rows (level-1, level-2, and level-3 employees), followed by an ELSE clause to handle any rows that do not meet the expected conditions.

The first 10 rows of `Work.Payrollmaster3`, after the rows have been updated, are shown below.
By comparing the values of **Salary** in the original and updated versions of *Work.Payrollmaster3* (as shown above), you can see how the values have changed according to the job level indicated in the **JobCode**.

---

**Updating Rows by Using the CASE Expression with a Case Operand**

If the expression in the SET clause uses an equals (=) comparison operator, you may use the optional case operand in the CASE expression. Let's take another look at the PROC SQL step that was shown in the preceding example, and see how the CASE expression in the UPDATE statement can be rewritten by using the alternate syntax.

**Example**

In the following PROC SQL step, which was shown earlier, the CASE expression contains three WHEN-THEN clauses. These clauses contain similar expressions, each of which specifies the same SUBSTR function:

```sql
proc sql;
update work.payrollmaster_new2
set salary=salary*
    case
        when substr(jobcode,3,1)='1'
            then 1.05
        when substr(jobcode,3,1)='2'
            then 1.10
        when substr(jobcode,3,1)='3'
            then 1.15
        else 1.08
    end;
end;
```

Because the expression in this SET clause uses an equals (=) operator, you can restructure the CASE expression for more efficient processing. In the alternate syntax, the repeated SUBSTR function is removed from each WHEN-THEN clause and is placed after the keyword CASE, as an operand:

```sql
proc sql;
update work.payrollmaster_new2
set salary=salary*
    case substr(jobcode,3,1)
        when '1'
            then 1.05
        when '2'
            then 1.10
        when '3'
            then 1.15
        else 1.08
    end;
end;
```

Using the alternate syntax, the SUBSTR function is evaluated only once, so this PROC SQL step is more efficient than the original version.

*Note:* You may use the case operand syntax only if the SET clause expression uses the equals (=) comparison operator.

---
Using the CASE Expression in the SELECT Statement

You can use the CASE expression in three different PROC SQL statements: UPDATE, INSERT, and SELECT. In the SELECT statement, you can use the CASE expression within a new column definition to specify different values for different subsets of rows.

Example

Suppose you want to generate an output report that displays employee names, job codes, and job levels. Your PROC SQL query selects LastName and FirstName from Sasuser.Staffmaster, and JobCode from Sasuser.Payrollmaster. The SELECT statement must define JobLevel as a new column, because it does not exist as a separate column in either table.

You want to assign the values of JobLevel, based on the number at the end of each jobcode. (The number at the end of each JobCode value is expected to be 1, 2, or 3.) To create JobLevel, you can use the case operand form of the CASE expression to specify the three possible conditions (plus an ELSE condition, just in case).

The PROC SQL query is shown below:

```sql
proc sql outobs=10;
select lastname, firstname, jobcode,
    case substr(jobcode,3,1)
        when '1'
            then 'junior'
        when '2'
            then 'intermediate'
        when '3'
            then 'senior'
        else 'none'
    end as JobLevel
from sasuser.payrollmaster,
    sasuser.staffmaster
where staffmaster.empid=
    payrollmaster.empid;
```

The SELECT clause uses the CASE expression to assign a value of junior, intermediate, senior, or none to each row in the new JobLevel column.

Deleting Rows in a Table

To delete some or all of the rows in a table, use the DELETE statement. When the statement is successfully executed, the SAS log shows a message that indicates the number of rows that have been deleted.
General form, DELETE statement for deleting rows in a table:

```
DELETE FROM table-name
  <WHERE expression>;
```

where

- `table-name` specifies the name of the table in which rows will be deleted.
- `WHERE` is optionally used to specify an `expression` that subsets the rows to be deleted.

**CAUTION:**

If you want to delete only a subset of rows in the table, you must specify a WHERE clause or all rows in the table will be deleted. △

**Note:** You can also use the DELETE statement to delete rows in a table that underlies a PROC SQL view. For more information about referencing a PROC SQL view in a DELETE statement, see Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243. △

---

**Example**

Suppose you want to delete the records for all frequent-flyer program members who have used up all of their frequent flyer miles or have spent more miles than they had in their accounts.

First, you create the temporary table `Work.Frequentflyers2` by copying a subset of columns from the existing table `Sasuser.Frequentflyers`:

```
proc sql;
  create table work.frequentflyers2 as
    select ffid, milestraveled, pointsearned, pointsused
    from sasuser.frequentflyers;
```

The first 10 rows of `Work.Frequentflyers2` are shown below.

<table>
<thead>
<tr>
<th>FFID</th>
<th>MilesTraveled</th>
<th>PointsEarned</th>
<th>PointsUsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD7152</td>
<td>30833</td>
<td>31333</td>
<td>0</td>
</tr>
<tr>
<td>WD8472</td>
<td>25570</td>
<td>26070</td>
<td>0</td>
</tr>
<tr>
<td>WD1576</td>
<td>56144</td>
<td>58644</td>
<td>27000</td>
</tr>
<tr>
<td>WD3947</td>
<td>40922</td>
<td>45922</td>
<td>23000</td>
</tr>
<tr>
<td>WD9347</td>
<td>4839</td>
<td>9839</td>
<td>0</td>
</tr>
<tr>
<td>WD8375</td>
<td>30007</td>
<td>30507</td>
<td>25000</td>
</tr>
<tr>
<td>WD7208</td>
<td>48943</td>
<td>49443</td>
<td>30000</td>
</tr>
<tr>
<td>WD6061</td>
<td>60142</td>
<td>60642</td>
<td>40000</td>
</tr>
<tr>
<td>WD0646</td>
<td>87044</td>
<td>89544</td>
<td>25000</td>
</tr>
<tr>
<td>WD9829</td>
<td>1901</td>
<td>4401</td>
<td>0</td>
</tr>
</tbody>
</table>
Next, you write a PROC SQL step that deletes the specified rows:

```sql
proc sql;
   delete from work.frequentflyers2
       where pointsearned-pointsused <= 0;
```

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

### Table 5.17 SAS Log

| NOTE: 13 rows were deleted from WORK.FREQUENTFLYERS2 |

**Tip:** To delete all of the rows in the table, remove the WHERE clause from the DELETE statement.

---

## Altering Columns in a Table

You’ve seen how to delete rows in a table using the DELETE statement. To add, drop (delete), or modify columns in a table, use the **ALTER TABLE statement**.

**General form, ALTER TABLE statement:**

```sql
ALTER TABLE table-name
   <ADD column-definition-1, ..., column-definition-n>>
   <DROP column-name-1, ..., column-name-n>>
   <MODIFY column-definition-1, ..., column-definition-n>>;
```

where

- **table-name** specifies the name of the table in which columns will be added, dropped, or modified.
- **<ADD, DROP, MODIFY>** at least one of the following clauses must be specified:
  - ADD specifies one or more **column-definitions** for columns to be added.
  - DROP specifies one or more **column-names** for columns to be dropped (deleted).
  - MODIFY specifies one or more **column-definitions** for columns to be modified, where
    - **column-definition** specifies a column to be added or modified, and is formatted as follows:
      ```sql
      column-name data-type <(column-width)> <column-modifier-1
      <...column-modifier-n>>
      ```

In all three clauses, multiple **column-definitions** or **column-names** must be separated by commas.

**Note:** You **cannot** use the ALTER TABLE statement with views.

**Note:** The ALTER TABLE statement also supports similar clauses that add, drop, and modify integrity constraints in an existing table. These clauses are not discussed in
this chapter. To find out more about adding, dropping, and modifying integrity constraints, see the SAS documentation for the SQL procedure.

Let's take a closer look at each type of modification that can be made to a column by using the ALTER TABLE statement.

### Adding Columns to a Table

To add columns to a table, use the **ADD clause** in the **ALTER TABLE statement**. The ADD clause specifies one or more **column definitions**. The syntax of a column definition is the same as in the CREATE TABLE statement:

```
column-name data-type <(column-width)> <column-modifier-1>
< ...column-modifier-n>>
```

However, in the ALTER statement, the entire group of column definitions is *not* enclosed in parentheses.

### Example

Suppose you are working with the temporary table `Work.Payrollmaster4`, which is an exact copy of the existing table `Sasuser.Payrollmaster`. The first 10 rows of `Work.Payrollmaster4` are shown below.

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>16SEP1968</td>
<td>07JUN1995</td>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>$48,126</td>
</tr>
<tr>
<td>19OCT1962</td>
<td>12AUG1998</td>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>$49,151</td>
</tr>
<tr>
<td>08NOV1965</td>
<td>19OCT1998</td>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>$41,677</td>
</tr>
<tr>
<td>04SEP1963</td>
<td>01AUG1998</td>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>$46,040</td>
</tr>
<tr>
<td>16DEC1946</td>
<td>21NOV1983</td>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>$54,351</td>
</tr>
<tr>
<td>29APR1952</td>
<td>11JUN1978</td>
<td>1499</td>
<td>M</td>
<td>ME3</td>
<td>$60,235</td>
</tr>
<tr>
<td>09JUN1960</td>
<td>04OCT1988</td>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>$26,212</td>
</tr>
<tr>
<td>03APR1959</td>
<td>14FEB1979</td>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>$124,048</td>
</tr>
<tr>
<td>20JAN1961</td>
<td>05DEC1998</td>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>$45,661</td>
</tr>
<tr>
<td>26DEC1966</td>
<td>09OCT1987</td>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>$54,298</td>
</tr>
</tbody>
</table>

The following PROC SQL step uses the ADD clause in the ALTER TABLE statement to add the columns **Bonus** and **Level** to `Work.Payrollmaster4`:

```
proc sql;
  alter table work.payrollmaster4
    add Bonus num format=comma10.2,
       Level char(3);

The first 10 rows of `Work.Payrollmaster4`, with the two new columns added, are shown below.
Dropping Columns from a Table

To drop (delete) existing columns from a table, use the DROP clause in the ALTER TABLE statement. The DROP clause specifies one or more column names, and multiple column names are separated by commas.

Example

Suppose you want to drop the existing columns Bonus and Level from the temporary table Work.Payrollmaster4. (These two columns were added to the table in the example in the previous section.) The first 10 rows of Work.Payrollmaster4 are shown below.

The following PROC SQL step uses the DROP clause in the ALTER TABLE statement to drop the columns Bonus and Level from Work.Payrollmaster4:

```
proc sql;
  alter table work.payrollmaster4
    drop bonus, level;
```
Modifying Columns in a Table

To modify the attributes of one or more existing columns in a table, use the MODIFY clause in the ALTER TABLE statement. You can use the MODIFY clause to change a column's

- length (column width) — for a character column only
- informat
- format
- label.

Note: You cannot use the MODIFY clause to

- change a character column to numeric or 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.
- change a column's name. You cannot change this attribute by using the ALTER TABLE statement; instead, you can use the SAS data set option RENAME= or the DATASETS procedure with the RENAME statement. You can find out more about the DATASETS procedure with the RENAME statement in Chapter 13, “Creating Samples and Indexes,” on page 451.

Like the ADD clause, the MODIFY clause specifies one or more column definitions, each of which consists of

\[
\text{column-name <data-type (column-width)> <column-modifier-1} \\
\text{<...column-modifier-n>}
\]

In each column definition, you specify the required element (the column name), followed by any of the optional attributes that you want to modify.

Note: When you use a column definition to add a new column by using the ADD clause in the ALTER TABLE statement, or to specify a new column in the CREATE TABLE statement, data-type is a required element. However, when you are using a column definition in the MODIFY clause in the ALTER TABLE statement, as shown in the following example, data-type is never required for numeric columns and is optional.
for character columns. You must specify `data-type (column-width)` only if you want to modify the column width of a character column.

---

**Example**

Suppose you want to modify the attributes of the existing column **Salary** in the temporary table `Work.Payrollmaster4`. The first 10 rows of `Work.Payrollmaster4` (as it existed at the end of the previous example) are shown below.

Example

Suppose you want to modify the attributes of the existing column **Salary** in the temporary table `Work.Payrollmaster4`. The first 10 rows of `Work.Payrollmaster4` (as it existed at the end of the previous example) are shown below.

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>16SEP1956</td>
<td>07JUN1985</td>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>$48,126</td>
</tr>
<tr>
<td>19OCT1962</td>
<td>12AUG1988</td>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>$49,151</td>
</tr>
<tr>
<td>08NOV1965</td>
<td>19OCT1988</td>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>$41,677</td>
</tr>
<tr>
<td>04SEP1963</td>
<td>01AUG1988</td>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>$46,040</td>
</tr>
<tr>
<td>16DEC1948</td>
<td>21NOV1983</td>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>$54,351</td>
</tr>
<tr>
<td>29APR1952</td>
<td>11JUN1976</td>
<td>1499</td>
<td>M</td>
<td>ME3</td>
<td>$60,235</td>
</tr>
<tr>
<td>03JUN1960</td>
<td>04OCT1988</td>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>$26,212</td>
</tr>
<tr>
<td>03APR1959</td>
<td>11FEB1979</td>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>$124,046</td>
</tr>
<tr>
<td>20JAN1961</td>
<td>05DEC1988</td>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>$45,661</td>
</tr>
<tr>
<td>26DEC1966</td>
<td>09OCT1987</td>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>$54,299</td>
</tr>
</tbody>
</table>

The column **Salary** is a numeric field that currently has the format DOLLAR9. The following PROC SQL step modifies the format and adds a label for **Salary**:

```sql
proc sql;
  alter table work.payrollmaster4
    modify salary format=dollar11.2 label="Salary Amt";
```

The first 10 rows of `Work.Payrollmaster4`, with the new column attributes for **Salary**, are shown below.

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>16SEP1956</td>
<td>07JUN1985</td>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>$48,126.00</td>
</tr>
<tr>
<td>19OCT1962</td>
<td>12AUG1988</td>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>$49,151.00</td>
</tr>
<tr>
<td>08NOV1965</td>
<td>19OCT1988</td>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>$41,677.00</td>
</tr>
<tr>
<td>04SEP1963</td>
<td>01AUG1988</td>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>$46,040.00</td>
</tr>
<tr>
<td>16DEC1948</td>
<td>21NOV1983</td>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>$54,351.00</td>
</tr>
<tr>
<td>29APR1952</td>
<td>11JUN1978</td>
<td>1499</td>
<td>M</td>
<td>ME3</td>
<td>$60,235.00</td>
</tr>
<tr>
<td>03JUN1960</td>
<td>04OCT1988</td>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>$26,212.00</td>
</tr>
<tr>
<td>03APR1959</td>
<td>11FEB1979</td>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>$124,046.00</td>
</tr>
<tr>
<td>20JAN1961</td>
<td>05DEC1988</td>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>$45,661.00</td>
</tr>
<tr>
<td>26DEC1966</td>
<td>09OCT1987</td>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>$54,299.00</td>
</tr>
</tbody>
</table>
Adding, Dropping, and Modifying Columns in a Single Statement

In the last few examples, the ALTER TABLE statement has made only one alteration to columns in a table, by using just one clause. However, you can include multiple clauses in a single ALTER TABLE statement to add, drop, and modify columns all at once.

Example

Suppose you want to use a single ALTER TABLE statement to make all of the following alterations to the table Work.Payrollmaster4:

- add the new column **Age**, by using the ADD clause
- change the format of the **DateOfHire** column (which is currently DATE9.) to MMDDYY10., by using the MODIFY clause
- drop the **DateOfBirth** and **Gender** columns, by using the DROP clause.

The first 10 rows of *Work.Payrollmaster4*, as it was at the end of the last example, are shown below.

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>16SEP1958</td>
<td>07JUN1985</td>
<td>1319</td>
<td>M</td>
<td>TA2</td>
<td>$48,126.00</td>
</tr>
<tr>
<td>19OCT1962</td>
<td>12AUG1968</td>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>$49,151.00</td>
</tr>
<tr>
<td>08NOV1965</td>
<td>19OCT1968</td>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>$41,677.00</td>
</tr>
<tr>
<td>04SEP1963</td>
<td>01AUG1968</td>
<td>1350</td>
<td>F</td>
<td>FA3</td>
<td>$46,040.00</td>
</tr>
<tr>
<td>16DEC1948</td>
<td>21NOV1963</td>
<td>1401</td>
<td>M</td>
<td>TA3</td>
<td>$54,351.00</td>
</tr>
<tr>
<td>29APR1952</td>
<td>11JUN1970</td>
<td>1439</td>
<td>M</td>
<td>ME3</td>
<td>$50,235.00</td>
</tr>
<tr>
<td>05JUN1960</td>
<td>04OCT1968</td>
<td>1101</td>
<td>M</td>
<td>SCP</td>
<td>$26,212.00</td>
</tr>
<tr>
<td>03APR1959</td>
<td>14FEB1979</td>
<td>1333</td>
<td>M</td>
<td>PT2</td>
<td>$124,048.00</td>
</tr>
<tr>
<td>20JAN1961</td>
<td>05DEC1968</td>
<td>1402</td>
<td>M</td>
<td>TA2</td>
<td>$45,661.00</td>
</tr>
<tr>
<td>26DEC1966</td>
<td>09OCT1967</td>
<td>1479</td>
<td>F</td>
<td>TA3</td>
<td>$54,299.00</td>
</tr>
</tbody>
</table>

The following PROC SQL step uses multiple clauses in the ALTER TABLE statement to make all three of the alterations listed above:

```sql
proc sql;
  alter table work.payrollmaster4
    add Age num
    modify DateOfHire date format=mmddyy10.
    drop DateOfBirth, Gender;
```

The first 10 rows of *Work.Payrollmaster4*, with the three alterations, are shown below.
To drop (delete) one or more entire tables, use the *DROP TABLE* statement.

**General form, DROP TABLE statement:**

```
DROP TABLE table-name-1 <, ... table-name-n>;
```

where

- `table-name` specifies the name of a table to be dropped, and can be one of the following:
  - a one-level name
  - a two-level libref.table name
  - a physical pathname that is enclosed in single quotation marks.

---

**Example**

In the last few examples, you made several alterations to the temporary table `Work.Payrollmaster4`. Now you decide that you do not need this table anymore. The following PROC SQL step uses the DROP TABLE statement to drop `Work.Payrollmaster4`:

```
proc sql;
  drop table work.payrollmaster4;
```

The SAS log displays a message indicating that the table has been dropped:

**Table 5.18  SAS Log**

<table>
<thead>
<tr>
<th>DateOfHire</th>
<th>EmplID</th>
<th>JobCode</th>
<th>Salary</th>
<th>Amt</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/07/1985</td>
<td>1919</td>
<td>TA2</td>
<td>$48,126.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>08/12/1988</td>
<td>1653</td>
<td>ME2</td>
<td>$49,151.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>10/19/1988</td>
<td>1400</td>
<td>ME1</td>
<td>$41,677.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>08/01/1988</td>
<td>1350</td>
<td>FA3</td>
<td>$46,040.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>11/21/1983</td>
<td>1401</td>
<td>TA3</td>
<td>$54,351.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>06/11/1978</td>
<td>1499</td>
<td>ME3</td>
<td>$60,235.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>10/04/1988</td>
<td>1101</td>
<td>SCP</td>
<td>$26,212.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>02/14/1979</td>
<td>1333</td>
<td>PT2</td>
<td>$124,048.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>12/05/1988</td>
<td>1402</td>
<td>TA2</td>
<td>$45,661.00</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>10/09/1987</td>
<td>1479</td>
<td>TA3</td>
<td>$54,299.00</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** Table WORK.PAYROLLMASTER4 has been dropped.
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Understanding Methods of Creating Tables

You can use the CREATE TABLE statement to create a table in three different ways:
- create a table with no rows (an empty table) by defining columns
- create an empty table that is like another table
- create a table that contains rows, based on a query result.

Creating an Empty Table by Defining Columns

You can create a table with no rows by using a CREATE TABLE statement that contains column specifications. A column specification includes the following elements: column name (required), data type (required), column width (optional), one or more column modifiers (optional), and a column constraint (optional).

Displaying the Structure of a Table

To display, in the SAS log, a list of a table's columns and their attributes and other information about a table, use the DESCRIBE TABLE statement.

Creating an Empty Table That Is Like Another Table

To create a table with no rows that has the same structure as an existing table, use a CREATE TABLE statement that contains the keyword LIKE. To specify a subset of columns to be copied from the existing table, use the SAS data set options DROP= or KEEP= in your CREATE TABLE statement.

Creating a Table from a Query Result

To create a new table that contains both columns and rows that are derived from an existing table or set of tables, use a CREATE TABLE statement that includes the keyword AS and the clauses that are used in a query. This method enables you to copy an existing table quickly.

Inserting Rows of Data into a Table

The INSERT statement can be used in three ways to insert rows of data in existing tables, either empty or populated. You can insert rows by using
- the SET clause to specify column names and values in pairs
- the VALUES clause to specify a list of values
- the clauses that are used in a query to return rows from an existing table.
Creating a Table That Has Integrity Constraints

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a column in a table. To create a table that has integrity constraints, use a CREATE TABLE statement. Integrity constraints can be defined in two different ways in the CREATE TABLE statement:
- by specifying a column constraint in a column specification
- by using a constraint specification.

Handling Errors in Row Insertions

When you add rows to a table that has integrity constraints, PROC SQL evaluates the new data to ensure that it meets the conditions that are determined by the integrity constraints. When you use the INSERT or UPDATE statement to add or modify data in a table, you can use the UNDO_POLICY= option in the PROC SQL statement to specify whether PROC SQL will make or undo the changes you submitted up to the point of the error.

Displaying Integrity Constraints for a Table

To display the integrity constraints for a specified table in the SAS log, use the DESCRIBE TABLE CONSTRAINTS statement.

Updating Values in Existing Table Rows

To modify data values in some or all of the existing rows in a table, use the UPDATE statement with
- a SET clause and (optionally) a WHERE clause that specifies a single expression to update rows. To update rows with multiple expressions, use multiple UPDATE statements.
- a CASE expression that specifies multiple expressions to update rows. The CASE expression can be specified without an optional case operand or, if the expression in the SET clause uses an equals (=) comparison operator, with a case operand.

The CASE expression can also be used in the SELECT statement in a new column definition to specify different values for different subsets of rows.

Deleting Rows in a Table

To delete some or all of the rows in a table, use the DELETE statement.

Altering Columns in a Table

To alter columns in a table, use the ALTER TABLE statement that contains one or more of the following clauses:
- the ADD clause, to add one or more columns to a table
- the DROP clause, to drop (delete) one or more columns in a table
- the MODIFY clause, to modify the attributes of columns in a table.

Dropping Tables

To drop (delete) one or more entire tables, use the DROP TABLE statement.
Syntax

PROC SQL <UNDO_POLICY=REQUIRED | NONE | OPTIONAL>;

CREATE TABLE table-name
  (column-specification-1,<,
   ...column-specification-n,<,
   constraint-specification-1,<,
   ...constraint-specification-n>);

CREATE TABLE table-name
  (column-definition <column-constraint-1<, ... column-constraint-n>>,
   column-definition <column-constraint-1<, ... column-constraint-n>);)

CREATE TABLE table-name
  (DROP | KEEP =column-1<...column-n>)
  LIKE table-1;

CREATE TABLE table-name AS
  SELECT column-1<, ... column-n
  FROM table-1 | view-1<, ... table-n | view-n
  <optional query clauses>;

DESCRIBE TABLE table-name-1<, ... table-name-n>;

DESCRIBE TABLE CONSTRAINTS table-name-1<, ... table-name-n>;

INSERT INTO table-name <(target-column-1<, ... target-column-n)> 
  SET column-1=value-1<, ... column-n=value-n>
  <... SET column-1=value-1<, ... column-n=value-n>>;

INSERT INTO table-name <(target-column-1<, ... target-column-n)> 
  VALUES (value-1<, ... value-n>)
  <... VALUES (value-1<, ... value-n)>;

INSERT INTO table-name <(target-column-1<, ... target-column-n)> 
  SELECT column-1<, ... column-n
  FROM table-1 | view-1<, ... table-n | view-n
  <optional query clauses>;

UPDATE table-name
  SET column-1=expression<, ... column-n=expression>
  <WHERE expression>;
UPDATE table-name
   SET column-1 expression, ... column-n expression>>
   CASE <case-operand>
      WHEN when-condition THEN result-expression
      ...WHEN when-condition THEN result-expression>
      <ELSE result-expression>
   END;

DELETE FROM table-name
   <WHERE expression>;

ALTER TABLE table-name
   <ADD column-definition-1<, ... column-definition-n>>
   <DROP column-name-1<, ... column-name-n>>
   <MODIFY column-definition-1<, ... column-definition-n>>;

DROP TABLE table-name-1 <, ... table-name-n>;
QUIT;

Sample Programs

Creating an Empty Table by Defining Columns
proc sql;
   create table work.discount
      (Destination char(3),
       BeginDate num Format=date9.,
       EndDate num format=date9.,
       Discount num);
quit;

Creating an Empty Table That Is Like Another Table
proc sql;
   create table work.flightdelays2
      (drop=delaycategory destinationtype)
      like sasuser.flightdelays;
quit;

Creating a Table from a Query Result
proc sql;
   create table work.ticketagents as
      select lastname, firstname,
           jobcode, salary
      from sasuser.payrollmaster,
           sasuser.staffmaster
      where payrollmaster.empid = staffmaster.empid
   quit;
and jobcode contains 'TA';
quit;

Displaying the Structure of a Table

proc sql;
   describe table work.discount;
quit;

Inserting Rows into a Table by Specifying Column Names and Values

proc sql;
   insert into work.discount
    set destination='LHR',
       begindate='01MAR2000'd,
       enddate='05MAR2000'd,
       discount=.33
    set destination='CPH',
       begindate='03MAR2000'd,
       enddate='10MAR2000'd,
       discount=.15;
quit;

Inserting Rows into a Table by Specifying Lists of Values

proc sql;
   insert into work.discount (destination, 
    begindate,enddate, discount) 
   values ('LHR','01MAR2000'd, 
    '05MAR2000'd,.33) 
   values ('CPH','03MAR2000'd, 
    '10MAR2000'd,.15);
quit;

Inserting Rows into a Table from a Query Result

proc sql;
   insert into work.payrollchanges2 
    select empid, salary, dateofhire 
    from sasuser.payrollmaster 
    where empid in ('1919','1350','1401');
quit;

Creating a Table That Has Integrity Constraints

proc sql;
   create table work.employees 
    (Name char(10),
     Gender char(1),
     HDate date label='Hire Date' not null,
     constraint prim_key primary key(name),
     constraint gender check(gender in ('M' 'F')));
quit;
Displaying Integrity Constraints for a Table

```
proc sql;
    describe table constraints work.discount4;
quit;
```

Updating Rows in a Table Based on an Expression

```
proc sql;
    update work.payrollmaster_new
        set salary=salary*1.05
        where jobcode like '__1';
quit;
```

Updating Rows in a Table by Using a CASE Expression

```
proc sql;
    update work.payrollmaster_new
        set salary=salary*
            case
                when substr(jobcode,3,1)='1'
                then 1.05
                when substr(jobcode,3,1)='2'
                then 1.10
                when substr(jobcode,3,1)='3'
                then 1.15
                else 1.08
            end;
quit;
```

Updating Rows in a Table by Using a CASE Expression (Alternate Syntax)

```
proc sql outobs=10;
    select lastname, firstname, jobcode,
        case substr(jobcode,3,1)
            when '1'
            then 'junior'
            when '2'
            then 'intermediate'
            when '3'
            then 'senior'
            else 'none'
        end as JobLevel
    from sasuser.payrollmaster,
         sasuser.staffmaster
    where staffmaster.empid=
         payrollmaster.empid;
quit;
```

Deleting Rows in a Table

```
proc sql;
    delete from work.frequentflyers2
        where pointsearned-pointsused<=0;
quit;
```
Adding, Modifying, and Dropping Columns in a Table

```sql
proc sql;
   alter table work.payrollmaster4
       add Age num
       modify dateofhire date format=mmddyy10.
       drop dateofbirth, gender;
quit;
```

Dropping a Table

```sql
proc sql;
   drop table work.payrollmaster4;
quit;
```

Points to Remember

- The CREATE TABLE statement generates only a table as output, not a report.
- The UPDATE statement does not insert new rows into a table. To insert rows, you must use the INSERT statement.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Which of the following PROC SQL steps creates a new table by copying only the column structure (but not the rows) of an existing table?
   a proc sql;
      create table work.newpayroll as
      select *
      from sasuser.payrollmaster;
   b proc sql;
      create table work.newpayroll
      like sasuser.payrollmaster;
   c proc sql;
      create table work.newpayroll
      copy sasuser.payrollmaster;
   d proc sql;
      create table work.newpayroll
      describe sasuser.payrollmaster;

2 Which of the following PROC SQL steps creates a table that contains rows for the level-1 flight attendants only?
   a proc sql;
      create table work.newpayroll as
      select *
      from sasuser.payrollmaster
      where jobcode='FA1';
b proc sql;
    create work.newpayroll as
    select *
        from sasuser.payrollmaster
        where jobcode='FA1';

c proc sql;
    create table work.newpayroll
        copy sasuser.payrollmaster
        where jobcode='FA1';

d proc sql;
    create table work.newpayroll as
        sasuser.payrollmaster
        where jobcode='FA1';

3 Which of the following statements is true regarding the UNDO_POLICY=REQUIRED option?
   a It must be used with the REQUIRED integrity constraint.
   b It ignores the specified integrity constraints if any of the rows that you want to insert or update do not meet the constraint criteria.
   c It restores your table to its original state if any of the rows that you try to insert or update do not meet the specified integrity constraint criteria.
   d It allows rows that meet the specified integrity constraint criteria to be inserted or updated, but rejects rows that do not meet the integrity constraint criteria.

4 Which of the following is not a type of integrity constraint?
   a CHECK
   b NULL
   c UNIQUE
   d PRIMARY KEY

5 Which of the following PROC SQL steps deletes rows for all frequent-flyer program members who traveled less than 10,000 miles?
   a proc sql;
       delete rows
           from work.frequentflyers
           where milestraveled < 10000;

   b proc sql;
       drop rows
           from work.frequentflyers
           where milestraveled < 10000;

   c proc sql;
       drop table
           from work.frequentflyers
           where milestraveled < 10000;

   d proc sql;
       delete
           from work.frequentflyers
           where milestraveled < 10000;
Which of the following PROC SQL steps gives bonuses (in points) to frequent-flyer program members as follows:

- a 50% bonus for members who traveled less than 10,000 miles
- a 100% bonus for members who traveled 10,000 miles or more?

a proc sql;
update work.frequentflyers
set pointsearned=pointsearned*
  case if milestraveled < 10000
    then 1.5
    if milestraveled >= 10000
    then 2
    else 1
  end;
b proc sql;
update work.frequentflyers
set pointsearned=pointsearned*
  case when milestraveled < 10000
    then 1.5
    when milestraveled >= 10000
    then 2
    else 1
  end;
c proc sql;
update work.frequentflyers
set pointsearned=pointsearned*
  case if milestraveled < 10000
    then pointsearned*1.5
    if milestraveled >= 10000
    then pointsearned*2
    else 1
  end;
d proc sql;
update work.frequentflyers
set pointsearned=pointsearned*
  case if milestraveled < 10000
    then pointsearned*1.5
    if milestraveled >= 10000
    then pointsearned*2
    else pointsearned*1
  end;

Which of the following statements is used to add new rows to a table?

a INSERT
b LOAD
c VALUES
d CREATE TABLE

Which of the following statements regarding the ALTER TABLE statement is false?

a It allows you to update column attributes.
b It allows you to add new columns in your table.
c It allows you to drop columns in your table.
d It allows you to change a character column to a numeric column.
9 Which of the following displays the structure of a table in the SAS log?

a proc sql;
   describe as
   select *
   from sasuser.payrollmaster;

b proc sql;
   describe contents sasuser.payrollmaster;

c proc sql;
   describe table sasuser.payrollmaster;

d proc sql;
   describe * from sasuser.payrollmaster;

10 Which of the following creates an empty table that contains the 2 columns **FullName** and **Age**?

a proc sql;
   create table work.names
      (FullName char(25), Age num);

b proc sql;
   create table work.names as
      (FullName char(25), Age num);

c proc sql;
   create work.names
      (FullName char(25), Age num);

d proc sql;
   create table work.names
      set (FullName char(25), Age num);
Creating and Managing Indexes
Using PROC SQL

Overview 222
Introduction 222
Objectives 223
Prerequisites 223
Understanding Indexes 223
Accessing Rows in a Table 223
Simple and Composite Indexes 224
Unique Indexes 224
Example 224
Deciding Whether to Create an Index 225
PROC SQL Queries That Can Be Optimized by an Index 225
Benefits of Using an Index 226
Example: Using an Index to Access a Small Subset of Data 226
Understanding the Costs of Using an Index 227
Guidelines for Creating Indexes 227

Creating an Index 228
Creating Multiple Indexes 228
Example: Creating a Simple Index 229
Example: Creating a Composite, Unique Index 229

Displaying Index Specifications 229
Alternatives to the DESCRIBE TABLE Statement 230
Example 230

Managing Index Usage 231
Understanding How SAS Decides Whether to Use an Index 231
Determining Whether SAS Is Using an Index 231
Example: Query That Uses an Index 232
Example: Query That Does Not Use an Index 232
Controlling Index Usage 233
Using IDXWHERE= to Direct SAS to Use or Not to Use an Index 233
Example 233
Using IDXNAME= to Direct SAS to Use a Specified Index 234
Example 234

Dropping Indexes 235
Example: Dropping a Composite Index 236

Summary 237
Text Summary 237
Understanding Indexes 237
Deciding Whether to Create an Index 237
Creating an Index 237
Displaying Index Specifications 237
Managing Index Usage 237
Overview

Introduction

When processing a query that contains a subsetting WHERE clause or that joins multiple tables, PROC SQL must locate specific rows in the referenced table(s). Creating an index for a table enables PROC SQL, in certain circumstances, to locate specific rows more quickly and efficiently. An index is an auxiliary file that stores the physical location of values for one or more specified columns (key columns) in a table. In an index, each unique value of the key column(s) is paired with a location identifier for the row that contains that value. In the same way that you use a book's subject index to find a page that discusses a particular subject, PROC SQL uses the system of directions in an index to access specific rows in the table directly, by index value. You can create more than one index for a single table. All indexes for a SAS table are stored in one index file.

Note: You cannot create an index on a view.

The following PROC SQL step uses the CREATE INDEX statement to create an index for a table, and uses the DESCRIBE TABLE statement to display information about the index, along with other information about the table, in the SAS log:

```
proc sql;
    create unique index empid
    on work.payrollmaster(empid);
    describe table work.payrollmaster;
```

Table 6.1 SAS Log

```
create table WORK.PAYROLLMASTER( bufsize=4096 )
(
    DateOfBirth num format=DATE9. informat=DATE9.,
    DateOfHire num format=DATE9. informat=DATE9.,
    EmpID char(4),
    Gender char(1),
    JobCode char(3),
    Salary num format=DOLLAR9.
); create unique index EmpID on WORK.PAYROLLMASTER(EmpID);
```

In this chapter, you will learn to use PROC SQL to create and manage various types of indexes.
Objectives

In this chapter, you learn to

- determine when it is appropriate to create an index
- create simple and composite indexes on a table
- create an index that ensures that values of the key column(s) are unique
- control whether PROC SQL uses an index or which index it uses
- display information about the structure of an index in the SAS log
- drop (delete) an index from a table.

Prerequisites

Before reading this chapter, you should complete the following chapters:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
- Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159.

Understanding Indexes

Accessing Rows in a Table

When you submit a query on a table that does not have an index, PROC SQL accesses rows sequentially, in the order in which they are stored in the table. For example, suppose you are working with a table that contains information about employees. You have written a PROC SQL query to select the rows in which the value of Name (the first column) is Smith. To access the rows that you want, PROC SQL begins with the first row and reads through all rows in the table, selecting the rows that satisfy the condition that is expressed in the WHERE clause.

<table>
<thead>
<tr>
<th>Name</th>
<th>HireDate</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>09JAN2000</td>
<td>34</td>
</tr>
<tr>
<td>Baker</td>
<td>14OCT2001</td>
<td>54</td>
</tr>
<tr>
<td>Davis</td>
<td>30MAR2000</td>
<td>49</td>
</tr>
<tr>
<td>Edwards</td>
<td>28JUN2002</td>
<td>52</td>
</tr>
<tr>
<td>Smith</td>
<td>15JAN2000</td>
<td>62</td>
</tr>
<tr>
<td>Yates</td>
<td>04AUG2002</td>
<td>59</td>
</tr>
</tbody>
</table>

When you execute a program that retrieves a small subset of rows from a large table, it can be time-consuming for PROC SQL to read the rows sequentially. In some situations, using an index on a table allows PROC SQL to access a subset of rows more efficiently.

An index stores unique values for a specified column or columns in ascending value order, and includes information about the location of those values in the table. That is, an index is composed of value/identifier pairs that enable you to access a row directly, by value. For example, suppose you have created an index on your table that is based
on the column Name. Using the index, PROC SQL will access the row(s) that you want directly, without having to read all the other rows.

<table>
<thead>
<tr>
<th>Name</th>
<th>DateOfBirth</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>15JAN2000</td>
<td>Y</td>
<td>ME1</td>
<td>$41,677</td>
</tr>
<tr>
<td>Yates</td>
<td>04AUG2002</td>
<td>X</td>
<td>ME2</td>
<td>$49,151</td>
</tr>
</tbody>
</table>

**Simple and Composite Indexes**

You can create two types of indexes:

- simple
- composite.

A **simple index** is based on one column that you specify. The indexed column can be either character or numeric. When you create a simple index by using PROC SQL, you must specify the name of the indexed column as the name of the index.

A **composite index** is based on two or more columns that you specify. The indexed columns can be character, numeric, or a combination of both. In the index, the values of the key columns are concatenated to form a single value.

For example, if you build a composite index on the key columns **LastName** and **FirstName**, a value for the index is composed of the value for **LastName** followed by the value for **FirstName**. Often, a WHERE clause might use only the first column (the primary key) of a composite index, which means that the program will read only the first part of each concatenated value.

When you create a composite index, you must specify a unique name for the index that is not the name of any existing column or index in the table. In the example described above, the composite index cannot be named **Lastname** or **Firstname**.

**Unique Indexes**

If you want to require that values for the key column(s) are unique for each row, you can create either a simple or a composite index as a **unique index**. Once a unique index is defined on one or more columns in a table, SAS will reject any change to the table that would cause more than one row to have the same value(s) for the specified column or composite group of columns.

**Example**

Suppose you are working with the table **Sasuser.Payrollmaster**. The first eight rows of this table are shown below.

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>16SEP1958</td>
<td>07JUN1985</td>
<td>1919</td>
<td>M</td>
<td>TA2</td>
<td>$48,126</td>
</tr>
<tr>
<td>19OCT1962</td>
<td>12AUG1988</td>
<td>1653</td>
<td>F</td>
<td>ME2</td>
<td>$49,151</td>
</tr>
<tr>
<td>08NOV1965</td>
<td>19OCT1988</td>
<td>1400</td>
<td>M</td>
<td>ME1</td>
<td>$41,677</td>
</tr>
</tbody>
</table>
If you know that the column `JobCode` is often specified in a `WHERE` clause expression, you might want to create a *simple index* on the column `JobCode`. You must specify the name of the key column, `JobCode`, as the index name.

Now suppose you are planning to write many queries that specify both `EmpID` and `DateOfHire` in a `WHERE` clause expression. In this case, you might want to create a *composite index* on these two columns. Because employee identification numbers should be unique, it is appropriate to create this index as a unique index. Therefore, you should specify a name for your index that is not the same as the name of any existing column or index in the table. For example, you could name this index `Whenhired`.

### Deciding Whether to Create an Index

An index can reduce the time required to locate a set of rows, especially for a large data file. However, there are costs associated with creating, storing, and maintaining the index. When deciding whether to create an index, you must weigh any benefits in performance improvement against the costs of increased resource usage.

*Note:* This chapter discusses the benefits and costs that are associated with using indexes specifically with PROC SQL. To learn about the costs and benefits of using indexes with other SAS procedures, see the SAS documentation.

### PROC SQL Queries That Can Be Optimized by an Index

To use indexes effectively with PROC SQL, it is important to know the classes of queries that can be processed more efficiently by using an index. The classes of queries that can be optimized are specified below.
Query performance is optimized when the key column occurs in...

<table>
<thead>
<tr>
<th>a WHERE clause expression that contains</th>
<th>Example</th>
</tr>
</thead>
</table>
| a comparison operator                 | `proc sql;
|                                       | select empid, jobcode, salary |
|                                       | from sasuser.payrollmaster |
|                                       | where jobcode='FA3'
|                                       | order by empid; |
| the TRIM or SUBSTR function           |         |
| the CONTAINS operator                 |         |
| the LIKE operator.                    |         |

Key Column(s): JobCode

| an IN subquery | `proc sql;
|               | select empid, lastname, firstname, 
|               | city, state 
|               | from sasuser.staffmaster 
|               | where empid in 
|               | (select empid 
|               | from sasuser.payrollmaster 
|               | where salary>40000); |

Key Column(s): EmpID

| a correlated subquery, in which the column being compared with the correlated reference is indexed | `proc sql;
|                                                                                     | select lastname, firstname 
|                                                                                     | from sasuser.staffmaster 
|                                                                                     | where 'NA'=
|                                                                                     | (select jobcategory 
|                                                                                     | from sasuser.supervisors 
|                                                                                     | where staffmaster.empid = supervisors.empid); |

Key Column(s): Supervisors.EmpID

<table>
<thead>
<tr>
<th>a join in which</th>
</tr>
</thead>
<tbody>
<tr>
<td>the join expression contains the equals (=) operator (an equijoin)</td>
</tr>
<tr>
<td>all the columns in the join expression are indexed in one of the tables being joined.</td>
</tr>
</tbody>
</table>

Key Column(s): Payrollmaster.EmpID or Staffmaster.EmpID

Benefits of Using an Index

For PROC SQL, there are three main benefits to using an index to access data directly (instead of reading the data sequentially):

- A small subset of data (<15% of rows) can be accessed more quickly. (As the size of the subset increases, the advantage of using an index decreases.)
- Equijoins can be performed without internal sorts.
- Uniqueness can be enforced by creating a unique index.

Example: Using an Index to Access a Small Subset of Data

Suppose you are writing a query that references the table Work.Payrollmaster. (Work.Payrollmaster is a duplicate of the table Sasuser.Payrollmaster.)
Creating and Managing Indexes Using PROC SQL

**Guidelines for Creating Indexes**

*Work.Payrollmaster* stores payroll information for employees, and it has a simple index defined on the column *JobCode*. Your query’s WHERE clause expression references the key column:

```sql
proc sql;
    select empid, jobcode, salary
    from work.payrollmaster
    where jobcode='FA3'
    order by empid;
```

If the value of *JobCode* for most of the rows in the table is *FA3*, then the use of an index will not significantly improve the efficiency of the following query. In fact, performance might be degraded.

However if only 10% of the rows have a value of *FA3*, then PROC SQL can process the query more efficiently by using the index.

*Note:* In this chapter, if you want to submit any sample code that references a temporary table (a table that is stored in the *Work* library), you will first need to create the temporary table by copying the table in the *Sasuser* library that has the same name.

---

**Understanding the Costs of Using an Index**

When you are deciding whether to create an index, you should consider the associated increase in resource usage, which includes the following:

- **Additional CPU time** is necessary to create an index, to maintain the index when the table is modified, and to use an index to read a row from a table.
- Using an index to read rows from a table might require **additional I/O (input/output) requests** when compared to reading the table sequentially.
- Using an index requires **additional memory** for buffers into which the index pages and code are loaded for processing.
- **Additional disk space** is required to store the index file, which can show up as a separate file (in the Windows and UNIX operating environments, for example) or can appear to be part of the data file (in the z/OS (OS/390) operating environment).

**Guidelines for Creating Indexes**

To use indexes effectively, follow these guidelines for creating indexes:

- Keep the number of indexes to a minimum to reduce disk storage and update costs.
- Do not create an index for small tables. Sequential access is faster on small tables.
- Do not create an index based on columns that have a very small number of distinct values (low cardinality), for example, a *Gender* column that contains only the two values *Male* and *Female*.
- Use indexes for queries that retrieve a relatively small subset of rows — that is, less than 15%.
- Do not create more than one index that is based on the same column as the primary key.

*Tip:* Many factors affect the processing of SAS programs. The most accurate way to find out whether to create an index for a particular table or column is to perform benchmarking tests.
Creating an Index

To create an index on one or more columns of a table, use the CREATE INDEX statement.

General form, CREATE INDEX statement:

```
CREATE <UNIQUE> INDEX index-name
    ON table-name (column-name-1, ..., column-name-n);
```

where

- UNIQUE is a keyword that specifies that all values of the column(s) specified in the statement must be unique.
- `index-name` specifies the name of the index to be created. If you are creating an index on one column only, then `index-name` must be the same as `column-name-1`. If you are creating an index on more than one column, then `index-name` cannot be the same as the name of any existing column or index in the table.
- `table-name` specifies the name of the table on which the index will be created.
- `column-name` specifies a column to be indexed. Columns can be specified in any order; however, column order is important for data retrieval. The first-named column is the primary key, the second-named column is the secondary key, and so on.

**Tip:** When creating a composite index, specify the columns in the same order as you would specify them in an ORDER BY clause.

**Tip:** You can achieve improved index performance if you create the index on a pre-sorted table.

SAS maintains indexes for all changes to the table, whether the changes originate from PROC SQL or some other source, as long as the entire table is not re-created. If you alter a column's definition or update its values, then SAS will update the indexes also. However, if a key column in a table is dropped (deleted), then the index on that column is also dropped.

Creating Multiple Indexes

You cannot create multiple simple indexes that are based on the same column or multiple composite indexes that are based on the same set of columns. Although it is possible to create both a simple and a composite index on the same column, it is usually not advantageous to do this. If a simple index is defined on a column and that column is also the primary key in a composite index, PROC SQL will use the composite index in processing a query that references that column.

You can create multiple indexes on the same table, but you must use a separate CREATE INDEX statement for each index that you want to create.
Example: Creating a Simple Index

The following PROC SQL step uses the CREATE INDEX statement to create a simple, unique index that is based on the column `EmpID` in the temporary table `Work.Payrollmaster`. (`Work.Payrollmaster` is a duplicate of the table `Sasuser.Payrollmaster`.)

```sql
proc sql;
    create unique index EmpID
    on work.payrollmaster(empid);
```

The specified index name (`EmpID`) must be the same as the name of the key column. When this step is submitted, the SAS log displays the following message.

Table 6.2 SAS Log

| NOTE: Simple index EmpID has been defined. |

Example: Creating a Composite, Unique Index

The following PROC SQL step uses the CREATE INDEX statement to create the composite, unique index `daily` on the columns `FlightNumber` and `Date`:

```sql
proc sql;
    create unique index daily
    on work.marchflights(flightnumber,date);
```

When this step is submitted, the SAS log displays the following message.

Table 6.3 SAS Log

| NOTE: Composite index daily has been defined. |

Note: The note in the SAS log displays the index name exactly as you specified it. In this example, the index name `daily` was specified in lowercase. In the previous example, the index name `EmpID` was specified in mixed case. However, the use of uppercase and lowercase for index names is not significant because SAS recognizes index names regardless of how they are formatted in code.

If the set of key columns `FlightNumber` and `Date` had duplicate values, the index would not be created. Instead, the SAS log would display a message like the following.

Table 6.4 SAS Log

| ERROR: Duplicate values not allowed on index daily for file MARCHFLIGHTS. |

Displaying Index Specifications

Sometimes you want to know whether an existing table has any indexes. To display a CREATE INDEX statement in the SAS log for each index that is defined for one or more specified tables, you can use the DESCRIBE TABLE statement. (The DESCRIBE
TABLE statement also writes a CREATE TABLE statement to the SAS log for each specified table.

General form, DESCRIBE TABLE statement:

```
DESCRIBE TABLE table-name-1<, ... table-name-n>;
```

where

- `table-name` specifies the table to be described as one of the following:
  - a one-level name
  - a two-level `libref.table` name
  - a physical pathname that is enclosed in single quotation marks.

If a specified table has no indexes defined, a CREATE INDEX statement will not appear.

### Alternatives to the DESCRIBE TABLE Statement

The DESCRIBE TABLE statement is only one of several methods that can be used to list information about indexes that are defined on a table. One alternative is to query the special table `Dictionary.Indexes`, which contains information about indexes that are defined for all tables that are known to the current SAS session. (`Dictionary.Indexes` is one of many read-only dictionary tables that are created at PROC SQL initialization. These tables contain information about SAS data libraries, SAS macros, and external files that are in use or available in the current SAS session.)

You can also use other SAS procedures, such as PROC CONTENTS and PROC DATASETS, to generate a report that contains information about indexes.

*Note:* To learn more about the use of dictionary tables, see Chapter 8, “Managing Processing Using PROC SQL,” on page 261. To learn more about using PROC CONTENTS and PROC DATASETS, see Chapter 13, “Creating Samples and Indexes,” on page 451.

### Example

Earlier in this chapter, the following code was used to create a unique composite index named `daily` on the columns `FlightNumber` and `Date` in the temporary table `Marchflights`.

```sql
proc sql;
create unique index daily
  on work.marchflights(flightnumber,date);
```

The following DESCRIBE TABLE statement writes a CREATE INDEX statement to the SAS log (after the CREATE TABLE statement) for the table `Marchflights`:

```sql
proc sql;
  describe table marchflights;
```
Table 6.5  SAS Log

NOTE: SQL table WORK.MARCHFLIGHTS was created like:

```sql
create table WORK.MARCHFLIGHTS( bufsize=8192 )
(
    Date num format=DATE9. informat=DATE9.,
    DepartureTime num format=TIME5. informat=TIME5.,
    FlightNumber char(3),
    Origin char(3),
    Destination char(3),
    Distance num, Mail num,
    Freight num, Boarded num,
    Transferred num,
    NonRevenue num,
    Deplaned num,
    PassengerCapacity num
);
create unique index daily on WORK.MARCHFLIGHTS(FlightNumber,Date);
```

If the table Marchflights had no index defined, no CREATE INDEX statement would appear in the SAS log.

Managing Index Usage

To manage indexes effectively, it is important to know

- how SAS decides whether to use an index and which index to use
- how to determine whether SAS is using an index
- how to control whether SAS uses an index, or which index it uses.

Understanding How SAS Decides Whether to Use an Index

By default, each time you submit a query (or other SAS program) that contains a WHERE expression, SAS decides whether to use an index or to read all the observations in the data file sequentially. To make this decision, SAS does the following:

1. Identifies an available index or indexes.
2. Estimates the number of rows that would be qualified. If multiple indexes are available, SAS selects the index that it estimates will return the smallest subset of rows.
3. Compares resource usage to decide whether it is more efficient to satisfy the WHERE expression by using the index or by reading all the observations sequentially.

Next, let's see how you can find out whether SAS is using an index.

Determining Whether SAS Is Using an Index

After you create an index, it is important to monitor whether the index is being used. If an index is not being used, the costs of maintaining the index might be greater than the benefits, and you should consider dropping (deleting) the index.
By default, when a PROC SQL query or any other program is submitted in SAS, only notes, warnings, and error messages are written to the SAS log. To display additional messages, such as information about indexes that have been defined and that have been used in processing the program, specify the SAS system option `MSGLEVEL=I`. You specify the `MSGLEVEL=` option in the OPTIONS statement, before the PROC SQL statement.

```
General form, MSGLEVEL= option:

    OPTIONS MSGLEVEL=N | I;

where

N   displays notes, warnings, and error messages only. This is the default.
I   displays additional notes pertaining to index usage, merge processing, and sort utilities along with standard notes, warnings, and error messages.
```

Usually, the option `MSGLEVEL=` is set to I for debugging and testing, and set to N for production jobs.

---

**Example: Query That Uses an Index**

Suppose you are writing a PROC SQL query that references the temporary table `Marchflights`. Earlier in this chapter, a unique composite index named `daily` was created on the columns `FlightNumber` and `Date` in `Marchflights`. The WHERE expression in your query specifies the key column `FlightNumber`. To determine whether PROC SQL uses the index `daily` when your query is processed, you specify `MSGLEVEL=I` before the query:

```
options msglevel=i;
proc sql;
select *
from marchflights
where flightnumber='182';
```

The message in the SAS log shows that the index was used in processing.

Table 6.6  SAS Log

INFO: Index daily selected for WHERE clause optimization.

---

**Example: Query That Does Not Use an Index**

Suppose you submit a different query that also references the key column `FlightNumber`:

```
proc sql;
select *
from marchflights
where flightnumber in ('182','202');
```
In this example, the SAS log shows that the query does not use the index.

Table 6.7  SAS Log

| INFO: Index daily not used. Sorting into index order may help. |
| INFO: Index daily not used. Increasing bufno to 8 may help. |

Note: SAS Version 8 displays informational messages that indicate when an index is used, but does not display messages that indicate when an index is not used. △

Tip: Because the OPTIONS statement is global, the settings remain in effect until you modify them or until you end your SAS session. Therefore, you do not need to specify MSGLEVEL=I in this second query or any subsequent queries until you want to change the setting or until your SAS session ends.

Controlling Index Usage

In general, it is recommended that you allow SAS to decide whether to use an index, or which index to use, in processing a PROC SQL query (or other SAS program). However, in some situations, such as testing, you might find it useful to control the use of indexes by SAS.

To control index usage, use the IDXWHERE= and IDXNAME= SAS data set options to override the default settings. You can use either of these options, but you cannot use both options at the same time. As with other SAS data set options, you specify the IDXWHERE= or IDXNAME= option in parentheses after the table name in the FROM clause of a PROC SQL query.

Using IDXWHERE= to Direct SAS to Use or Not to Use an Index

The IDXWHERE= option enables you to override the decision that SAS makes about whether to use an index.

General form, IDXWHERE= option:

\[
\text{IDXWHERE=} \text{YES | NO};
\]

where

YES

tells SAS to choose the best index to optimize a WHERE expression, and to disregard the possibility that a sequential search of the table might be more resource-efficient.

NO

tells SAS to ignore all indexes and satisfy the conditions of a WHERE expression with a sequential search of the table.

Example

In an earlier example, you used the option MSGLEVEL=I to verify that PROC SQL does use an index to process the following query:

```sql
options msglevel=i;
proc sql;
```
Using IDXNAME= to Direct SAS to Use a Specified Index

The IDXNAME= option directs SAS to use an index that you specify, even if SAS would have selected not to use an index or to use a different index.

General form, IDXNAME= option:

```
IDXNAME=index-name;
```

where

`index-name`

specifies the name of the index that should be used for processing.

SAS uses the specified index if the following conditions are true:

- The specified index must exist.
- The specified index must be suitable by having at least its first or only column match a condition in the WHERE expression.

Example

In an earlier example, a composite index named `daily` was defined on the columns `FlightNumber` and `Date` in the temporary table `Marchflights`. Suppose you create a second index, a simple index, on the column `Date` (the secondary key in the composite index) by using the following PROC SQL step:

```
proc sql;
  create index Date
  on work.marchflights(Date);
```

Next, you submit the following query:

```
proc sql;
  select *
  from marchflights
  where flightnumber='182';
```

To force SAS to ignore the index and to process the rows of the table sequentially, specify IDXWHERE=NO in the query:

```
proc sql;
  select *
  from marchflights (idxwhere=no)
  where flightnumber='182';
```

A message in the SAS log indicates that SAS was forced to process the data sequentially.

Table 6.8  SAS Log

```
INFO: Data set option (IDXWHERE=NO) forced a sequential pass of the data rather than use of an index for WHERE-clause processing.
```
from marchflights
where date='01MAR2000'd;

The WHERE clause in this query references the key column Date. By default, SAS decides whether to use an index and, if an index is used, which index to use. The SAS log indicates that, with both a simple and a composite index defined on Date, PROC SQL used the simple index Date to process the query.

Table 6.9  SAS Log

INFO: Index Date selected for WHERE clause optimization.

Note: This example assumes that the option MSGLEVEL=I, which was specified in the previous example, is still in effect.

You decide that you want to force PROC SQL to use the index daily instead of Date, so you add IDXNAME= to your query:

```sql
proc sql;
  select *
    from marchflights (idxname=daily)
    where flightnumber='182';
```

After this query is submitted, a message in the SAS log indicates that PROC SQL used the index daily:

Table 6.10  SAS Log

INFO: Index daily selected for WHERE clause optimization.

### Dropping Indexes

To drop (delete) one or more indexes, use the **DROP INDEX statement**.

General form, DROP INDEX statement:

```
DROP INDEX index-name-1 <, ...index-name-2>
FROM table-name;
```

where

- **index-name** specifies an index that exists.
- **table-name** specifies a table that contains the specified index(es). The table-name can be one of the following:
  - a one-level name
  - a two-level libref.table name
  - a physical pathname that is enclosed in single quotation marks.
Example: Dropping a Composite Index

The following PROC SQL step uses the DROP INDEX statement to drop the composite, unique index `daily` from the temporary table `Marchflights`. (This index was created in an example earlier in this chapter.)

```
proc sql;
  drop index daily
  from work.marchflights;
```

When this step is submitted, the SAS log displays a message indicating that the index has been dropped.

Table 6.11  SAS Log

```
NOTE: Index daily has been dropped.
```
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Understanding Indexes

An index is an auxiliary file that is defined on one or more of a table's columns, which are called key columns. The index stores the unique column values and a system of directions that enable access to rows in that table by index value. When an index is used to process a PROC SQL query, PROC SQL accesses directly (without having to read all the prior rows) instead of sequentially.

- You can create two types of indexes:
  - simple index (an index on one column)
  - composite index (an index on two or more columns).

- You can define either type of index as a unique index, which requires that values for the key column(s) be unique for each row.

Deciding Whether to Create an Index

When deciding whether to create an index, you must weigh any benefits in performance improvement against the costs of increased resource usage. Certain classes of PROC SQL queries can be optimized by using an index. To optimize the performance of your PROC SQL queries, you can follow some basic guidelines for creating indexes.

Creating an Index

To create an index on one or more columns of a table, use the CREATE INDEX statement. To specify a unique index, you add the keyword UNIQUE.

Displaying Index Specifications

To display a CREATE INDEX statement in the SAS log for each index that is defined for one or more specified tables, use the DESCRIBE TABLE statement.

Managing Index Usage

To manage indexes effectively, it is important to know how SAS decides whether to use an index and which index to use.

- To find out whether an index is being used, specify the SAS option MSGLEVEL=I in an OPTIONS statement before the PROC SQL statement. This option enables SAS to write informational messages about index usage (and other additional information) to the SAS log. The default setting MSGLEVEL=N displays notes, warnings, and error messages only.
- To force SAS to use the best available index, to use a specific index, or not to use an index at all, include either the SAS data set option IDXWHERE= or IDXNAME= in your PROC SQL query.
Dropping Indexes

To drop (delete) one or more indexes, use the DROP INDEX statement.

Syntax

OPTIONS MSGLEVEL=N | I;

PROC SQL;

CREATE <UNIQUE> INDEX index-name
    ON table-name (column-name-1<, ...column-name-n>);

DESCRIBE TABLE table-name-1<, ... table-name-n>;

SELECT column-1<, ... column-n>
    FROM table-1 (IDXWHERE=Y | N)
    <WHERE expression>;

SELECT column-1<, ... column-n>
    FROM table-1 (IDXNAME=index-name)
    <WHERE expression>;

DROP INDEX table-name-1<, ... table-name-n>;
    FROM table-name;

QUIT;

Sample Programs

Creating a Simple, Unique Index and a Composite Index

proc sql;
    create unique index EmpID
        on work.payrollmaster(empid);
    create index daily
        on work.marchflights(flightnumber,date);
quit;

Displaying Index Specifications

proc sql;
    describe table marchflights;
quit;
Determining Whether SAS Is Using an Index

options msglevel=i;
proc sql;
   select *
       from marchflights
       where flightnumber='182';
quit;

Directing SAS to Ignore All Indexes

proc sql;
   select *
       from marchflights (idxwhere=no)
       where flightnumber='182';
quit;

Directing SAS to Use a Specified Index

proc sql;
   select *
       from marchflights (idxname=daily)
       where flightnumber='182';
quit;

Dropping an Index

proc sql;
   drop index daily
       from work.marchflights;
quit;

Points to Remember

☐ An index cannot be created on a view.
☐ Keep the number of indexes to a minimum to reduce disk storage and update costs.
☐ Do not create an index for small tables; sequential access is faster on small tables.
☐ Do not create an index based on columns that have a very small number of distinct values (low cardinality), for example, a Gender column that contains only the two values Male and Female.
☐ Use indexes for queries that retrieve a relatively small subset of rows — that is, less than 15%.
☐ Do not create more than one index that is based on the same column as the primary key.
Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Which of the following will create an index on the column EmpID for the table Sasuser.Staffmaster?

   a proc sql;
      create simple index(empid)
      on sasuser.staffmaster;

   b proc sql;
      create empid index
      on sasuser.staffmaster(empid);

   c proc sql;
      create simple index
      on empid from sasuser.staffmaster;

   d proc sql;
      create index empid
      on sasuser.staffmaster(empid);

2 Which keyword must you add to your index definition in the CREATE INDEX statement to ensure that no duplicate values of the key column can exist?

   a KEY
   b UNIQUE
   c NODUPS
   d NODUPKEY

3 Which of the following will create a composite index for the table Sasuser.Flightdelays? (Sasuser.Flightdelays contains the following columns: Date, FlightNumber, Origin, Destination, DelayCategory, DestinationType, DayOfWeek, and Delay.)

   a proc sql;
      create index destination
      on sasuser.flightdelays(flightnumber, destination);

   b proc sql;
      create composite index places
      on sasuser.flightdelays (flightnumber, destination);

   c proc sql;
      create index on flightnumber,destination
      from sasuser.flightdelays (places);

   d proc sql;
      create index places
      on sasuser.flightdelays (flightnumber, destination);
4 Which of the following will write a message to the SAS log that shows whether PROC SQL has used an index?

a options msglevel=i;
   proc sql;
   select *
     from sasuser.internationalflights
     where date between '01mar2000'd
     and '07mar2000'd;

b options index=yes;
   proc sql;
   select *
     from sasuser.internationalflights
     where date between '01mar2000'd
     and '07mar2000'd;

c proc sql;
   select * (idxwhere=yes)
     from sasuser.internationalflights
     where date between '01mar2000'd
     and '07mar2000'd;

d proc sql;
   select * (msglevel=i)
     from sasuser.internationalflights
     where date between '01mar2000'd
     and '07mar2000'd;

5 Which of the following will drop (delete) an index from a table?

a proc sql;
   drop composite index flights
     from sasuser.marchflights;

b proc sql;
   delete index flights
     on sasuser.staffmaster(flightnumber, date);

c proc sql;
   drop index flights
     from sasuser.marchflights;

d proc sql;
   delete index
     on sasuser.marchflights(flightnumber,
                              flightdate);

6 Which of the following statements will show you all the indexes that are defined for a table?

a DESCRIBE INDEX
b DESCRIBE TABLE
c SELECT
d IDXNAME

7 What is the purpose of specifying the data set option IDXWHERE=YES?

a It forces SAS to use the best available index to process the WHERE expression.
b It creates an index from the expression in the WHERE clause.
c It writes messages about index usage to the SAS log.
8 Which of the following is false regarding the use of an index?
   
   a Equijoins can be performed without internal sorts.
   b Indexes provide fast access to a small subset of data.
   c Indexes can be created for numeric columns only.
   d Indexes can enforce uniqueness.

9 Using an index is not likely to optimize a PROC SQL query in which of the following situations?
   
   a The query contains an IN subquery that references the key column.
   b The key column is specified in a WHERE clause expression that contains a comparison operator, the TRIM or SUBSTR function, the CONTAINS operator, or the LIKE operator.
   c The query is an equijoin, and all the columns in the join expression are indexed in one of the tables being joined.
   d The key column is specified only in a SELECT clause.

10 Which of the following is false regarding the IDXNAME= data set option?
   
   a The specified index must exist.
   b The specified index must be suitable by having at least its first or only column match a condition in the WHERE expression.
   c The option allows you to create and name an index on the table.
   d The option directs SAS to use an index that you specify.
CHAPTER
7
Creating and Managing Views
Using PROC SQL

Overview 244
Introduction 244
Objectives 244
Prerequisites 244
Creating and Using PROC SQL Views 245
PROC SQL Views 245
Creating PROC SQL Views 245
Example 246
Using PROC SQL Views 246
Displaying the Definition for a PROC SQL View 247
Example 248
Managing PROC SQL Views 248
Guidelines for Using PROC SQL Views 248
Omitting the Libref 249
Using an Embedded LIBNAME Statement 249
Example 250
Creating a View to Enhance Table Security 250
Example 250
Updating PROC SQL Views 251
Example 251
Dropping PROC SQL Views 253
Example 253
Summary 254
Text Summary 254
Using PROC SQL Views 254
Creating SQL Views 254
Displaying the Definition for a PROC SQL View 254
Managing PROC SQL Views 254
Updating PROC SQL Views 255
Dropping PROC SQL Views 255
Syntax 255
Sample Programs 256
Creating a PROC SQL View 256
Displaying the Definition for a PROC SQL View 256
Using a PROC SQL View in a Query 256
Updating a PROC SQL View 256
Dropping a PROC SQL View 256
Points to Remember 257
Quiz 257
Overview

Introduction

A PROC SQL view is a stored query expression that reads data values from its underlying files, which can include SAS data files, DATA step views, other PROC SQL views, or DBMS data.

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 table(s) or view(s).

PROC SQL views

- can be used in SAS programs in place of an actual SAS data file
- can be joined with tables or other views
- can be derived from one or more tables, PROC SQL views, or DATA step views
- can access data from a SAS data set, a DATA step view, a PROC SQL view, or a relational database table
- extract underlying data, which enables you to access the most current data.

Objectives

In this chapter, you learn to

- create and use PROC SQL views
- display the definition for a PROC SQL view
- manage PROC SQL views
- update PROC SQL views
- drop (delete) PROC SQL views.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25

```sql
proc sql;
create view sasuser.raisev as
  select empid, jobcode,
  salary format=dollar12.2,
  salary/12 as MonthlySalary
  format=dollar12.
  from payrollmaster;
select *
from sasuser.raisev
where jobcode in ('PT2','PT3');
```
Creating and Using PROC SQL Views

PROC SQL Views

A PROC SQL view is a stored query that is executed when you use the view in a SAS procedure, DATA step, or function. A view contains only the descriptor and other information required to retrieve the data values from other SAS files (SAS data files, DATA step views, or other PROC SQL views) or external files (DBMS data files). The view contains only the logic for accessing the data, not the data itself.

Because PROC SQL views are not separate copies of data, they are referred to as virtual tables. They do not exist as independent entities like real tables. However, views use the same naming conventions as tables and can be used in SAS programs in place of an actual SAS table. Like tables, views are considered to be SAS data sets.

Views are useful because they

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

Creating PROC SQL Views

You use the CREATE VIEW statement to create a view.

General form, CREATE VIEW statement:

```
CREATE VIEW proc-sql-view AS
  SELECT column-1<, ... column-n>
  FROM table-1 | view-1<, ... table-n | view-n>
  <optional query clauses>;
```

where

- `proc-sql-view` specifies the name of the PROC SQL view that you are creating.
- `SELECT` specifies the column(s) that will appear in the table.
- `FROM` specifies the table(s) or view(s) to be queried.
- `optional query clauses` are used to refine the query further and include the WHERE, GROUP BY, HAVING, and ORDER BY clauses.

A PROC SQL view derives its data from the tables or views that are listed in the FROM clause. The data that is accessed by a view is a subset or superset of the data
that is in its underlying table(s) or view(s). When a view is referenced by a SAS procedure or in a DATA step, it is executed and, conceptually, an internal table is built. PROC SQL processes this internal table as if it were any other table.

Example

The following PROC SQL step creates a view that contains information for flight attendants. The view always returns the employee’s age as of the current date.

The view Sasuser.Faview creates a virtual table from the accompanying SELECT statement. Although the underlying tables, Sasuser.Payrollmaster and Sasuser.Staffmaster, can change, the instructions that comprise the view stay constant. The libref specified in the FROM clause is optional. It is assumed that the contributing tables are stored in the same library as the view itself, unless otherwise specified.

```
proc sql;
    create view sasuser.faview as
    select lastname, firstname, gender,
        int((today()-dateofbirth)/365.25) as Age,
        substr(jobcode,3,1) as Level,
        salary
    from sasuser.payrollmaster,
        sasuser.staffmaster
    where jobcode contains 'FA' and
        staffmaster.empid=payrollmaster.empid;
```

When this PROC SQL step is submitted, SAS does not actually execute the SELECT statement that follows the AS keyword, but partially compiles and stores the SELECT statement in a data file with a member type of VIEW. A message in the SAS log confirms that the view has been defined.

Table 7.1 SAS Log

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>proc sql;</td>
<td>create view sasuser.faview as select lastname, firstname, gender, int((today())-dateofbirth)/365.25 as Age, substr(jobcode,3,1) as Level, salary from sasuser.payrollmaster, sasuser.staffmaster where jobcode contains ‘FA’ and staffmaster.empid=payrollmaster.empid;</td>
</tr>
</tbody>
</table>

**Tip**: It is helpful to give a PROC SQL view a name that easily identifies it as a view. For example, Faview or Fav.

**Note**: In the Windows and UNIX operating environments, the default extension for PROC SQL views (and DATA step views) is .sas7bview.

Using PROC SQL Views

You can use a view in a subsequent PROC SQL step, or later in the same step, just as you would use an actual SAS table. In the following example, the PROC SQL view
Sasuser.Faview is used in a query. Because the query stored in the view calculates the age of each flight attendant based on the current date, the resulting output from this PROC SQL step shows each flight attendant’s age as of the current date. If Sasuser.Faview were a static table, instead of a view, the age shown for each flight attendant would never change.

```sql
proc sql;
  select *
  from sasuser.faview;
```

Partial output is shown below.

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
<th>Gender</th>
<th>Age</th>
<th>Level</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARTHUR</td>
<td>BARBARA</td>
<td>F</td>
<td>39</td>
<td>3</td>
<td>$46,040</td>
</tr>
<tr>
<td>CAHILL</td>
<td>MARSHALL</td>
<td>M</td>
<td>44</td>
<td>2</td>
<td>$40,001</td>
</tr>
<tr>
<td>CARTER</td>
<td>DOROTHY</td>
<td>F</td>
<td>44</td>
<td>3</td>
<td>$46,346</td>
</tr>
<tr>
<td>COOPER</td>
<td>ANTHONY</td>
<td>M</td>
<td>45</td>
<td>3</td>
<td>$45,104</td>
</tr>
<tr>
<td>DEAN</td>
<td>SHARON</td>
<td>F</td>
<td>43</td>
<td>3</td>
<td>$46,767</td>
</tr>
<tr>
<td>DUNLAP</td>
<td>DONNA</td>
<td>F</td>
<td>36</td>
<td>2</td>
<td>$40,443</td>
</tr>
<tr>
<td>EATON</td>
<td>ALICIA</td>
<td>F</td>
<td>43</td>
<td>2</td>
<td>$38,902</td>
</tr>
<tr>
<td>FIELDS</td>
<td>DIANA</td>
<td>F</td>
<td>46</td>
<td>1</td>
<td>$32,448</td>
</tr>
<tr>
<td>FLETCHER</td>
<td>MARIE</td>
<td>F</td>
<td>40</td>
<td>1</td>
<td>$31,436</td>
</tr>
<tr>
<td>GOMEZ</td>
<td>ALAN</td>
<td>M</td>
<td>34</td>
<td>1</td>
<td>$31,175</td>
</tr>
</tbody>
</table>

Tip: You can use PROC SQL views in other SAS procedures and DATA steps. In the following example, PROC TABULATE calculates the flight attendants’ mean age by level, using the view Sasuser.Faview:

```sas
proc tabulate data=sasuser.faview;
  class level;
  var age;
  table level*age*mean;
run;
```

Note: The values for the variable Age will vary, because the calculation is dependent on the date on which the code is executed.

Note: For information about the TABULATE procedure, see the SAS documentation.

### Displaying the Definition for a PROC SQL View

You can use a DESCRIBE VIEW statement to display a definition of a view in the SAS log.
Example

The following PROC SQL step writes the view definition for Sasuser.Faview to the SAS log:

```
proc sql;
   describe view sasuser.faview;
```

Table 7.2  SAS Log

```
NOTE: SQL view SASUSER.FAVIEW is defined as:
   select lastname, firstname, gender,
       INT((TODAY()-dateofbirth)/365.25) as Age,
       SUBSTR(jobcode, 3, 1) as Level, salary
   from SASUSER.PAYROLLMASTER, SASUSER.STAFFMASTER
   where jobcode contains 'FA' and
       (staffmaster.empid=payrollmaster.empid);
```

Managing PROC SQL Views

Guidelines for Using PROC SQL Views

When you are working with PROC SQL views, it is best to follow these guidelines:

- Avoid using an ORDER BY clause in a view definition, which causes the data to be sorted every time the view is executed. Users of the view might differ in how or whether they want the data to be sorted, so it is more efficient to specify an ORDER BY clause in a query that references the view.

- If the same data is used many times in one program or in multiple programs, it is more efficient to create a table rather than a view because the data must be accessed at each view reference. (This table can be a temporary table in the Work library.)
Avoid creating views that are based on tables whose structure might change. A view is no longer valid when it references a nonexistent column.

If a view resides in the same SAS data library as the contributing table(s), it is best to specify a one-level name in the FROM clause.

---

**Omitting the Libref**

The default libref for the table or tables in the FROM clause is the libref of the library that contains the view. Using a one-level name in the FROM clause prevents you from having to change the view if you assign a different libref to the SAS data library that contains the view and its contributing table or tables.

The following PROC SQL step creates the view `Sasuser.Payrollv`. The FROM clause specifies a two-level name for the contributing table, `Sasuser.Payrollmaster`. However, it isn’t necessary to specify the libref `Sasuser` because the contributing table is assumed to be stored in the same library as the view.

```sql
proc sql;
  create view sasuser.payrollv as
    select *
    from sasuser.payrollmaster;
```

When the one-level name `Payrollmaster` is used in the FROM clause, `Sasuser.Payrollmaster` is being specified, though it appears that `Work.Payrollmaster` is being specified.

```sql
proc sql;
  create view sasuser.payrollv as
    select *
    from payrollmaster;
```

**CAUTION:**

If you are creating a view that is stored in a different library than the table(s) referenced in the FROM clause, you must specify a two-level name for the table(s).

---

**Using an Embedded LIBNAME Statement**

As an alternative to omitting the libref in the FROM clause, you can embed a LIBNAME statement in a USING clause to store a SAS libref in a view. Embedding a LIBNAME statement is a more flexible approach because

- it can be used regardless of whether the view and the underlying tables reside in the same library
- it avoids the confusion that might arise if a libref is omitted from a table name in the FROM clause.

An embedded LIBNAME statement can only be used with a PROC SQL view. A libref created with an embedded LIBNAME statement will not conflict with an identically named libref in the SAS session.
General form, USING clause:

\[
\text{USING } \text{libname-clause-1, ... libname-clause-n};
\]

where

.libname-clause

is one of the following:

- a valid LIBNAME statement
- a valid SAS/ACCESS LIBNAME statement.

---

**CAUTION:**

The USING clause must be the last clause in the CREATE VIEW statement.

---

**Example**

In the following example, while the view `Sasuser.Payrollv` is executing in the PROC PRINT step, the libref `Airline`, which is specified in the USING clause, becomes active. This overrides the earlier assignment of the libref in the LIBNAME statement for the duration of the view’s execution. After the view executes, the original libref assignment is re-established and the embedded assignment is cleared.

```sql
libname airline 'SAS-library one';
proc sql;
    create view sasuser.payrollv as
        select * from airline.payrollmaster
    using libname airline 'SAS-library two';
quit;
proc print data=sasuser.payrollv;
run;
```

---

**Creating a View to Enhance Table Security**

One advantage of PROC SQL views is that they can bring data together from separate sources. This enables views to be used to shield sensitive or confidential columns from some users while enabling the same users to view other columns in the same table.

**CAUTION:**

Although PROC SQL views can be used to enhance table security, it is strongly recommended that you use the security features that are available in your operating environment to maintain table security.

---

**Example**

The following PROC SQL step creates the view `Manager.Infoview`. The view accesses data about flight attendants that is stored in three SAS libraries: `Fa1`, `Fa2`, and `Fa3`. The `Fa1`, `Fa2`, and `Fa3` libraries can be assigned access privileges at the operating system level to prevent

- Level 1 flight attendants from reading the data stored in the `Fa2` and `Fa3` libraries
Level 2 flight attendants from reading the data stored in the Fa1 and Fa3 libraries
Level 3 flight attendants from reading the data stored in the Fa1 and Fa2 libraries.

Access privileges can also be assigned to permit managers (who are authorized to access all SAS libraries) to view all of the information.

```sql
proc sql;
create view manager.infoview as
    select *
    from fa1.info
outer union corr
    select *
    from fa2.info
outer union corr
    select *
    from fa3.info;
```

### Updating PROC SQL Views

You can update the data underlying a PROC SQL view using the INSERT, DELETE, and UPDATE statements under the following conditions:

- You can only update a single table through a view. The table cannot be joined or linked to another table, nor can it contain a subquery.
- You can update a column using the column’s alias, but you cannot update a derived column (a column that is produced by an expression).
- You can update a view that contains a WHERE clause. The WHERE clause can be specified in the UPDATE clause or in the view. You cannot update a view that contains any other clause such as an ORDER BY or a HAVING clause.
- You cannot update a summary view (a view that contains a GROUP BY clause).

Updating a view does not change the stored instructions for the view. Only the data in the underlying table(s) is updated.

#### Example

The following PROC SQL step creates the view Sasuser.Raisev, which includes the columns Salary and MonthlySalary. A subsequent query that references the view shows the columns.
proc sql;
create view sasuser.raisev as
  select empid, jobcode,
    salary format=dollar12.,
    salary/12 as MonthlySalary
  format=dollar12.
  from payrollmaster;
proc sql;
select *
  from sasuser.raisev
where jobcode in ('PT2','PT3');

Suppose you want to update the view to show a salary increase for employees whose job code is PT3. You can use an UPDATE statement to change the column Salary and a WHERE clause in the UPDATE clause to identify the rows where the value of JobCode equals PT3. Though MonthlySalary is a derived column and cannot be changed using an UPDATE statement, it will be updated because it is derived from Salary.

When the PROC SQL step is submitted, a note appears in the SAS log that indicates how many rows were updated:

proc sql;
update sasuser.raisev
  set salary=salary * 1.20
  where jobcode='PT3';

Table 7.3  SAS Log

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
<th>MonthlySalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1333</td>
<td>PT2</td>
<td>$124,648</td>
<td>$10,387</td>
</tr>
<tr>
<td>1404</td>
<td>PT2</td>
<td>$127,928</td>
<td>$10,661</td>
</tr>
<tr>
<td>1118</td>
<td>PT3</td>
<td>$155,931</td>
<td>$12,994</td>
</tr>
<tr>
<td>1410</td>
<td>PT2</td>
<td>$119,659</td>
<td>$9,969</td>
</tr>
<tr>
<td>1777</td>
<td>PT3</td>
<td>$153,482</td>
<td>$12,799</td>
</tr>
<tr>
<td>1106</td>
<td>PT2</td>
<td>$125,495</td>
<td>$10,467</td>
</tr>
<tr>
<td>1442</td>
<td>PT2</td>
<td>$119,390</td>
<td>$9,963</td>
</tr>
<tr>
<td>1470</td>
<td>PT2</td>
<td>$117,094</td>
<td>$9,724</td>
</tr>
<tr>
<td>1390</td>
<td>PT2</td>
<td>$120,254</td>
<td>$10,021</td>
</tr>
<tr>
<td>1107</td>
<td>PT2</td>
<td>$125,098</td>
<td>$10,407</td>
</tr>
<tr>
<td>1930</td>
<td>PT2</td>
<td>$119,259</td>
<td>$9,955</td>
</tr>
<tr>
<td>1938</td>
<td>PT2</td>
<td>$125,601</td>
<td>$10,463</td>
</tr>
</tbody>
</table>

Note: Remember that the rows were updated in the table that underlies the view Sasuser.Raisev. Δ

When you resubmit the query, the updated values for Salary and MonthlySalary appear in the rows where JobCode equals PT3:
PROC SQL Views

To drop (delete) a view, use the DROP VIEW statement.

General form, DROP VIEW statement:

```
DROP VIEW view-name-1 <,...,view-name-n>
```

where

- `view-name` specifies a SAS data view of any type (PROC SQL view or DATA step view) and can be one of the following:
  - a one-level name
  - a two-level `libref.view` name
  - a physical pathname that is enclosed in single quotation marks.

Example

The following PROC SQL step drops the view `Sasuser.Raisev`. After the step is submitted, a message appears in the SAS log to confirm that the view has been dropped.

```sql
proc sql;
  drop view sasuser.raisev;
```

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
<th>MonthlySalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1333</td>
<td>PT2</td>
<td>$124,000</td>
<td>$10,333</td>
</tr>
<tr>
<td>1404</td>
<td>PT2</td>
<td>$127,000</td>
<td>$10,583</td>
</tr>
<tr>
<td>1118</td>
<td>PT3</td>
<td>$187,117</td>
<td>$15,593</td>
</tr>
<tr>
<td>1410</td>
<td>PT2</td>
<td>$166,599</td>
<td>$13,883</td>
</tr>
<tr>
<td>1777</td>
<td>PT3</td>
<td>$184,175</td>
<td>$15,346</td>
</tr>
<tr>
<td>1108</td>
<td>PT2</td>
<td>$125,465</td>
<td>$10,427</td>
</tr>
<tr>
<td>1442</td>
<td>PT2</td>
<td>$116,260</td>
<td>$9,683</td>
</tr>
<tr>
<td>1470</td>
<td>PT2</td>
<td>$117,004</td>
<td>$9,704</td>
</tr>
<tr>
<td>1890</td>
<td>PT2</td>
<td>$130,254</td>
<td>$10,021</td>
</tr>
<tr>
<td>1107</td>
<td>PT2</td>
<td>$125,663</td>
<td>$10,437</td>
</tr>
<tr>
<td>1600</td>
<td>PT2</td>
<td>$116,260</td>
<td>$9,683</td>
</tr>
<tr>
<td>1928</td>
<td>PT2</td>
<td>$125,601</td>
<td>$10,483</td>
</tr>
</tbody>
</table>

Table 7.4   SAS Log

21 proc sql;
22 drop view sasuser.raisev;

NOTE: View SASUSER.RAISEV has been dropped.
Summary

This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Using PROC SQL Views

A PROC SQL view is a stored query that is executed when you use the view in a SAS procedure, DATA step, or function. A view contains only the descriptor and other information required to retrieve the data values from other SAS files (SAS data files, DATA step views, or other PROC SQL views) or external files (DBMS data files). When executed, a PROC SQL view's output can be a subset or superset of one or more underlying files. A view contains no data, but describes or defines data that is stored elsewhere.

PROC SQL views
- can be used in SAS programs in place of an actual SAS data file
- can be joined with tables or other views
- can be derived from one or more tables, PROC SQL views, or DATA step views
- extract underlying data, which enables you to access the most current data.

Because PROC SQL views are not separate copies of data, they are referred to as virtual tables. They do not exist as independent entities like real tables. However, views use the same naming conventions as tables and can be used in SAS programs in place of an actual SAS table. Like tables, views are considered to be SAS data sets.

Creating SQL Views

You use the CREATE VIEW statement to create a view. A PROC SQL view derives its data from the tables or views that are listed in the FROM clause. The data that is accessed by a view is a subset or superset of the data that is in its underlying tables(s) or view(s). When a view is referenced by a SAS procedure or in a DATA step, it is executed and, conceptually, an internal table is built. PROC SQL processes this internal table as if it were any other table. A view can be used in a subsequent PROC SQL step just as you would use an actual SAS table.

Displaying the Definition for a PROC SQL View

You can use a DESCRIBE VIEW statement to display a definition of a view in the SAS log.

Managing PROC SQL Views

The default libref for the table or tables in the FROM clause is the libref of the library that contains the view. Using a one-level name 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.
As a more flexible alternative to omitting the libref in the FROM clause, you can embed a LIBNAME statement in a USING clause if you want to store a SAS libref in a view. Embedding a LIBNAME statement in a USING clause does not conflict with an identically named libref in the SAS session.

One advantage of PROC SQL views is that they can bring data together from separate sources. This enables views to be used to shield sensitive or confidential columns from some users while enabling the same users to view other columns in the same table. Although PROC SQL views can be used to enhance table security, it is strongly recommended that you use the security features that are available in your operating environment to maintain table security.

Updating PROC SQL Views

You can update the data underlying a PROC SQL view using the INSERT, DELETE, and UPDATE statements under the following conditions:

- You can only update a single table through a view. The table cannot be joined or linked to another table, nor can it contain a subquery.
- You can update a column using the column’s alias, but you cannot update a derived column (a column that is produced by an expression).
- 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 an ORDER BY or a HAVING clause.
- You cannot update a summary view (a view that contains a GROUP BY clause).

Dropping PROC SQL Views

To drop (delete) a view, use the DROP VIEW statement.

Syntax

PROC SQL;

CREATE VIEW proc-sql-view AS
SELECT column-1<, ... column-n>
FROM table-1 | view-1<, ... table-n | view-n>
<optional query clauses>;
USING libname-clause<,...libname-clause>;

DESCRIBE VIEW proc-sql-view<,...proc-sql-view>;

INSERT INTO table-name | proc-sql-view
(SELECT-column-1<, ... target-column-n>)
SET column-1=value-1<, ... column-n=value-n>
<... SET column-1=value-1<, ... column-n=value-n>>;

DELETE FROM table-name | proc-sql-view
<WHERE expression>;


UPDATE table-name | proc-sql-view
   SET column-1=expression, ... column-n=expression>
   <WHERE expression>;

DROP VIEW view-name<,...view-name>;

QUIT;

---

Sample Programs

Creating a PROC SQL View

proc sql;
   create view sasuser.raisev as
      select empid, jobcode,
         salary format=dollar12.2,
         salary/12 as MonthlySalary
         format=dollar12.
      from payrollmaster
      using libname airline 'c:\data\ia';
   quit;

Displaying the Definition for a PROC SQL View

proc sql;
   describe view sasuser.raisev;
   quit;

Using a PROC SQL View in a Query

proc sql;
   select *
      from sasuser.raisev
      where jobcode in ('PT2','PT3');
   quit;

Updating a PROC SQL View

proc sql;
   update sasuser.raisev
      set salary=salary * 1.20
      where jobcode='PT3';
   quit;

Dropping a PROC SQL View

proc sql;
   drop view sasuser.raisev;
   quit;
Points to Remember

- Avoid using an ORDER BY clause in a view definition, which causes the data to be sorted every time the view is executed. Users of the view might differ in how or whether they want the data to be sorted, so it is more efficient to specify an ORDER BY clause in a query that references the view.
- If the same data is used many times in one program or in multiple programs, it is more efficient to create a table rather than a view because the data must be accessed at each view reference. (This table can be a temporary table in the Work library.)
- Avoid creating views that are based on tables whose structure might change. A view is no longer valid when it references a nonexistent column.
- If a view resides in the same SAS library as the contributing table(s), it is best to specify a one-level name in the FROM clause.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which of the following statements is false regarding a PROC SQL view?
   a. A view cannot be used in a join.
   b. A view accesses the most current underlying data.
   c. A view follows the same naming conventions as a table.
   d. A view can be used in SAS programs in place of an actual SAS data file.

2. Which of the following statements describes an advantage of using a PROC SQL view?
   a. Views often save space, because a view is usually quite small compared with the data that it accesses.
   b. Views prevent users from continually submitting queries to omit unwanted columns or rows.
   c. Views hide complex joins or queries from users.
   d. all of the above

3. Which PROC SQL step creates a view that queries the table Sasuser.Payrollmaster?
   a. proc sql;
      insert into sasuser.newview
      select * from sasuser.payrollmaster;
   b. proc sql;
      create sasuser.newview as
      select * from sasuser.payrollmaster;
   c. proc sql;
      create view sasuser.newview as
      select * from sasuser.payrollmaster;
   d. proc sql;
      select * from sasuser.payrollmaster
      into view sasuser.newview;
4 Which of the following PROC SQL steps enables you to see a description of the view definition?

a) `proc sql;
   select * from sasuser.payrollmasterv;

b) `proc sql;
   describe view sasuser.payrollmasterv;

c) `proc sql;
   list sasuser.payrollmasterv;

d) `proc sql;
   contents view=sasuser.payrollmasterv;

5 Which PROC SQL step correctly references the view `Data.Empview`?

a) `proc sql;
   select * from data.empview;

b) `proc sql;
   select * from view data.empview;

c) `proc sql;
   select view * from data.empview;

d) `proc sql;
   select * from data where view='empview';

6 Which of the following PROC SQL steps correctly embeds a LIBNAME statement with a view definition?

a) `proc sql;
   insert into sasuser.newview
   select * from airline.supervisors
   libname airline 'c:\mysql';

b) `proc sql;
   create view sasuser.newview as
   from airline.supervisors
   embed libname airline 'c:\mysql';

c) `proc sql;
   using airline 'c:\mysql';
   insert into sasuser.newview
   select * from airline.supervisors;

d) `proc sql;
   create view sasuser.newview as
   select * from airline.supervisors
   using libname airline 'c:\mysql';`
7 PROC SQL views can access data from
   a  a SAS data file.
   b  another PROC SQL view.
   c  a relational database table.
   d  all of the above

8 When you are working with PROC SQL views, it is best to
   a  avoid using an ORDER BY clause in a view.
   b  avoid creating views that are based on tables whose structure might change.
   c  specify a one-level name in the FROM clause if the view resides in the same
      SAS data library as the contributing table(s).
   d  all of the above

9 You can update the data underlying PROC SQL view using the INSERT, DELETE, and UPDATE statements under which of the following conditions:
   a  The view is joined or linked to another table.
   b  The view contains a subquery.
   c  The view contains a WHERE clause.
   d  all of the above

10 Which of the following programs drops (deletes) a view?
    a  proc sql;
       delete sasuser.newview;
    b  proc sql;
       drop view sasuser.newview;
    c  proc sql;
       erase view sasuser.newview;
    d  proc sql;
       remove newview from sasuser;
CHAPTER 8
Managing Processing Using PROC SQL

Overview 262
  Introduction 262
  Objectives 262
  Prerequisites 263
Specifying SQL Options 263
Controlling Execution 264
  Restricting Row Processing 264
  Example 264
Controlling Output 265
  Including a Column of Row Numbers 265
  Example 266
  Double-Spacing Output 266
  Example 266
  Flowing Characters within a Column 267
  Example 268
Testing and Evaluating Performance 269
  Writing Timing Information for Each Statement 269
  Example 269
Resetting Options 271
  Example 271
Using Dictionary Tables 273
  Exploring and Using Dictionary Tables 274
  Example 275
  Example 276
Additional Features 277
  Restricting the Number of Loops 277
  Stopping Execution in PROC SQL after an Error 278
Summary 279
  Text Summary 279
  Specifying SQL Options 279
  Restricting Row Processing 279
  Controlling Output 279
  Testing and Evaluating Performance 279
  Resetting Options 279
  Using Dictionary Tables 280
  Additional Features 280
Syntax 280
Sample Programs 280
  Querying a Table Using PROC SQL Options 280
  Describing and Querying a Dictionary Table 281
Points to Remember 281
Introduction

The SQL procedure offers a variety of options that control processing. Some options control execution. For example, you can limit the number of rows read or written during a query. Other options control output. For example, you can control the appearance of long character columns, double-space output, or (as shown below) number your rows. Options are also available for testing and evaluating performance.

<table>
<thead>
<tr>
<th>Row</th>
<th>FlightNumber</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>182</td>
<td>YYZ</td>
</tr>
<tr>
<td>2</td>
<td>219</td>
<td>LHR</td>
</tr>
<tr>
<td>3</td>
<td>387</td>
<td>CPH</td>
</tr>
<tr>
<td>4</td>
<td>622</td>
<td>FRA</td>
</tr>
<tr>
<td>5</td>
<td>821</td>
<td>LHR</td>
</tr>
<tr>
<td>6</td>
<td>132</td>
<td>YYZ</td>
</tr>
<tr>
<td>7</td>
<td>271</td>
<td>CDG</td>
</tr>
<tr>
<td>8</td>
<td>182</td>
<td>YYZ</td>
</tr>
<tr>
<td>9</td>
<td>219</td>
<td>LHR</td>
</tr>
<tr>
<td>10</td>
<td>387</td>
<td>CPH</td>
</tr>
</tbody>
</table>

Metadata is a description or definition of data or information. SAS session metadata is stored in Dictionary tables, which are special, read-only SAS tables that contain information about SAS data libraries, SAS data sets, SAS macros, and external files that are available in the current SAS session. Dictionary tables also contain the settings for SAS system options and SAS titles and footnotes that are currently in effect. You can use the SQL procedure to access the metadata stored in Dictionary tables. For example, you can query a Dictionary table to find out which tables in a SAS library contain a specified column.

Objectives

In this chapter, you learn to

- use PROC SQL options to control execution
- use PROC SQL options to control output
- use PROC SQL to evaluate performance
- reset PROC SQL options without re-invoking the procedure
- use Dictionary tables and views to obtain information about SAS files.
Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3

Specifying SQL Options

Remember that PROC SQL options are specified in the PROC SQL statement.

General form, PROC SQL statement:

```sql
PROC SQL <option(s)>;
```

where

- `option(s)` names the option(s) to be used.

**CAUTION:**

After you specify an option, it remains in effect until you change it or you re-invoke PROC SQL.

The following tables list the options for controlling processing that are covered in this chapter. A complete description and an example of each option appears in the following sections.

**Table 8.1 Options to Control Execution**

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this option...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restrict the number of input rows</td>
<td>INOBS=</td>
</tr>
<tr>
<td>Restrict the number of output rows</td>
<td>OUTOBS=</td>
</tr>
</tbody>
</table>

**Table 8.2 Options to Control Output**

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this option...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-space the output</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>Flow characters within a column</td>
<td>FLOW</td>
</tr>
</tbody>
</table>

**Table 8.3 Options for Testing and Evaluating Performance**

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this option...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify whether PROC SQL writes timing information for each statement to the SAS log</td>
<td>STIMER</td>
</tr>
</tbody>
</table>

**Note:** For a complete list of options, see the SAS documentation for the SQL procedure.
Restricting Row Processing

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

You already know that you can use the OUTOBS= option to restrict the number of rows that PROC SQL displays or writes to a table. However, the OUTOBS= option does not restrict the rows that are read. The INOBS= option restricts the number of rows that PROC SQL takes as input from any single source. The INOBS= option is similar to the SAS system option OBS= and is useful for debugging queries on large tables.

Note: For more information about the OUTOBS= option, see Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25.

Example

In the following PROC SQL set operation, INOBS=5 is specified. As indicated in the log, only five rows from each source table, Sasuser.Mechanicslevel1 and Sasuser.Mechanicslevel2, are used. The resulting table contains 10 rows.

```sql
proc sql inobs=5;
    select *
    from sasuser.mechanicslevel1
    outer union corr
    select *
    from sasuser.mechanicslevel2;
```
Managing Processing Using PROC SQL  △  Including a Column of Row Numbers 265

Table 8.4  SAS Log

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>ME1</td>
<td>$41,677</td>
</tr>
<tr>
<td>1403</td>
<td>ME1</td>
<td>$39,301</td>
</tr>
<tr>
<td>1120</td>
<td>ME1</td>
<td>$40,067</td>
</tr>
<tr>
<td>1121</td>
<td>ME1</td>
<td>$40,757</td>
</tr>
<tr>
<td>1412</td>
<td>ME1</td>
<td>$38,919</td>
</tr>
<tr>
<td>1653</td>
<td>ME2</td>
<td>$49,151</td>
</tr>
<tr>
<td>1782</td>
<td>ME2</td>
<td>$49,483</td>
</tr>
<tr>
<td>1244</td>
<td>ME2</td>
<td>$51,695</td>
</tr>
<tr>
<td>1065</td>
<td>ME2</td>
<td>$49,126</td>
</tr>
<tr>
<td>1129</td>
<td>ME2</td>
<td>$48,901</td>
</tr>
</tbody>
</table>

Tip: You can use the PROMPT | NOPROMPT option with the INOBS= and OUTOBS= options so that you are prompted to stop or continue processing when the limits set by these options are reached.

Note: For more information about PROC SQL set operations, see Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125. △

CAUTION:
In a simple query, there might be no apparent differences between using INOBS= or OUTOBS=. Other times, it is important to choose the correct option. For example, using the average function on a column with the PROC SQL option INOBS=10 returns an average of only the 10 values read for that column. △

Controlling Output

Including a Column of Row Numbers

The NUMBER | NONUMBER option specifies whether the output from a query should include a column named ROW, which displays row numbers. NONUMBER is the default. The option is similar to the OBS | NOOBS option in the PRINT procedure.
Example

The following PROC SQL step specifies the NUMBER option. Output from the step includes a column named Row, which contains row numbers.

```
proc sql inobs=10 number;
  select flightnumber, destination
    from sasuser.internationalflights;
```

---

Double-Spacing Output

In some cases, double-spacing your output can make it easier to read. The `DOUBLE | NODOUBLE` option specifies whether PROC SQL output is double-spaced. The default is NODOUBLE.

Note: The DOUBLE | NODOUBLE option does not affect the appearance of the HTML output. To see the effect of this option, you must have text output selected in SAS Enterprise Guide.

Example

The following PROC SQL step specifies the DOUBLE option. The listing output from this step is double spaced. The HTML output from this step remains single-spaced.

```
proc sql inobs=10 double;
  select flightnumber, destination
    from sasuser.internationalflights;
```
Flowing Characters within a Column

The FLOW | NOFLOW | FLOW=n | FLOW=n m option controls the appearance of wide character columns in listing output. The FLOW option causes text to be flowed in its column instead of wrapping the entire row. n sets the width of the flowed column. Specifying n and m floats the width of the column between limits to achieve a balanced layout.
Note: The FLOW | NOFLOW | FLOW=n | FLOW=m option does not affect the appearance of HTML, PDF, or RTF output. To see the effect of this option, you must have text output selected in SAS Enterprise Guide.

Example

The following PROC SQL step does not specify the FLOW option. Notice that in the output the name and values for the column ZipCode appear under the name and values for the column FFID due to the wide character columns.

```sql
proc sql inobs=5;
  select ffid, membertype, name, address, city, state, zipcode
  from sasuser.frequentflyers
  order by pointsused;

Figure 8.3 Output from PROC SQL Step without FLOW Option
```

<table>
<thead>
<tr>
<th>FFID</th>
<th>MemberType</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD7152</td>
<td>BRONZE</td>
<td>COOPER, LESLIE</td>
<td>66 DRIVING WAY</td>
<td>Little Rock</td>
<td>AR</td>
</tr>
<tr>
<td>WD6472</td>
<td>BRONZE</td>
<td>LONG, RUSSELL</td>
<td>9813 SUMMER SQUARE</td>
<td>Monticello</td>
<td>AR</td>
</tr>
<tr>
<td>WD1576</td>
<td>GOLD</td>
<td>BRYANT, ALTON</td>
<td>726 THISTLE DR.</td>
<td>Bauxita</td>
<td>AR</td>
</tr>
<tr>
<td>WD947</td>
<td>SILVER</td>
<td>MOPRIS, DIANE</td>
<td>77 PARKWAY PLAZA</td>
<td>North Little Rock</td>
<td>AR</td>
</tr>
<tr>
<td>WD9347</td>
<td>SILVER</td>
<td>PEARSON, BRYAN</td>
<td>9999 MARKUP MANOR</td>
<td>Bella Vista</td>
<td>AR</td>
</tr>
</tbody>
</table>

Specifying `flow=10 15` causes the text within each character column to float between 10 and 15 spaces, which prevents the ZipCode column from wrapping underneath the FFID column.

```sql
proc sql inobs=5 flow=10 15;
  select ffid, membertype, name, address, city, state, zipcode
  from sasuser.frequentflyers
  order by pointsused;

Figure 8.4 Output from PROC SQL Step with FLOW Option
```

<table>
<thead>
<tr>
<th>FFID</th>
<th>MemberType</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD7152</td>
<td>BRONZE</td>
<td>COOPER, LESLIE</td>
<td>66 DRIVING WAY</td>
<td>Little Rock</td>
<td>AR</td>
</tr>
<tr>
<td>WD6472</td>
<td>BRONZE</td>
<td>LONG, RUSSELL</td>
<td>9813 SUMMER SQUARE</td>
<td>Monticello</td>
<td>AR</td>
</tr>
<tr>
<td>WD1576</td>
<td>GOLD</td>
<td>BRYANT, ALTON</td>
<td>726 THISTLE DRIVE</td>
<td>Bauxita</td>
<td>AR</td>
</tr>
<tr>
<td>WD947</td>
<td>SILVER</td>
<td>MOPRIS, DIANE</td>
<td>77 PARKWAY PLAZA</td>
<td>North Little Rock</td>
<td>AR</td>
</tr>
<tr>
<td>WD9347</td>
<td>SILVER</td>
<td>PEARSON, BRYAN</td>
<td>9999 MARKUP MANOR</td>
<td>Bella Vista</td>
<td>AR</td>
</tr>
</tbody>
</table>
Testing and Evaluating Performance

Writing Timing Information for Each Statement

The PROC SQL option `STIMER | NOSTIMER` specifies whether PROC SQL writes timing information for each statement to the SAS log, instead of writing a cumulative value for the entire procedure. NOSTIMER is the default.

In order to use the STIMER option in PROC SQL, the SAS system option `STIMER` (the default) must also be in effect. Some host operating environments require that you specify the SAS system option STIMER when you invoke SAS. The STIMER system option controls the printing of performance statistics in the SAS log. If you use the system option alone, the results will contain timing information for the entire procedure, not on a statement-by-statement basis.

You can use the OPTIONS procedure to list the current settings of SAS system options. To find out if the SAS system STIMER option is enabled on your operating environment, submit the following program:

```
proc options option=stimer value;
run;
```

Table 8.5 SAS Log

<table>
<thead>
<tr>
<th>Option Value Information For SAS Option STIMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option Value: STIMER</td>
</tr>
<tr>
<td>Option Scope: SAS Session</td>
</tr>
<tr>
<td>How option value set: Shipped Default</td>
</tr>
</tbody>
</table>

Note: PROC OPTIONS produces additional information that is specific to the operating environment under which you are running SAS. For more information about this and for descriptions of host-specific options, see the SAS documentation for your operating environment.

Example

Both of the queries in the following PROC SQL step list the name, address, city, state, and zip code of customers listed in the `Sasuser.FrequentFlyers` table. However, the second query only lists this information for customers who have earned more than 7000 points and used less than 3000 points.

When the PROC SQL statement is submitted without the STIMER option, timing information for both queries is written to the SAS log as a cumulative value for the entire procedure.

```
proc sql;
  select name, address, city, state, zipcode
    from sasuser.frequentflyers;
  select name, address, city, state, zipcode
    from sasuser.frequentflyers
    where pointsearned gt 7000 and pointsused lt 3000;
quit;
```

Note: Timing information for a PROC SQL step is not written to the SAS log until a QUIT statement is submitted or another PROC or DATA step is started.
Table 8.6 SAS Log

28 proc sql;
29 select name, address, city, state, zipcode
     from sasuser.frequentflyers;
30 select name, address, city, state, zipcode
     from sasuser.frequentflyers
     where pointsearned gt 7000 and pointsused lt 3000;
31 quit;

NOTE: PROCEDURE SQL used (Total process time):
     real time 0.34 seconds
     cpu time 0.30 seconds

When the PROC SQL statement is submitted with the STIMER option, timing information is written to the SAS log for each SELECT statement.

proc sql stimer;
   select name, address, city, state, zipcode
       from sasuser.frequentflyers;
   select name, address, city, state, zipcode
       from sasuser.frequentflyers
           where pointsearned gt 7000 and pointsused lt 3000;
   quit;

Table 8.7 SAS Log

33 proc sql stimer;
NOTE: SQL Statement used (Total process time):
     real time 0.00 seconds
     cpu time 0.00 seconds
34 select name, address, city, state, zipcode
     from sasuser.frequentflyers;
NOTE: SQL Statement used (Total process time):
     real time 0.22 seconds
     cpu time 0.17 seconds
35 select name, address, city, state, zipcode
     from sasuser.frequentflyers
36     where pointsearned gt 7000 and pointsused lt 3000;
NOTE: SQL Statement used (Total process time):
     real time 0.08 seconds
     cpu time 0.25 seconds
37 quit;
NOTE: PROCEDURE SQL used (Total process time):
     real time 0.03 seconds
     cpu time 0.29 seconds

Note: When the STIMER option is used in PROC SQL, the exact wording of the Notes that are written to the SAS log may vary for different versions of SAS.

Note: The STIMER option in PROC SQL is useful when an operation can be accomplished in more than one way and you are benchmarking each technique. Although factors such as code readability and maintenance come into consideration, you might also want to know which PROC SQL step runs the fastest.
Resetting Options

After you specify an option, it remains in effect until you change it, or you re-invoke PROC SQL. You can use the RESET statement to add, drop, or change PROC SQL options without re-invoking the SQL procedure.

General form, RESET statement:

\[
\text{RESET } \text{<option(s)>};
\]

where

\[
\text{option(s)}
\]

lists the options in any order.

Options are additive. For example, you can specify the NOPRINT option in a PROC SQL statement, submit a query, and submit the RESET statement with the NUMBER option, without affecting the NOPRINT option.

Example

Suppose you want to submit two PROC SQL queries in a single PROC SQL step. You want

- both queries to display only the first five rows of output
- the second query to display row numbers in the output.

In the following PROC SQL step, the PROC SQL statement specifies the OUTOBS= option to restrict the number of rows that will be displayed in the output. After the first SELECT statement, the RESET statement adds the NUMBER option to display row numbers in the result set.

```sql
proc sql outobs=5;
   select flightnumber, destination
   from sasuser.internationalflights;
reset number;
   select flightnumber, destination
   from sasuser.internationalflights
   where boarded gt 200;
```

The output, which contains two result sets, is shown below. The result set from the first SELECT statement reflects only by the OUTOBS= option. The result set from the second SELECT statement reflects both the OUTOBS= option and the NUMBER option that is specified in the RESET statement.
Now suppose you want to modify the PROC SQL step so that the result set from only the first SELECT statement is restricted to five rows of output. In the modified PROC SQL step, the OUTOBS= option is added to the RESET statement to change (reset) the OUTOBS= option that is specified in the PROC SQL statement. The modified step follows:

```sql
proc sql outobs=5;
    select flightnumber, destination
    from sasuser.internationalflights;
reset outobs= number;
    select flightnumber, destination
    from sasuser.internationalflights
    where boarded gt 200;
```

In the output, the result set from the second SELECT statement now contains all the rows that are generated by the query.
Dictionary tables are commonly used to monitor and manage SAS sessions because the data is easier to manipulate than the output from procedures such as PROC DATASETS.

Dictionary tables are special, read-only SAS tables that contain information about SAS data libraries, SAS macros, and external files that are in use or available in the current SAS session. Dictionary tables also contain the settings for SAS system options and SAS titles and footnotes that are currently in effect. 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.

Dictionary tables are

- created each time they are referenced in a SAS program
- updated automatically
Accessing a Dictionary table causes SAS to determine the current state of the SAS session and return the information that you want. Dictionary tables can be accessed by running a PROC SQL query against the table, using the Dictionary libref. Though SAS librefs are usually limited to eight characters, Dictionary is an automatically assigned, reserved word. You can also access a Dictionary table by referring to the PROC SQL view of the table that is stored in the Sashelp library.

The following table describes some of the Dictionary tables that are available and lists the corresponding Sashelp views. For a complete list of Dictionary tables, see the SAS documentation for the SQL procedure.

<table>
<thead>
<tr>
<th>Dictionary table</th>
<th>Sashelp view</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalogs</td>
<td>Vcatalg</td>
<td>information about catalog entries</td>
</tr>
<tr>
<td>Columns</td>
<td>Vcolumn</td>
<td>detailed information about variables and their attributes</td>
</tr>
<tr>
<td>Extfiles</td>
<td>Vextfl</td>
<td>currently assigned filerefs</td>
</tr>
<tr>
<td>Indexes</td>
<td>Vindex</td>
<td>information about indexes defined for data files</td>
</tr>
<tr>
<td>Macros</td>
<td>Vmacro</td>
<td>information about both user and system defined macro variables</td>
</tr>
<tr>
<td>Members</td>
<td>Vmember Vsaccess Vscatlg Vlsub Vstable Vstabvw Vsvew</td>
<td>general information about data library members</td>
</tr>
<tr>
<td>Options</td>
<td>Voption</td>
<td>current settings of SAS system options</td>
</tr>
<tr>
<td>Tables</td>
<td>Vtable</td>
<td>detailed information about data sets</td>
</tr>
<tr>
<td>Titles</td>
<td>Vtitle</td>
<td>text assigned to titles and footnotes</td>
</tr>
<tr>
<td>Views</td>
<td>Vview</td>
<td>general information about data views</td>
</tr>
</tbody>
</table>

---

**Exploring and Using Dictionary Tables**

You can query Dictionary tables the same way that you query any other table, including subsetting with a WHERE clause, ordering the results, creating tables, and creating PROC SQL views. Because Dictionary tables are read-only objects, you cannot insert rows or columns, alter column attributes, or add integrity constraints to them.

To see how each Dictionary table is defined, submit a DESCRIBE TABLE statement. The DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the table specified in the DESCRIBE TABLE statement. After you know how a table is defined, you can use its column names in a subsetting WHERE clause in order to retrieve specific information.
Example

The *Dictionary.Tables* table contains detailed information about tables. The following DESCRIBE TABLE statement displays information about the *Dictionary.Tables* table in the log window. The information includes the names of the columns stored in the table.

```ribbon
proc sql;
   describe table dictionary.tables;
```

Table 8.8 SAS Log

```pascal
create table DICTIONARY.TABLES
   (   libname char(8) label='Library Name',
       memname char(32) label='Member Name',
       memtype char(8) label='Member Type',
       memlabel char(256) label='Dataset Label',
       typemem char(8) label='Dataset Type',
       crdate num format=DATETIME informat=DATETIME label='Date Created',
       ...
   );
```

To display information about the files in a specific library, specify the column names in a SELECT statement and the Dictionary table name in the FROM clause. For example, the following PROC SQL step displays the columns

- Memname (name)
- Nobs (number of observations)
- Nvar (number of variables)
- Crdate (creation date) of the tables in the `Sasuser` library.

The Dictionary column names are specified in the SELECT statement and the Dictionary table name, *Dictionary.Tables*, is specified in the FROM clause. The library name, `Sasuser`, is specified in the WHERE clause.

**CAUTION:**

Note that you must specify the library name in the WHERE clause in uppercase letters (because that is how it is stored within SAS) and enclose it in quotation marks.

```ribbon
proc sql;
   select memname format=$20., nobs, nvar, crdate
       from dictionary.tables
       where libname='SASUSER';
```

Partial output is shown below.
Note: Your output might differ from that shown above, depending on the contents of your Sasuser library.

You can also use Dictionary tables to determine more specific information such as which tables in a SAS library contain a specific column.

Example

The Dictionary.Columns table contains detailed information about variables and their attributes. As in Dictionary.Tables, the Dictionary.Columns table contains a column that is titled Memname, which lists the name of each table within a library.

```sql
proc sql;
    describe table dictionary.columns;
```

Table 8.9  SAS Log

```sql
create table DICTIONARY.COLUMNS
(    libname char(8) label='Library Name',
    memname char(32) label='Member Name',
    memtype char(8) label='Member Type',
    name char(32) label='Column Name',
    type char(4) label='Column Type',
    length num label='Column Length',
    ...);
```

The following PROC SQL step lists all the tables in the Sasuser library that contain a column named EmpID. The dictionary column name, Memname, is specified in the SELECT statement. The Dictionary table, Dictionary.Columns, is specified in the FROM clause. The library name, Sasuser, and the column name, EmpID, are specified in the WHERE clause.

```sql
proc sql;
    select memname
    from dictionary.columns
    where libname='SASUSER'
```
and name='EmpID';

Partial output is shown below.

<table>
<thead>
<tr>
<th>memname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALEMEMPS</td>
</tr>
<tr>
<td>CONTRIB</td>
</tr>
<tr>
<td>ECONTRIB</td>
</tr>
<tr>
<td>EMPDATA</td>
</tr>
<tr>
<td>EMPDATU</td>
</tr>
<tr>
<td>EMPDATU2</td>
</tr>
<tr>
<td>FLIGHTATTENDANTS</td>
</tr>
<tr>
<td>FLIGHTSCHEDULE</td>
</tr>
<tr>
<td>MECHANICSLEVEL1</td>
</tr>
<tr>
<td>MECHANICSLEVEL2</td>
</tr>
</tbody>
</table>

Remember that you can also access a Dictionary table by referring to the PROC SQL view of the table that is stored in the Sashelp library. In the following PROC SQL step, the Sashelp view Vcolumn is specified in the FROM clause. The results of the query are identical to the preceding output.

```sql
proc sql;
select memname
  from sashelp.vcolumn
  where libname='SASUSER'
    and name='EmpID';
```

**CAUTION:**

Note that column names in the WHERE clause must be specified in the same case that is used in the Dictionary table and must be enclosed in quotation marks.

**Note:** You can use Sashelp views in any SAS procedure or DATA step. However, Dictionary tables can only be read by using the SQL procedure.

### Additional Features

#### Restricting the Number of Loops

The LOOPS= option restricts the number of iterations of the inner loop in PROC SQL. By setting a limit, you can prevent queries from consuming excessive 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 PROMPT | NOPROMPT option to modify the effect of the LOOPS= option so that you are prompted to stop or continue processing when the limit set by the LOOPS= option is reached.
Note: 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 about the SQLOOPS macro variable, see the SAS documentation for the SQL procedure.

### Stopping Execution in PROC SQL after an Error

You already know that you can use the EXEC | NOEXEC option to specify whether a statement should be executed after its syntax is checked for accuracy. If the EXEC option is in effect, SAS checks the PROC SQL syntax for accuracy and, if no error is found, executes the SQL statement.

The ERRORSTOP | NOERRORSTOP option specifies whether PROC SQL stops executing if it encounters an error. This option is useful only when the EXEC option is in effect. The default is ERRORSTOP in batch or in a noninteractive session and NOERRORSTOP in an interactive SAS session.

ERRORSTOP instructs PROC SQL to stop executing the statements but to continue checking the syntax after it has encountered an error. ERRORSTOP has an effect only when SAS is running in batch or in noninteractive execution mode.

NOERRORSTOP instructs PROC SQL to execute the statements and to continue checking the syntax after an error occurs. NOERRORSTOP is useful if you want a batch job to continue executing SQL procedure statements after an error is encountered.
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Specifying SQL Options

The SQL procedure offers a variety of options that affect processing. Some options control execution. For example, you can limit the number of rows read or written during a query or limit the number of internal loops PROC SQL performs. Other options control output. For example, you can flow character columns, number your rows, or double-space output. Options are also available for testing and evaluating performance. Options are specified in the PROC SQL statement.

Restricting Row Processing

The OUTOBS= option restricts the number of rows that PROC SQL displays or writes to a table. The INOBS= option restricts the number of rows that PROC SQL takes as input from any single source. The INOBS= option is similar to the SAS system option OBS= and is useful for debugging queries on large tables.

Controlling Output

The NUMBER | NONUMBER option specifies whether the SELECT statement should include a column named ROW, which is the row number of the data as it is retrieved. NONUMBER is the default. The option is similar to the OBS | NOOBS option in the PRINT procedure.

In some cases, double-spacing your output can make it easier to read. The DOUBLE | NODOUBLE option specifies whether PROC SQL output is double-spaced in the listing output. The default is NODOUBLE.

The FLOW | NOFLOW | FLOW=n | FLOW=n m option controls the appearance of wide character columns in the listing output. The FLOW option causes text to be flowed in its column instead of wrapping the entire row. Specifying n sets the width of the flowed column. Specifying n and m floats the width of the column between limits to achieve a balanced layout.

Testing and Evaluating Performance

The STIMER | NOSTIMER option specifies whether PROC SQL writes timing information for each statement to the SAS log, in addition to writing a cumulative value for the entire procedure. NOSTIMER is the default. In order to use the STIMER option in PROC SQL, the SAS system option STIMER (the default) must also be in effect.

Resetting Options

After you specify an option, it remains in effect until you change it or you re-invoke PROC SQL. You can use the RESET statement to add, drop, or change PROC SQL options without re-invoking the SQL procedure.
Using Dictionary Tables

SAS session metadata is stored in Dictionary tables, which are special, read-only SAS tables that contain information about SAS data libraries, SAS macros, and external files that are available in the current SAS session. A Dictionary table also contains the settings for SAS system options and SAS titles and footnotes that are currently in effect.

Accessing a Dictionary table causes PROC SQL to determine the current state of the SAS session and return the information that you want. Dictionary tables can be accessed by running a PROC SQL query against the table, using the Dictionary libref. You can also access a Dictionary table by referring to the PROC SQL view of the table that is stored in the Sashelp library.

To see how each Dictionary table is defined, submit a DESCRIBE TABLE statement. After you know how a table is defined, you can use its column names in a subsetting WHERE clause in order to retrieve specific information. To display information about the files in a specific library, specify the column names in a SELECT statement and the dictionary table name in the FROM clause. You can also use Dictionary tables to determine more specific information such as which tables in a SAS library contain a specific column.

Additional Features

The LOOPS= option restricts the number of iterations of the inner loop in PROC SQL. By setting a limit, you can prevent queries from consuming excessive resources.

The ERRORSTOP | NOERRORSTOP option specifies whether PROC SQL stops executing if it encounters an error.

Syntax

PROC SQL <option(s)>;
   DESCRIBE TABLE table-name <, ...table-name>;
   SELECT column-1<, ...column-n>
      FROM table-1 | view-1<, ...table-n | view-n>
      <WHERE expression>>;
   RESET <option(s)>;
   QUIT;

Sample Programs

Querying a Table Using PROC SQL Options

proc sql outobs=5;
   select flightnumber, destination
      from sasuser.internationalflights;
   reset number;
   select flightnumber, destination
      from sasuser.internationalflights
      where boarded gt 200;
quit;
Describing and Querying a Dictionary Table

```sql
proc sql;
    describe table dictionary.columns;
    select memname
    from dictionary.columns
    where libname='SASUSER'
    and name='EmpID';
quit;
```

Points to Remember

- After you specify an option, it remains in effect until you change it or you re-invoke PROC SQL.
- The DOUBLE | NODOUBLE and the FLOW | NOFLOW | FLOW=n | FLOW=n\_m options do not affect the appearance of HTML, PDF, or RTF output that is created with the Output Delivery System.
- If you query a Dictionary table about the files in a specific library, the library name used in the WHERE clause must be specified in uppercase letters because that is how it is stored in SAS. Column names used in the WHERE clause must be specified in the same case as they appear in the Dictionary table.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. PROC SQL options are specified in
   a. the PROC SQL statement.
   b. an OPTIONS statement.
   c. a SELECT statement.
   d. the OPTIONS procedure.

2. Which of the following options restricts the number of rows that PROC SQL takes as input from any single source?
   a. OUTOBS=  
   b. INOBS=  
   c. OBS=  
   d. none of the above
3 Which PROC SQL step creates the output shown below?

<table>
<thead>
<tr>
<th>Row</th>
<th>EmpID</th>
<th>JobCode</th>
<th>LastName</th>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1574</td>
<td>FA2</td>
<td>CAHILL</td>
<td>MARSHALL</td>
</tr>
<tr>
<td>2</td>
<td>1125</td>
<td>FA2</td>
<td>DUNAP</td>
<td>DONNA</td>
</tr>
<tr>
<td>3</td>
<td>1475</td>
<td>FA2</td>
<td>EATON</td>
<td>ALICIA</td>
</tr>
<tr>
<td>4</td>
<td>1388</td>
<td>FA2</td>
<td>JEPSEN</td>
<td>RONALD</td>
</tr>
<tr>
<td>5</td>
<td>1411</td>
<td>FA2</td>
<td>JOHNSON</td>
<td>JACKSON</td>
</tr>
<tr>
<td>6</td>
<td>1441</td>
<td>FA2</td>
<td>LAWRENCE</td>
<td>KATHY</td>
</tr>
<tr>
<td>7</td>
<td>1477</td>
<td>FA2</td>
<td>MEYERS</td>
<td>PRESTON</td>
</tr>
<tr>
<td>8</td>
<td>1424</td>
<td>FA2</td>
<td>PATTERSON</td>
<td>RENEE</td>
</tr>
<tr>
<td>9</td>
<td>1413</td>
<td>FA2</td>
<td>PETERS</td>
<td>RANDALL</td>
</tr>
<tr>
<td>10</td>
<td>1555</td>
<td>FA2</td>
<td>RODRIGUEZ</td>
<td>JULIA</td>
</tr>
</tbody>
</table>

a proc sql nonumber outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

b proc sql number;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   reset nonumber outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

c proc sql nonumber;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   reset number outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

d proc sql;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   reset outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA2';
4 Which of the following options does not affect the appearance of HTML, PDF, or RTF output?
   a NUMBER | NONUMBER
   b DOUBLE | NODOUBLE
   c FLOW | NOFLOW | FLOW=n | FLOW=n m
   d b and c

5 Which of the following statements is true regarding the STIMER option in PROC SQL?
   a The STIMER option in PROC SQL writes timing information for each statement to the SAS log.
   b The STIMER option in PROC SQL writes only cumulative timing information for the entire procedure to the SAS log.
   c When using the STIMER option in PROC SQL, the SAS system option STIMER must also be in effect.
   d a and c

6 A Dictionary table contains
   a information about SAS data libraries.
   b information about SAS data sets.
   c information about SAS macros.
   d all of the above

7 Dictionary tables are
   a created each time they are referenced in a SAS program.
   b updated automatically.
   c limited to read-only access.
   d all of the above

8 Dictionary tables can be accessed
   a by running a PROC SQL query against the table, using the Dictionary libref.
   b by referring to the PROC SQL view of the table that is stored in the Sashelp library.
   c by referring to the PROC SQL view of the table that is stored in the Sasuser library.
   d a and b

9 Which of the following PROC SQL steps displays information about the Dictionary table Dictionary.Titles?
   a proc sql;
      describe dictionary.titles;
   b proc sql;
      describe table dictionary.titles;
   c proc sql describe table dictionary.titles;
   d proc sql describe dictionary titles;
10 Which of the following PROC SQL steps displays the name (**Memname**), modification date (**Modate**), number of variables (**Nvar**), and the number of observations (**Nobs**) for each table in the **Sasuser** library?

a proc sql;
    select memname, modate, nvar, nobs
    from dictionary.tables
    where libname='SASUSER';

b proc sql;
    select memname, modate, nvar, nobs
    from dictionary.tables
    where libname='Sasuser';

c proc sql;
    select memname, modate, nvar, nobs
    from 'SASUSER'
    where table=dictionary.tables;

d proc sql;
    select SASUSER
    from dictionary.tables
    where cols= 'memname, modate, nvar, nobs';
SAS Macro Language

Chapter 9 .......... Introducing Macro Variables  287
Chapter 10 ......... Processing Macro Variables at Execution Time  325
Chapter 11 .......... Creating and Using Macro Programs  371
Chapter 12 .......... Storing Macro Programs  423
CHAPTER 9

Introducing Macro Variables

Overview 288
Introduction 288
Objectives 289
Basic Concepts 289
Macro Variables 290
Referencing Macro Variables 290
Example: Referencing a Macro Variable 291
Example: Referencing a Macro Variable in a Title 291
Using Automatic Macro Variables 292
Example 293
Using User-Defined Macro Variables 293
The %LET Statement 293
Example 294
%LET Statement Examples 294
Processing Macro Variables 295
SAS Processing 295
Tokenization 296
Examples 297
Macro Triggers 298
Displaying Macro Variable Values in the SAS Log 298
The SYMBOLGEN Option 298
Example 299
The %PUT Statement 299
Example 300
Using Macro Functions to Mask Special Characters 301
Macro Quoting Functions 301
Example 302
The %STR Function 302
Example 303
The %NRSTR Function 304
Example 304
The %BQUOTE Function 305
Example 305
Using Macro Functions to Manipulate Character Strings 306
Macro Character Functions 306
The %UPCASE Function 306
Example 306
The %QUPCASE Function 307
Example 308
The %SUBSTR Function 308
Example 309
Introduction

SAS macro variables enable you to substitute text in your SAS programs. Macro variables can supply a variety of information, including

- operating system information
- SAS session information
- text strings.

When you reference a macro variable in a SAS program, SAS replaces the reference with the text value that has been assigned to that macro variable. By substituting text into programs, SAS macro variables make your programs more reusable and dynamic.

The following sample code shows how a macro variable might be used to substitute a year value throughout a program, enabling you to quickly and easily change the value of year throughout the program:
Objectives

In this chapter, you learn to

- recognize some benefits of using macro variables
- substitute the value of a macro variable anywhere in a program
- identify and display automatic macro variables
- create and display user-defined macro variables
- recognize how macro variables are processed
- display macro variable values and other text in the SAS log
- use macro character functions
- combine macro references with adjacent text or with other macro variable references
- use macro quoting functions.

Basic Concepts

In the SAS programs that you write, you might find that you need to reference the same variable, data set, or text string multiple times.

title "Total Sales for 2002";
data perm.sales2002;
   set perm.sales;
   if year(enddate)=2002;
run;
proc print data=perm.sales2002;
run;

Then, you might need to change the references in your program in order to reference a different variable, data set, or text string. Especially if your programs are lengthy, scanning for specific references and updating them manually can take a lot of time, and it is easy to overlook a reference that needs to be updated.

title "Total Sales for 2001";
data perm.sales2001;
   set perm.sales;
   if year(enddate)=2002;
run;
proc print data=perm.sales2001;
run;
If you use a macro variable in your program, these updates are quick and easy because you only need to make the change in one place.

```sas
%let year=2002;
title "Total Sales for &year";
data perm.sales&year;
   set perm.sales;
   if year(enddate)=&year;
run;
proc print data=perm.sales&year;
run;
```

The value of the macro variable is inserted into your program, so you can make one change and have the change appear throughout the program. Let's take a closer look at how macro variables work.

---

**Macro Variables**

Macro variables are part of the SAS macro facility, which is a tool for extending and customizing SAS and for reducing the amount of program code you must enter in order to perform common tasks. The macro facility has its own language, which enables you to package small or large amounts of text into units that have names. From that point on, you can work with the names rather than with the text itself.

There are two types of macro variables:

- **automatic macro variables**, which are provided by SAS
- **user-defined macro variables**, whose values you create and define.

Whether automatic or user-defined, a macro variable is independent of a SAS data set and contains one text string value that remains constant until you change it. The value of a macro variable is substituted into your program wherever the macro variable is referenced.

The value of a macro variable is stored in a symbol table. The values of automatic macro variables are always stored in the global symbol table, meaning that these values are always available in your SAS session. The values of user-defined macro variables are often stored in the global symbol table as well.

```sas
%let city=Dallas;
%let date=05JAN2000;
%let amount=975;
```

Macro variables can be defined and referenced anywhere in a SAS program except within the data lines of a DATALINES statement. You will learn more about how to define and reference macro variables throughout this chapter.

---

**Referencing Macro Variables**

In order to substitute the value of a macro variable in your program, you must reference the macro variable. A macro variable reference is created by preceding the
macro variable name with an *ampersand* (&). The reference causes the *macro processor* to search for the named variable in the symbol table and to return the value of the variable if the variable exists. If you need to reference a macro variable within quotation marks, such as in a title, you must use double quotation marks. The macro processor will not resolve macro variable references that appear within single quotation marks.

*Note:* You will learn more about the macro processor later in this chapter.

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>CITY</td>
</tr>
<tr>
<td>DATE</td>
</tr>
<tr>
<td>AMOUNT</td>
</tr>
</tbody>
</table>

**Example: Referencing a Macro Variable**

To reference the macro variable `amount` from the global symbol table that is represented above, you place `&amount` in your program, as follows:

```sas
data new;
  set perm.mast;
  where fee>&amount;
run;
proc print;
run;
```

*Note:* You will see representations of code after substitution throughout this chapter. In a SAS session, you will not see this code. These representations are meant to show you what happens to your code behind the scenes, after macro processing.

**Example: Referencing a Macro Variable in a Title**

To reference the macro variable `city` in a title, you must use double quotation marks to enclose the title text in the TITLE statement, as follows:

```sas
title "Students from &city";
```

When the macro processor cannot resolve a macro variable reference, a message is printed in the *SAS log*. For example, referencing a nonexistent macro variable results in a warning message. Referencing an invalid macro variable name results in an error message.
Using Automatic Macro Variables

SAS creates and defines several automatic macro variables for you. Automatic macro variables contain information about your computing environment, such as the date and time of the session, and the version of SAS you are running. These automatic macro variables

- are created when SAS is invoked
- are global (always available)
- are usually assigned values by SAS
- can be assigned values by the user in some cases.

Some automatic macro variables have fixed values that are set when SAS is invoked.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSDATE</td>
<td>the date of the SAS invocation (DATE7.)</td>
</tr>
<tr>
<td>SYSDATE9</td>
<td>the date of the SAS invocation (DATE9.)</td>
</tr>
<tr>
<td>SYSDAY</td>
<td>the day of the week of the SAS invocation</td>
</tr>
<tr>
<td>SYSTIME</td>
<td>the time of the SAS invocation</td>
</tr>
<tr>
<td>SYSENV</td>
<td>FORE (interactive execution) or BACK (noninteractive or batch execution)</td>
</tr>
<tr>
<td>SYSSCP</td>
<td>an abbreviation for the operating system that is being used, such as OpenVMS, WIN, HP 300</td>
</tr>
<tr>
<td>SYSVER</td>
<td>the release of SAS that is being used</td>
</tr>
<tr>
<td>SYSJOBID</td>
<td>an identifier for the current SAS session or for the current batch job (the user ID or job name for mainframe systems, the process ID (PID) for other systems)</td>
</tr>
</tbody>
</table>

Some automatic macro variables have values that automatically change based on submitted SAS statements.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSLAST</td>
<td>the name of the most recently created SAS data set, in the form LIBREF.NAME. This value is always stored in all capital letters. If no data set has been created, the value is <em>NULL</em>.</td>
</tr>
<tr>
<td>SYSPARM</td>
<td>text that is specified when SAS is invoked</td>
</tr>
<tr>
<td>SYSERR</td>
<td>contains a return code status that is set by the DATA step and some SAS procedures to indicate if the step or procedure executed successfully</td>
</tr>
</tbody>
</table>
### Introducing Macro Variables

The %LET Statement

### Example

You can substitute system information such as the time, day, and date your SAS session was invoked and the version of SAS you are running in footnotes for a report.

```sas
footnote1 "Created &systime &sysday, &sysdate9";
footnote2 "on the &sysscp system using Release &sysver";
title "REVENUES FOR DALLAS TRAINING CENTER";
proc tabulate data=sasuser.all(keep=location course_title fee);
    where upcase(location)="DALLAS";
    class course_title;
    var fee;
    table course_title=" " all="TOTALS",
                    fee=" "*(n*f=3. sum*f=dollar10.)
                   / rts=30 box="COURSE";
run;
```

#### REVENUES FOR DALLAS TRAINING CENTER

<table>
<thead>
<tr>
<th>Course</th>
<th>N</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artificial Intelligence</td>
<td>25</td>
<td>$10,000</td>
</tr>
<tr>
<td>Basic Telecommunications</td>
<td>18</td>
<td>$14,310</td>
</tr>
<tr>
<td>Computer Aided Design</td>
<td>19</td>
<td>$30,400</td>
</tr>
<tr>
<td>Database Design</td>
<td>23</td>
<td>$8,625</td>
</tr>
<tr>
<td>Local Area Networks</td>
<td>24</td>
<td>$15,600</td>
</tr>
<tr>
<td>Structured Query Language</td>
<td>24</td>
<td>$27,800</td>
</tr>
<tr>
<td>TOTALS</td>
<td>133</td>
<td>$108,535</td>
</tr>
</tbody>
</table>

*Created 11:16\(^1\) Wednesday\(^2\), 05JUN2002\(^3\) on the WIN\(^4\) system using Release 9.00\(^5\)*

1. time of day (**SYSTIME**)
2. day of the week (**SYSDAY**)
3. date (day, month, and year) (**SYSDATE9**)
4. operating environment (**SYSSCP**)
5. release of SAS (**SYSVER**)

### Using User-Defined Macro Variables

#### The %LET Statement

You’ve seen that SAS provides a variety of automatic macro variables for you. You can also create your own macro variables.

The simplest way to define your own macro variables is to use a `%LET statement`. The `%LET statement` enables you to define a macro variable and to assign a value to it.
General form, %LET statement:

\[
\text{%LET } \text{variable=value;}
\]

where

- \text{variable} is any name that follows the SAS naming convention.
- \text{value} can be any string from 0 to 65,534 characters.
- \text{variable or value} if either contains a reference to another macro variable (such as &macvar), the reference is evaluated before the assignment is made.

\[\text{Example}\]

To create a macro variable named \text{time} and assign a value of \text{afternoon} to it, you would submit the following %LET statement:

\[
\text{%let } \text{time=afternoon;}
\]

When you use the %LET statement to define macro variables, you should keep in mind the following rules:

- All values are stored as character strings.
- Mathematical expressions are not evaluated.
- The case of the value is preserved.
- Quotation marks that enclose literals are stored as part of the value.
- Leading and trailing blanks are removed from the value before the assignment is made.

\[\text{%LET Statement Examples}\]

When you define a macro variable, remember that its value is always a character string. This table provides examples of macro variable assignment statements to illustrate the rules that are listed in the previous section.

<table>
<thead>
<tr>
<th>%LET Statement</th>
<th>Variable Name</th>
<th>Variable Value</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>%let name= Ed Norton ;</td>
<td>name</td>
<td>Ed Norton</td>
<td>9</td>
</tr>
<tr>
<td>%let name2=’ Ed Norton ‘;</td>
<td>name2</td>
<td>‘Ed Norton’</td>
<td>13</td>
</tr>
<tr>
<td>%let title=&quot;Joan’s Report&quot;;</td>
<td>title</td>
<td>&quot;Joan’s Report&quot;</td>
<td>15</td>
</tr>
<tr>
<td>%let start=;</td>
<td>start</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>%let total=0;</td>
<td>total</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>%let sum=4+3;</td>
<td>sum</td>
<td>4+3</td>
<td>3</td>
</tr>
<tr>
<td>%let total=+Σ</td>
<td>total</td>
<td>0+4+3</td>
<td>5</td>
</tr>
</tbody>
</table>
In the following example, the value *DALLAS* is assigned to the macro variable *site*. The macro variable *site* is then used to control program output.

```sas
%let site=DALLAS;

title "REVENUES FOR &site TRAINING CENTER";
proc tabulate data=sasuser.all(keep=location
    course_title fee);
    where upcase(location)="&site";
    class course_title;
    var fee;
    table course_title=' ' all='TOTALS',
        fee=' '*\(n*f=3. sum*f=dollar10.)
        / rts=30 box='COURSE';
run;
```

### REVENUES FOR DALLAS TRAINING CENTER

<table>
<thead>
<tr>
<th>Course</th>
<th>N</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artificial Intelligence</td>
<td>25</td>
<td>$10,000</td>
</tr>
<tr>
<td>Basic Telecommunications</td>
<td>18</td>
<td>$14,310</td>
</tr>
<tr>
<td>Computer Aided Design</td>
<td>19</td>
<td>$30,400</td>
</tr>
<tr>
<td>Database Design</td>
<td>23</td>
<td>$8,625</td>
</tr>
<tr>
<td>Local Area Networks</td>
<td>24</td>
<td>$15,600</td>
</tr>
<tr>
<td>Structured Query Language</td>
<td>24</td>
<td>$27,600</td>
</tr>
<tr>
<td>TOTALS</td>
<td>133</td>
<td>$106,535</td>
</tr>
</tbody>
</table>

**Processing Macro Variables**

**SAS Processing**

You have seen how to create and reference macro variables. In order to work with macro variables in the programs that you write, you need to understand how macro variables are processed and stored. First, it is important that you understand how SAS processing works.

A SAS program can be any combination of

- DATA steps and PROC steps
- global statements
- SAS Component Language (SCL) code
- Structured Query Language (SQL) code
SAS macro language code.

When you submit a program, it goes to an area of memory called the input stack. This is true for all code that you submit, such as a DATA step, SCL code, or SQL code.

Once SAS code is in the input stack, SAS
- reads the text in the input stack (left-to-right, top-to-bottom)
- routes text to the appropriate compiler upon demand
- suspends this activity when a step boundary such as a RUN statement is reached
- executes the compiled code if there are no compilation errors
- repeats this process for any subsequent steps.

Let’s take a closer look.

**Tokenization**

Between the input stack and the compiler, SAS programs are tokenized into smaller pieces. A component of SAS known as the word scanner divides program text into fundamental units called tokens.
- Tokens are passed on demand to the compiler.
- The compiler requests tokens until it receives a semicolon.
- The compiler performs a syntax check on the statement.
SAS stops sending statements to the compiler when it reaches a step boundary. Examples of step boundaries include a RUN statement (run;) or the beginning of a new DATA or PROC step. Once the entire step has been compiled, it is executed.

The word scanner recognizes four types of tokens:

- A literal token is a string of characters that are treated as a unit. The string is enclosed in either single or double quotation marks.
  Examples: "Any text" ‘Any text’

- A number token is a string of numerals that can include a period or E-notation (real numbers). Date constants, time constants, datetime constants, and hexadecimal constants are also number tokens.
  Examples: 23 109 ‘01jan2002’d 5e8 42.7

- A name token is a string of characters that begins with a letter or underscore and that continues with underscores, letters, or digits. A period can sometimes be part of a name.
  Examples: infile _n_ item3 univariate dollar10.2

- A special token is any character or group of characters that has a reserved meaning to the compiler.
  Examples: * / + - ** ; $ ( ) . & %

A token ends when the word scanner detects
- the beginning of another token
- a blank after a token.

The maximum length of any token is 32767 characters.

---

**Examples**

- var x1-x10 z;
  This example contains six tokens: var x1 - x10 z;
- title ‘Report for May’;
This example contains three tokens: `title 'Report for May' ;`

### Macro Triggers

Macro variable references and %LET statements are part of the macro language. The macro facility includes a macro processor that is responsible for handling all macro language elements. Certain token sequences, known as macro triggers, alert the word scanner that the subsequent code should be sent to the macro processor.

The word scanner recognizes the following token sequences as macro triggers:
- `%` followed immediately by a name token (such as `%let`)
- `&` followed immediately by a name token (such as `&amt`).

When a macro trigger is detected, the word scanner passes it to the macro processor for evaluation. The macro processor
- examines these tokens
- requests additional tokens as necessary
- performs the action indicated.

For macro variables, the processor does one of the following:
- creates a macro variable in the symbol table and assigns a value to the variable
- changes the value of an existing macro variable in the symbol table
- looks up an existing macro variable in the symbol table and returns the variable’s value to the input stack in place of the original reference.

The word scanner then resumes processing tokens from the input stack.

**Note:** The word scanner will not recognize macro triggers that are enclosed in single quotation marks. Remember that if you need to reference a macro variable within a literal token, such as the title text in a TITLE statement, you must enclose the text string in double quotation marks or the macro variable reference will not be resolved.

### Displaying Macro Variable Values in the SAS Log

#### The SYMBOLGEN Option

When you submit a macro variable reference, the macro processor resolves the reference and passes the value directly back to the input stack. Therefore, you will not see the value that the compiler receives. In order to debug your programs, it might be useful for you to see the value that replaces your macro variable reference. You can use the `SYMBOLGEN` system option to monitor the value that is substituted for a macro variable reference.
Introducing Macro Variables

General form, OPTIONS statement with SYMBOLGEN option:

```
OPTIONS NOSYMBOLGEN | SYMBOLGEN;
```

where

- **NOSYMBOLGEN**
  specifies that log messages about macro variable references will *not* be displayed. This is the default.

- **SYMBOLGEN**
  specifies that log messages about macro variable references will be displayed.

This system option displays the results of resolving macro variable references in the SAS log. That is, when the SYMBOLGEN option is turned on, SAS writes a message to the log for each macro variable that is referenced in your program. The message states the macro variable name and the resolved value.

*Note:* Remember that since SYMBOLGEN is a system option, its setting remains in effect until you modify it or until you end your SAS session.

---

**Example**

Suppose you have previously assigned values to the macro variables `amount`, `city`, and `company`, and you submit the following code:

```
data new;
  set sasuser.all;
  where fee>&amount;
  where also city_state contains "&city";
  where also student_company contains '&company';
run;
```

Here is a sample SAS log that shows the messages that are generated by the SYMBOLGEN option for this code.

<table>
<thead>
<tr>
<th>Line</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>where fee&gt;&amp;amount;</td>
</tr>
<tr>
<td>111</td>
<td>SYMBOLGEN: Macro variable AMOUNT resolves to 975</td>
</tr>
<tr>
<td>112</td>
<td>where city_state contains &quot;&amp;city&quot;;</td>
</tr>
<tr>
<td>113</td>
<td>SYMBOLGEN: Macro variable CITY resolves to Dallas</td>
</tr>
</tbody>
</table>

Notice that no message is displayed for the final macro variable reference (`'&company'`). Because this macro variable reference is enclosed in *single quotation marks* rather than in double quotation marks, the word scanner *does not resolve it*.

---

**The %PUT Statement**

Another way of verifying the values of macro variables is to write your own messages to the SAS log. The %PUT statement writes text to the SAS log.
General form, basic %PUT statement:

```sas
%PUT text;
```

where

- `text` is any text string.

The %PUT statement
- writes only to the SAS log
- always writes to a new log line, starting in column one
- writes a blank line if text is not specified
- does not require quotation marks around text
- resolves macro triggers in text before text is written
- removes leading and trailing blanks from text unless a macro quoting function is used
- wraps lines when the length of text is greater than the current line size setting
- can be used either inside or outside a macro definition.

**Example**

Suppose you want to verify the value of the macro variable `city`. Since the %PUT statement resolves macro references in text before writing text to the SAS log, you can use it to show the stored value of `city`.

```sas
%put The value of the macro variable CITY is: &city;
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result in SAS Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ALL</em></td>
<td>Lists the values of all macro variables</td>
</tr>
<tr>
<td><em>AUTOMATIC</em></td>
<td>Lists the values of all automatic macro variables</td>
</tr>
<tr>
<td><em>USER</em></td>
<td>Lists the values of all user-defined macro variables</td>
</tr>
</tbody>
</table>
Introducing Macro Variables

Macro Quoting Functions

The SAS programming language uses matched pairs of either double or single quotation marks to distinguish character constants from names. The quotation marks are not stored as part of the token that they define. For example, in the following program, `var` is stored as a four-byte variable that has the value `text`. If `text` were not enclosed in quotation marks, it would be treated as a variable name. `var2` is stored as a seven-byte variable that has the value `example`.

```sas
data one;
  var='text';
  text='example';
  var2=text;
run;
```

Similarly, the title text in the following example is *Joan’s Report*. Although the TITLE statement contains a matched pair of double quotation marks, the title itself does not include these outer quotation marks. However, the outer quotation marks cause the unmatched single quotation mark within the text to be interpreted as an apostrophe that is part of the title text.

```sas
proc print;
  title "Joan’s Report";
run;
```
Example

Earlier you learned that macro variable values are character strings, and you saw examples of macro variables whose values included special characters. Now, suppose you want to store one or more SAS statements in a macro variable. For example, suppose you want to create a macro variable named `prog` with `data new; x=1; run;` stored as its value.

```sas
options symbolgen;
%let prog=data new; x=1; run;
&prog
proc print;
run;
```

Here is part of the SAS log that results from the above program.

Table 9.5 SAS Log

```
25 options symbolgen;
26 %let prog=data new; x=1; run;
27 &prog
28 proc print;
29 run;
```

Notice that according to the SYMBOLGEN statement in the log, the macro variable `prog` has been assigned a value of `data new`. SAS interpreted the first semicolon as the end of the macro assignment statement. In this case, we want the semicolon to be part of the macro variable value, but SAS has no way of knowing that. In this situation, you need to mask text that you want to assign to a macro variable. That is, you need to hide the normal meaning of the semicolon from the macro processor. You can use a macro quoting function to do this.

The %STR Function

The %STR function is used to mask (or quote) tokens during compilation so that the macro processor does not interpret them as macro-level syntax. That is, the %STR function hides the normal meaning of a semicolon and other special tokens and mnemonic equivalents of comparison or logical operators so that they appear as constant text. Special tokens and mnemonic equivalents include:

- `<`, `>`, `>=`, `<=`, `<>`, `,`, `=`, `blank`
- `LT`, `LE`, `GT`, `GE`, `AND`, `OR`, `NOT`, `NE`

The %STR function also
- enables macro triggers to work normally
Introducing Macro Variables

General form, %STR function:

%STR (argument)

where

argument

is any combination of text and macro triggers.

Applying this to our previous example, there are a number of ways that text can be quoted. Remember that we wanted to create a macro variable named prog that has data new; x=1; run; as its value.

Method One
You could quote all text.

%let prog=%str(data new; x=1; run;);

Method Two
You could quote only the semicolons.

%let prog=data new%str(;) x=1%str(;)run%str(;;);

Method Three
You could create an additional macro variable, assign a quoted value to it, and reference it in the assignment statement for the prog macro variable.

%let s=%str(;;);
%let prog=data new&s x=1&s run&s;

Each of these methods accomplishes the same thing: they all assign the value data=new; x=1; run; to the macro variable prog.
The %STR function can also be used to quote tokens that typically occur in pairs:

" " ()

Example

Suppose you want to assign text that contains an apostrophe (‘) to a macro variable. Without any quoting, this will lead to errors.

options symbolgen;
%let text=Joan’s Report;
proc print data=sasuser.courses;
  where days > 3;
title "&text";
run;
The word scanner interprets the apostrophe as the beginning of a literal that is defined by a pair of single quotation marks. You can use the %STR function to avoid this error. In the last section you saw several methods of using the %STR function to mask the normal meaning of a semicolon. However, none of the methods shown will correctly mask the apostrophe in our current example.

When you quote tokens that typically appear in pairs, such as quotation marks or parentheses, you must take one additional step. To perform this quoting, you precede the token that you want to quote with a percent sign (%) within the %STR function argument.

%let text=%str(Joan%'s Report);
%let text=Joan%str(‘)s Report;

The value of text is Joan's Report in both cases.

### The %NRSTR Function

Sometimes you might want to hide the normal meaning of an ampersand or a percent sign. The %NRSTR function performs the same quoting function as %STR, except it also masks macro triggers (& and %). The NR in the name %NRSTR stands for No Resolution. %NRSTR has the same syntax as %STR.

### Example

Suppose you want to create a macro variable named period and to assign a value of May&Jun to it. If you attempt to use the %STR function in the assignment statement, SAS will interpret the ampersand as a macro trigger and generate a warning message. You need to use the %NRSTR function instead.

%let Period=%str(May&Jun);
%let Period=%nrstr(May&Jun);

The following portion of a SAS log shows the results of both the %STR and the %NRSTR functions for this example.
Introducing Macro Variables

Example 305

Table 9.7 SAS Log

1 %let Period=%str(May&Jun);
WARNING: Apparent symbolic reference JUN not resolved.
2 %put Period resolves to &period:
WARNING: Apparent symbolic reference JUN not resolved.
Period resolves to: May&Jun
3
4 %let Period=%nrstr(May&Jun);
5 %put Period resolves to &period;
Period resolves to: May&Jun

The %BQUOTE Function

Like the %STR function, the %BQUOTE function is used to mask (or quote) special characters and mnemonic operators. However, while the %STR function performs during compilation, the %BQUOTE function performs during execution. That is, the %BQUOTE function masks a character string or resolved value of a text expression during execution of a macro or macro language statement so that special characters and mnemonic operators are not interpreted as anything other than plain text. Special tokens and mnemonic equivalents include

' '' ( ) ; + - * / , < > = blank
LT EQ GT AND OR NOT LE GE NE

The %BQUOTE function also
- does not require that quotation marks be marked
- enables macro triggers to work normally
- preserves leading and trailing blanks in its argument.

General form, %BQUOTE function:

%BQUOTE (argument)

where

argument

is any combination of text and macro triggers.

Example

Remember the example where you want to assign text that contains an apostrophe (') to a macro variable. You used the %STR function to mask the apostrophe.

%let text=%str(Joan%'s Report);
%let text=Joan%str(\')s Report;

You can accomplish this task using the %BQUOTE function. The %BQUOTE function does not require that unmatched quotation marks be marked, so the title that contains an apostrophe requires no special syntax.

%let text=%bquote(Joan's Report);

Note: The %NRBQUOTE function works the same as the %BQUOTE function except that %NRBQUOTE also masks the special characters listed below. △
Macro Character Functions

Often when working with macro variables, you will need to manipulate character strings. You can do this by using macro character functions. With macro character functions, you can

- change lowercase letters to uppercase
- produce a substring of a character string
- extract a word from a character string
- determine the length of a character string, and more.

Macro character functions have the same basic syntax as the corresponding DATA step functions, and they yield similar results. It is important to remember that although they might be similar, macro character functions are distinct from DATA step functions. As part of the macro language, macro functions enable you to communicate with the macro processor in order to manipulate text strings that you insert into your SAS programs. The next few sections explore several macro character functions in greater detail.

The %UPCASE Function

The %UPCASE function enables you to change the value of a macro variable from lowercase to uppercase before substituting that value in a SAS program. Since most comparison operators in the SAS language are case sensitive, it is often necessary to change values to uppercase.

General form, %UPCASE function:

```
%UPCASE (argument)
```

where

- `argument` is a character string.

Example

The `Sasuser:All` data set contains student information and registration information for computer training courses. Suppose you want to create a summary of the uncollected course fees:

```
%let paidval=n;
proc means data=sasuser.all sum maxdec=0;
  where paid="&paidval";
  var fee;
  class course_title;
```
Introducing Macro Variables

The %QUPCASE Function

Title "Uncollected Fees for Each Course"
run;

Table 9.8 SAS Log

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>163</td>
<td>%let paidval=n;</td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>proc means data=sasuser.all</td>
<td>sum maxdec=0;</td>
</tr>
<tr>
<td></td>
<td>where paid=&quot;&amp;paidval&quot;;</td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>var fee;</td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>class course_title;</td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>title &quot;Uncollected Fees for Each Course&quot;;</td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>run;</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: No observations were selected from data set SASUSER.ALL.

Because the value of the macro variable `paidval` was specified in lowercase, the WHERE expression finds no matching observations. All the values of the data set variable `Paid` are stored in uppercase.

Now let's use the %UPCASE function in the WHERE statement:

```sas
%let paidval=n;
proc means data=sasuser.all sum maxdec=0;
    where paid="%upcase(&paidval)";
    var fee;
    class course_title;
    title "Uncollected Fees for Each Course";
run;
```

You can see that this time the WHERE expression does find matching observations.

Uncollected Fees for Each Course

**The MEANS Procedure**

<table>
<thead>
<tr>
<th>Analysis Variable : Fee Course Fee</th>
<th>Description</th>
<th>N Obs</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Artificial Intelligence</td>
<td>17</td>
<td>6600</td>
</tr>
<tr>
<td>Description</td>
<td>Basic Telecommunications</td>
<td>25</td>
<td>19675</td>
</tr>
<tr>
<td>Description</td>
<td>Computer Aided Design</td>
<td>18</td>
<td>26600</td>
</tr>
<tr>
<td>Description</td>
<td>Database Design</td>
<td>14</td>
<td>5250</td>
</tr>
<tr>
<td>Description</td>
<td>Local Area Networks</td>
<td>15</td>
<td>9750</td>
</tr>
<tr>
<td>Description</td>
<td>Structured Query Language</td>
<td>16</td>
<td>18400</td>
</tr>
</tbody>
</table>

**The %QUPCASE Function**

If the argument contains a special character, a mnemonic operator, or a macro trigger, you will need to use the %QUPCASE function. %QUPCASE has the same syntax as the %UPCASE function, and it works the same as %UPCASE except that it also masks mnemonic operators and special characters (including macro triggers).
Example

These statements show the results produced by %UPCASE and %QUPCASE:

```sas
%let a=begin;
%let b=%nrstr(&a);

%put UPCASE produces: %upcase(&b);
%put QUPCASE produces: %qupcase(&b);
```

In the first %PUT statement, the macro reference &b resolves to &a, which is converted to &A because of the %UPCASE function. Since the resolved value contains a macro trigger, it is treated as a macro variable reference and &A resolves to the value `begin`. The second %PUT statement uses the %QUPCASE function, which masks the ampersand in the resolved value of the macro variable b so that this value is not treated as another macro variable reference. Executing these statements produces the following messages in the SAS log.

**Table 9.9 SAS Log**

```sas
6  %let a=begin;
7  %let b=%nrstr(&a);
8  
9  %put UPCASE produces: %upcase(&b);
10 %put QUPCASE produces: %qupcase(&b);
```

The %SUBSTR Function

The %SUBSTR function enables you to extract part of a character string from the value of a macro variable.

**General form, %SUBSTR function:**

```sas
%SUBSTR(argument, position<n>)
```

where

- **argument**
  - is a character string or a text expression from which a substring will be returned.

- **position**
  - is an integer or an expression (text, logical, or mathematical) that yields an integer, which specifies the position of the first character in the substring.

- **n**
  - is an optional integer or an expression (text, logical, or mathematical) that yields an integer that specifies the number of characters in the substring.

**Note:** If the length of n is greater than the number of characters following position in argument, %SUBSTR issues a warning message and returns a substring that contains the characters from position to the end of the string. If n is not specified, %SUBSTR also returns a substring that contains the characters from position to the end of the string. △
Introducing Macro Variables

Example 309

For example, assume that the macro variable `date` has the value `05JAN2002`.
- The code `%substr(&date, 3)` will return the value `JAN2002`.
- The code `%substr(&date, 3, 3)` will return the value `JAN`.
- The code `%substr(&date, 3, 9)` will return the value `JAN2002` and will produce a warning message.

The values of `position` and `n` can also be the result of a mathematical expression that yields an integer. For example, `%substr(&var, %length(&var) - 1)` returns the last two characters of the value of the macro variable `var`.

Note: The `%LENGTH` function accepts an argument that is either a character string or a text expression. If the argument is a character string, `%LENGTH` returns the length of the string. If the argument is a text expression, `%LENGTH` returns the length of the resolved value. If the argument has a null value, `%LENGTH` returns 0.

Example

Suppose you want to print a report on all courses that have been taught since the start of the current month. You can use the `%SUBSTR` function and the `SYSDATE9` macro variable to determine the month and year, as follows:

```sasproc print data=sasuser.schedule;
  where begin_date between "01%substr(&sysdate9, 3)"d and "&sysdate9"d;
  title "All Courses Held So Far This Month";
  title2 "(as of &sysdate9)";
run;
```

`All Courses Held So Far This Month
(as of 25MAR2002)`

<table>
<thead>
<tr>
<th>Obs</th>
<th>Course_Number</th>
<th>Course_Code</th>
<th>Location</th>
<th>Begin_Date</th>
<th>Teacher</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>18</td>
<td>C006</td>
<td>Dallas</td>
<td>25MAR2002 Berthan, Ms. Judy</td>
<td></td>
</tr>
</tbody>
</table>

The `%QSUBSTR` Function

If the argument contains a special character, a mnemonic operator, or a macro trigger, you will need to use the `%QSUBSTR` function. `%QSUBSTR` has the same syntax as the `%SUBSTR` function, and it works the same as `%SUBSTR` except that it also masks mnemonic operators and special characters (including macro triggers).

Example

These statements show the results produced by `%SUBSTR` and `%QSUBSTR`:

```sas%let a=one;
%let b=two;
%let c=%nrstr(&a &b);```
%put C: &c
%put With SUBSTR: %substr(&c,1,2);
%put With QSUBSTR: %qsubstr(&c,1,2);

Executing these statements produces the following messages in the SAS log. As you can see, the first %PUT statement shows that &c resolves to the value &a &b. In the second %PUT statement, the %SUBSTR function extracts the value &a from the resolved value of the macro variable reference &c, and resolves &a to one. The third %PUT statement shows that the %QSUBSTR function prevents the value &a from being resolved further.

Table 9.10  SAS Log

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>%let a=one;</td>
</tr>
<tr>
<td>12</td>
<td>%let b=two;</td>
</tr>
<tr>
<td>13</td>
<td>%let c=%nrstr(&amp;a &amp;b);</td>
</tr>
<tr>
<td>14</td>
<td>%put C: &amp;c;</td>
</tr>
<tr>
<td>15</td>
<td>C: &amp;a &amp;b</td>
</tr>
<tr>
<td>16</td>
<td>%put With SUBSTR: %substr(&amp;c,1,2);</td>
</tr>
<tr>
<td>With SUBSTR: one</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>%put With QSUBSTR: %qsubstr(&amp;c,1,2);</td>
</tr>
<tr>
<td>With QSUBSTR: &amp;a</td>
<td></td>
</tr>
</tbody>
</table>

The %INDEX Function

The %INDEX function enables you to determine the position of the first character of a string within another string.

General form, %INDEX function:

```
%INDEX(source,string)
```

where

- `source` and `string` both are character strings or text expressions that can include:
  - constant text
  - macro variable references
  - macro functions
  - macro calls.

The %INDEX function

- searches `source` for the first occurrence of `string`
- returns a number representing the position in `source` of the first character of `string`
  - when there is an exact pattern match
  - returns 0 when there is no pattern match.

Example

The following statements find the first character V in a string:
Introducing Macro Variables

Example 311

```sas
%let a=a very long value;
%let b=%index(&a,v);
%put V appears at position &b.;
```

Executing these statements writes the following line to the SAS log.

Table 9.11  SAS Log

V appears at position 3.

The %SCAN Function

The %SCAN function enables you to extract words from the value of a macro variable.

General form, %SCAN function:

```
%SCAN (argument, n<delimiters>)
```

where

- `argument` consists of constant text, macro variable references, macro functions, or macro calls.
- `n` is an integer or a text expression that yields an integer, which specifies the position of the word to return. If `n` is greater than the number of words in `argument`, the functions return a null string.
- `delimiters` specifies an optional list of one or more characters that separate "words" or text expressions that yield one or more characters.

**CAUTION:**

If `argument` contains a comma, you must enclose `argument` in a quoting function. Similarly, in order to use a single blank or a single comma as the only `delimiter`, you must enclose the character in the %STR function.

The delimiters that %SCAN recognizes vary between ASCII and EBCDIC systems. If you omit delimiters, SAS treats the following characters as default delimiters:

- ASCII systems: `blank . < ( + & ! $ * ) ; ^ - / , % |`
- EBCDIC systems: `blank . < ( + | & ! $ * ) ; ~ - / , % | ¦`

If `delimiters` includes any of the default delimiters for your system, the remaining default delimiters are treated as text.

**Example**

You can use PROC DATASETS along with the %SCAN function and the `SYSLAST` macro variable to investigate the structure of the most recently created data set:

```sas
data work.thisyear;
  set sasuser.schedule;
  where year(begin_date) =
      year("&sysdate9"d);
```
run;

%let libref=%scan(&syslast,1,.,);
%let dsname=%scan(&syslast,2,.,);
proc datasets lib=&libref nolist;
title "Contents of the Data Set &syslast";
  contents data=&dsname;
run;
quit;

---

The %QSCAN Function

If the argument contains a special character, a mnemonic operator, or a macro trigger, you will need to use the %QSCAN function. %QSCAN has the same syntax as the %SCAN function, and it works the same as %SCAN except that it also masks mnemonic operators and special characters (including macro triggers).

Example

These statements show the results produced by %SCAN and %QSCAN:

```sas
%let a=one;
%let b=two;
%let c=%nrstr(&a*&b);

%put C: &c;
%put With SCAN: %scan(&c,1,*);
%put With QSCAN: %qscan(&c,1,*);
```

Executing these statements produces the following messages in the SAS log.

---

### Contents of the Data Set WORK.THISYEAR

**The DATASETS Procedure**

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>WORK.THISYEAR</th>
<th>Observations</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>5</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>15:33 Thursday, June 27, 2002</td>
<td>Observation Length</td>
<td>48</td>
</tr>
<tr>
<td>Last Modified</td>
<td>15:33 Thursday, June 27, 2002</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td></td>
<td>NO</td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted</td>
<td></td>
<td>NO</td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>WINDOWS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>watin1 Western (Windows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introducing Macro Variables

The %SYSFUNC Function

You've learned that by using the automatic macro variables SYSDATE9 and SYSTIME you can include the date and time in a title:

```sas
title1 "Report Produced on &sysdate9";
title2 "at &systime";
```

Report Produced on 08JUN2000
at 12:11

SYSDATE9 represents the date on which the SAS session started, and SYSTIME represents the time at which the SAS session started. Suppose you would rather see the date in some other format, or suppose you would rather see the current date or time. You can use the %SYSFUNC function to execute other SAS functions as part of the macro facility.

General form, %SYSFUNC function:

```sas
%SYSFUNC (function (argument(s)) <,format>)
```

where

- `function` is the name of the SAS function to execute.
- `argument(s)` is one or more arguments that are used by `function`. Use commas to separate all arguments. An argument can be a macro variable reference or a text expression that produces arguments for a function.
- `format` is an optional format to apply to the result of `function`. By default, numeric results are converted to a character string using the BEST12. format, and character results are used as they are, without formatting or translation.

All SAS functions can be used with %SYSFUNC except

---

**Table 9.12  SAS Log**

```sas
47 %let a=one;
48 %let b=two;
49 %let c=%nrstr(&a*&b);
50
51 %put C: &c;
C: &a*&b
52 %put With SCAN: %scan(&c,1,*);
With SCAN: one
53 %put With QSCAN: %qscan(&c,1,*);
With QSCAN: &a
```
Example

Suppose the following code was submitted on Friday, June 7, 2007:

```
title "%sysfunc(today(),weekdate.) - SALES REPORT";
```

The title on the next report would be Friday, June 7, 2007 - SALES REPORT.

Quoting with %QSYSFUNC

As with macro character functions, if the argument for a %SYSFUNC function contains special characters or mnemonic operators, you must use the quoting version of the function. The %QSYSFUNC function has the same syntax as the %SYSFUNC function. %QSYSFUNC works the same as %SYSFUNC except that it also masks mnemonic operators and special characters.

Example

Suppose you want to create a report title that includes the current date in WORDDATE. format. You could use this statement:

```
title "Report Produced on %sysfunc(today(),worddate.)";
```

However, that would result in the following title:

Report Produced on June 7, 2007

The extra blanks are from the default length of the WORDDATE. format. You need to left justify the resulting formatted date. You cannot nest functions within %SYSFUNC, but you can use a %SYSFUNC for each function that you need, as shown in this example:

```
title "Report Produced on
       %sysfunc(left(%sysfunc(today(),worddate.))))";
```
However, this statement results in the following error message.

Table 9.13 SAS Log

| ERROR: The function LEFT referenced by the %SYSFUNC or %QSYSFUNC macro function has too many arguments. |

The LEFT function expects only one argument, but you are passing “June 7, 2007” to it. It interprets the comma as the delimiter between two arguments.
You can mask the comma by using the %QSYSFUNC function instead, as follows:

```sas
title "Report Produced on 
    %sysfunc(left(%qsysfunc(today(),worddate.)))";
```

The modified statement generates the following title:

*Report Produced on June 7, 2007*

### Combining Macro Variable References with Text

You can reference macro variables anywhere in your program. Some applications might require placing a macro variable reference adjacent to leading text (`text&variable`) or trailing text (`&variabletext`) or referencing adjacent macro variables (`&variable&variable`) in order to build a new token. When you combine macro variable references and text, it is important to keep in mind how SAS interprets tokens.

Remember that a token ends when the word scanner detects either the beginning of a new token or a blank after a token.

You can place text immediately before a macro variable reference to build a new token. For example, suppose that data sets are stored in a SAS library, using the naming convention `Yyymon`, where `yy` is a two-digit year such as `02` or `01`, and `mon` is a three-letter month such as `JUN` or `AUG`. Data set names could include examples such as `Y01DEC` and `Y02MAR`. You can write a program that uses a macro variable to build the month portion of the SAS data set name.

```sas
%let month=jan;
proc chart data=sasuser.y02&month;
   hbar week / sumvar=sale;
run;
proc plot data=sasuser.y02&month;
   plot sale*day;
run;
```

Table 9.14 Code After Substitution

| `proc chart data=sasuser.y02jan;` |
| `   hbar week / sumvar=sale;` |
| `run;` |
| `proc plot data=sasuser.y02jan;` |
| `   plot sale*day;` |
| `run;` |

**CAUTION:**

If you are using the SAS Learning Edition, you will not be able to submit this code because it uses PROC CHART and the CHART procedure is not included in the software. This example is used in the next several sections and in the chapter summary.
You can reference macro variables that have *no blanks between them* to build new tokens.

For example, you can modify the previous program to enable both the month and the year to be substituted:

```sas
%let year=02;
%let month=jan;
proc chart data=sasuser.y&year&month;
    hbar week / sumvar=sale;
run;
proc plot data=sasuser.y&year&month;
    plot sale*day;
run;
```

Table 9.15  Code After Substitution

<table>
<thead>
<tr>
<th>Code After Substitution</th>
</tr>
</thead>
</table>
| proc chart data=sasuser.y02jan;
    hbar week / sumvar=sale;
run; |
| proc plot data=sasuser.y02jan;
    plot sale*day;
run; |

The generated program is identical to the program in the previous example. That is, the compiler sees the same code for both of these examples.

You can place text *immediately after* a macro variable reference as long as the macro variable name can still be tokenized correctly.

For example, you can modify the previous program to substitute the name of an analysis variable:

```sas
%let year=02;
%let month=jan;
%let var=sale;
proc chart data=sasuser.y&year&month;
    hbar week / sumvar=&var;
run;
proc plot data=sasuser.y&year&month;
    plot &var*day;
run;
```

Table 9.16  Code After Substitution

<table>
<thead>
<tr>
<th>Code After Substitution</th>
</tr>
</thead>
</table>
| proc chart data=sasuser.y02jan;
    hbar week / sumvar=sale;
run; |
| proc plot data=sasuser.y02jan;
    plot sale*day |

The generated program is identical to the program in the previous two examples. That is, although you are changing the code that you submit, you are not changing the code that the compiler sees.

**Delimiters in Macro Variable Names**

Sometimes you might want to place a macro variable name immediately before text other than a special character. For example, you might want to modify the previous
Introducing Macro Variables

Delimiters in Macro Variable Names

program so that it is easy to switch between using the CHART and PLOT procedures of Base SAS software and the GCHART and GPLOT procedures of SAS/GRAPH software.

```sas
/* GRAPHICS should be null or G */
%let graphics=g;
%let year=02;
%let month=jan;
%let var=sale;
proc &graphics.chart data=sasuser.y&year&month;
  hbar week / sumvar=&var;
run;
proc &graphics.plot data=sasuser.y&year&month;
  plot &var*day;
run;
```

The messages written to the SAS log reveal problems with this program.

Table 9.17  SAS Log

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>%let graphics=g;</td>
</tr>
<tr>
<td>14</td>
<td>%let year=02;</td>
</tr>
<tr>
<td>15</td>
<td>%let month=jan;</td>
</tr>
<tr>
<td>16</td>
<td>%let var=sale;</td>
</tr>
<tr>
<td>17</td>
<td>proc &amp;graphics.chart data=sasuser.y&amp;year&amp;month;</td>
</tr>
</tbody>
</table>

- 10

WARNING: Apparent symbolic reference GRAPHICSCHART not resolved.

ERROR 10-205: Expecting the name of the procedure to be executed.

SAS interprets the macro variable’s name to be `graphicschart` instead of `graphics` because there is no delimiter between the macro variable reference and the trailing text. The word scanner recognizes the end of a macro variable name when it encounters a special character that cannot be part of the name token. In other words, the special character acts as a delimiter. For example, a period (.) is a special character that is treated as part of the macro variable reference and that does not appear when the macro variable is resolved.

To correct the problem in the previous example, you need to add a period after the reference to the macro variable `graphics`.

```sas
%let graphics=g;
%let year=02;
%let month=jan;
%let var=sale;
proc &graphics.chart data=sasuser.y&year&month;
  hbar week / sumvar=&var;
run;
proc &graphics.plot data=sasuser.y&year&month;
  plot &var*day;
run;
```

When these SAS statements are executed

- the word scanner treats `&graphics` as the reference
- the value of the macro variable `graphics` is returned to the input stack
- the word scanner processes `gchart` as one token.
Table 9.18  Code After Substitution

```sas
proc gchart data=sasuser.y02jan;
   hbar week / sumvar=sale;
run;
proc gplot data=sasuser.y02jan;
   plot sale*day;
run;
```

We can extend this example and further modify the previous program to include a macro variable that is used to define the libref:

```sas
%let lib=sasuser;
%let graphics=g;
%let year=02;
%let month=jan;
%let var=sale;
libname &lib 'SAS-data-library';
proc &graphics.chart data=&lib.y&year&month;
   hbar week / sumvar=&var;
run;
proc &graphics.plot data=&lib.y&year&month;
   plot &var*day;
run;
```

Notice, however, that this code does not perform the desired substitutions.

Table 9.19  Code After Substitution

```sas
libname sasuser 'SAS-data-library';
proc gchart data=sasuser.y02jan;
   hbar week / sumvar=sale;
run;
proc gplot data=sasuser.y02jan;
   plot sale*day;
run;
```

The period after &lib is interpreted as a delimiter. You need to use a second period after the delimiter period to supply the necessary token:

```sas
%let lib=sasuser;
...
libname &lib 'SAS-data-library';
proc &graphics.chart data=&lib..y&year&month;
...
proc &graphics.plot data=&lib..y&hear&month;
```

The first period is treated as a delimiter, and the second period is treated as text.

Table 9.20  Code After Substitution

```sas
proc gchart data=sasuser.y02jan;
   ...
proc gplot data=sasuser.y02jan;
```
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Basic Concepts

Macro variables can supply a variety of information, from operating system information, to SAS session information, to any text string that you define. Updating multiple references to a variable, data set, or text string is a simple process if you use macro variables in your programs. Macro variables are part of the SAS macro facility, which is a tool for extending and customizing SAS and for reducing the amount of text you must enter in order to perform common tasks.

Values of macro variables are stored in symbol tables. Values that are stored in the global symbol table are always available. In order to substitute the value of a macro variable in your program, you must reference that macro variable by preceding the macro variable name with an ampersand. You can reference a macro variable anywhere in a SAS program except within data lines.

Using Automatic Macro Variables

SAS provides automatic macro variables that contain information about your computing environment. Automatic macro variables are created when SAS is invoked. Many of these variables have fixed values that are assigned by SAS and which remain constant for the duration of your SAS session. Others have values that are updated automatically based on submitted SAS statements.

Using User-Defined Macro Variables

You can create and define your own macro variables with the %LET statement. The %LET statement enables you to assign a value for your new macro variable and to store that value in the global symbol table. Macro variable values are character strings; except for leading and trailing blanks, values are stored exactly as they appear in the assignment statement.

Processing Macro Variables

When submitted, a SAS program goes to an area of memory called the input stack. From there, the word scanner divides the program into small chunks called tokens and passes them to the appropriate compiler for eventual execution. Certain token sequences are macro triggers, which are sent to the macro processor for resolution. Once a macro variable has been resolved by the macro processor, the stored value is substituted back into the program in the input stack, and word scanning continues.

Displaying Macro Variable Values in the SAS Log

You can use the SYMBOLOGEN system option to monitor the value that is substituted for a macro variable reference. You can also use the %PUT statement to write messages, which can include macro variable values, to the SAS log.
Using Macro Functions to Mask Special Characters

The %STR function enables you to quote tokens during compilation in order to mask them from the macro processor. The %NRSTR function enables you to quote tokens that include macro triggers from the macro processor. The %BQUOTE function enables you to quote a character string or resolved value of a text expression during execution of a macro or macro language statement.

Using Macro Functions to Manipulate Character Strings

You can use macro character functions to apply character string manipulations to the values of macro variables. The %UPCASE function enables you to change values from lowercase to uppercase. The %QUPCASE function works the same as %UPCASE except that it also masks special characters and mnemonic operators. The %SUBSTR function enables you to extract part of a string from a macro variable value. The %QSUBSTR function works the same as %SUBSTR except that it also masks special characters and mnemonic operators. The %INDEX function enables you to determine the location of the first character of a character string within a source. Using the %SCAN function, you can extract words from the value of a macro variable. The %QSCAN function works the same as %SCAN except that it also masks special characters and mnemonic operators.

Using SAS Functions with Macro Variables

You can use the %SYSFUNC function to execute other SAS functions. The %QSYSFUNC function works the same as the %SYSFUNC function except that it also masks special characters and mnemonic operators.

Combining Macro Variable References with Text

You might sometimes need to combine a macro variable reference with other text. You can place text immediately before or immediately after a macro variable reference. You can also combine two macro variable references in order to create a new token. You might need to use a delimiter when you combine macro variable references with text.

Syntax

OPTIONS NOSYMBOLGEN | SYMBOLGEN;

%PUT text;
%LET variable=value;
%STR (argument)
%NRSTR (argument)
%BQUOTE (argument)
%UPCASE (argument)
%QUPCASE (argument)
%SUBSTR (argument, position <,n>)
%QSUBSTR (argument, position <,n>)
%INDEX (source,string)
%SCAN (argument, n <,delimiters>)
%QSCAN (argument, n <,delimiters>)
%SYSFUNC (function(argument(s)<,format>)
%QSYSFUNC (function(argument(s)<,format>)

Sample Programs

Creating Macro Variables with a %LET Statement

```sas
options symbolgen;
%let year=2002;
proc print data=sasuser.schedule;
  where year(begin_date)=&year;
  title "Scheduled Classes for &year";
run;
proc means data=sasuser.all sum;
  where year(begin_date)=&year;
  class location;
  var fee;
  title1 "Total Fees for &year Classes";
  title2 "by Training Center";
run;
```

Using Automatic Macro Variables

```sas
footnote1 "Created &systime &sysday, &sysdate9";
footnote2 "on the &sysscp system using Release &sysver";
title "REVENUES FOR DALLAS TRAINING CENTER";
proc tabulate data=sasuser.all(keep=location course_title fee);
  where upcase(location)="DALLAS";
  class course_title;
  var fee;
  table course_title=" " all="TOTALS",
                fee=" "+(n*f=3. sum*f=dollar10.)
      / rts=30 box="COURSE";
run;
```

Inserting Macro Variables Immediately After Text

```sas
%let year=02;
%let month=jan;
proc chart data=sasuser.y&year&month;
  hbar week / sumvar=sale;
run;
proc plot data=sasuser.y&year&month;
  plot sale*day;
run;
```

Inserting Macro Variables Immediately Before Text

```sas
%let graphics=g;
%let year=02;
%let month=jan;
%let var=sale;
proc &graphics.chart data=sasuser.y&year&month;
```
 Points to Remember

- Macro variables can make your programs more reusable and dynamic.
- When you submit code to SAS, macro variable references are resolved by the macro processor, and their values are substituted into your program.
- You can use the %PUT statement to write any text, including resolved macro variables, to the SAS log.
- If you reference a macro variable within quotation marks, you must use double quotation marks. Macro variable references that are enclosed in single quotation marks will not be resolved.
- Most macro character functions have corresponding functions (such as %QSUBSTR and %QSCAN) that also mask special characters and mnemonic operators.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Which of the following statements is false?
   a A macro variable can be defined and referenced anywhere in a SAS program except within data lines.
   b Macro variables are always user-defined, and their values remain constant until they are changed by the user.
   c Macro variables are text strings that are independent of SAS data sets.
   d The values of macro variables can be up to 65,534 characters long.

2 Which of the following TITLE statements correctly references the macro variable `month`?
   a title "Total Sales for ‘&month’ ";
   b title "Total Sales for ‘month’";
   c title "Total Sales for &month";
   d title Total Sales for "&month";

3 Which of the following statements does not correctly display the value of the macro variable `month` in the SAS log?
   a options &month;
   b PUT &month;
   c options symbolgen;
   d PUT the macro variable MONTH has the value &month;
4. Which statement will create a macro variable named `location` that has the value `storage`?
   a. `&let location = storage;`
   b. `let &location = storage;`
   c. `%let location = "storage";`
   d. `%let location = storage;`

5. What value will these statements assign to the macro variable `reptitle`:
   ```
   %let area = "Southeast";
   %let reptitle = * Sales Report for &area Area *;
   ```
   a. `Sales Report for Southeast Area`
   b. `Sales Report for "Southeast" Area`
   c. `"Sales Report for "Southeast" Area"`
   d. `* Sales Report for "Southeast" Area *`

6. Assuming that you began your SAS session today, which of the following statements correctly sets the macro variable `currdate` to today’s date:
   a. `%let currdate = %sysfunc(today(), worddate.);`
   b. `%let currdate = &sysdate9;`
   c. `%let currdate = %sysfunc(date());`
   d. all of the above

7. Macro character functions
   a. can be used to manipulate character strings in macro variable values.
   b. have the same basic syntax as the corresponding DATA step functions and yield similar results.
   c. all of the above
   d. none of the above

8. The four types of tokens that SAS recognizes are
   a. expressions, literals, names, and special characters.
   b. literals, names, numbers, and special characters.
   c. expressions, names, numbers, and special characters.
   d. expressions, literals, numbers, and special characters.

9. What are the resulting values for the macro variables that are defined here?
   ```
   %let month1 = June;
   %let month2 = July;
   %let period1 = &month1&month2;
   %let period2 = May&month1;
   %let period3 = &month2.Aug;
   ```
   a. `month1 June
      month2 July
      period1 June July
      period2 May June
      period3 July Aug`
   b. `month1 June
      month2 July
      period1 JuneJuly
      period2 MayJune
      period3 July.Aug`
10. Which of the following correctly produces a title in which the current date is left justified in order to remove extra blanks?

a. `title “Report for %sysfunc(left(%sysfunc(today(), worddate.)))”;

b. `title “Report for %sysfunc(left(today(), worddate.))”;

c. `title “Report for %sysfunc(left(%qsysfunc(today(), worddate.)))”;

d. `title “Report for %left(today(), worddate.)”;

---

c. month1 June
month2 July
period1 JuneJuly
period2 MayJune
period3 JulyAug

d. month1 June
month2 July
period1 junejuly
period2 Mayjune
period3 julyaug
Processing Macro Variables at Execution Time

Overview 326
Introduction 326
Objectives 327
Prerequisites 327
Creating a Macro Variable During DATA Step Execution 327
  Example 329
  The SYMPUT Routine 330
  Using SYMPUT with a Literal 331
  Example 331
  Using SYMPUT with a DATA Step Variable 332
  Example 333
  The SYMPUTX Routine 334
  Example 335
  Using CALL SYMPUT with DATA Step Expressions 336
  Example 338
  PUT Function 339
  Example 340
Creating Multiple Macro Variables During DATA Step Execution 341
  Creating Multiple Macro Variables with CALL SYMPUT 341
  Example 342
Referencing Macro Variables Indirectly 344
  Introduction 344
  The Forward Re-Scan Rule 345
  Example 345
  Example 346
  Example 348
Obtaining Macro Variable Values During DATA Step Execution 350
  The SYMGET Function 350
  Example 351
Creating Macro Variables During PROC SQL Step Execution 352
  The INTO Clause and the NOPRINT Option 352
  Example 353
  Creating Variables with the INTO Clause 354
  Example 355
  Example 356
  Creating a Delimited List of Values 357
  Example 358
Working with PROC SQL Views 359
Using Macro Variables in SCL Programs 360
  The SYMPUTN Routine 360
  Example 361
Overview

Introduction

Because the macro facility performs its tasks before SAS programs execute, the information that the macro facility supplies does not depend on values that are accessed or computed during the execution of a SAS program. However, sometimes it is necessary to access or create macro variables during the execution of a SAS program. There are several methods that enable the macro facility to create or access macro variables at execution time. In this chapter, you learn to use macro variables during execution of

- a DATA step
- a PROC SQL step
- an SCL program.
Objectives
In this chapter, you learn to
- create macro variables during DATA step execution
- describe the difference between the %LET statement and the SYMPUT routine
- reference macro variables indirectly, using multiple ampersands for delayed resolution
- obtain the value of a macro variable during DATA step execution
- describe the difference between the SYMGET function and macro variable references
- create macro variables during PROC SQL execution
- store several values in one macro variable, using the SQL procedure
- create, update, and obtain the values of macro variables during the execution of an SCL program.

Prerequisites
Before beginning this chapter, you should complete the following chapters:
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243
- Chapter 9, “Introducing Macro Variables,” on page 287.

Creating a Macro Variable During DATA Step Execution
In many applications, you need to create macro variables during DATA step execution. You might need to create macro variables and to assign values to them based on
- data values in SAS data sets or in external files
- programming logic
- computed values.

For example, suppose you want to create a report that lists students who are enrolled in a specific course, according to data in the Sasuser.All data set. Suppose you want to include a footnote in your report to indicate whether any student fees are unpaid.

The following program uses SAS programming logic to determine which value is assigned to the macro variable foot. Then foot is referenced in the FOOTNOTE statement later in the program.

```sas
options symbolgen pagesize=30;
%let crsnum=3;
data revenue;
   set sasuser.all end=final;
   where course_number=&crsnum;
   total+1;
   if paid='Y' then paidup+1;
   if final then do;
      put total= paidup=; /* Write information to the log. */
   end;
   if paidup<total then do;
      %let foot=Some Fees Are Unpaid;
   end;
```

```sas
%let foot=Some Fees Are Unpaid;
```

```sas
end;
```
else do;
    %let foot=All Students Have Paid;
end;
end;
run;

proc print data=revenue;
  var student_name student_company paid;
  title "Payment Status for Course &crsnum";
  footnote "&foot";
run;

Running the program produces the following report:

### Payment Status for Course 3

<table>
<thead>
<tr>
<th>Obs</th>
<th>Student_Name</th>
<th>Student_Company</th>
<th>Paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bills, Ms. Paulette</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>Chevarley, Ms. Arlene</td>
<td>Motor Communications</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>Clough, Ms. Patti</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>Crace, Mr. Ron</td>
<td>Von Crump Seafood</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>Davis, Mr. Bruce</td>
<td>SemiConductor</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>Elsins, Ms. Marisa F.</td>
<td>SSS Inc.</td>
<td>N</td>
</tr>
<tr>
<td>7</td>
<td>Gandy, Dr. David</td>
<td>Paralegal Assoc.</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>Gash, Ms. Hedy</td>
<td>QA Information Systems Center</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>Haubold, Ms. Ann</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>Hudock, Ms. Cathy</td>
<td>So. Cal. Medical Center</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>Kimble, Mr. John</td>
<td>Alforone Chemical</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>Kocher, Mr. Dennis</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>13</td>
<td>Larocque, Mr. Bret</td>
<td>Physicians IPA</td>
<td>Y</td>
</tr>
<tr>
<td>14</td>
<td>Licht, Mr. Eryan</td>
<td>SII</td>
<td>Y</td>
</tr>
<tr>
<td>15</td>
<td>McKnight, Ms. Mauree E.</td>
<td>Federated Bank</td>
<td>Y</td>
</tr>
<tr>
<td>16</td>
<td>Scannell, Ms. Robin</td>
<td>Amberley Corp.</td>
<td>N</td>
</tr>
<tr>
<td>17</td>
<td>Seitz, Mr. Adam</td>
<td>Lomax Services</td>
<td>Y</td>
</tr>
<tr>
<td>18</td>
<td>Smith, Ms. Jan</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>19</td>
<td>Sulzbach, Mr. Bill</td>
<td>Sailbest Ships</td>
<td>Y</td>
</tr>
<tr>
<td>20</td>
<td>Williams, Mr. Gene</td>
<td>Snowing Petroleum</td>
<td>Y</td>
</tr>
</tbody>
</table>

### All Students Have Paid

Although you can see that several students still have unpaid fees, the footnote indicates that all students have paid. Obviously, the footnote is wrong. That is, the
macro variable **foot** resolves to the value *All Students Have Paid* when it should not do so. Let’s take a closer look at what’s going on here.

---

**Example**

In order to understand the problem with this example, you should consider how macro variable processing works in conjunction with SAS processing. Remember that when both macro language statements and SAS language statements occur in the same step, the *macro processor* executes macro language statements *before* any SAS language statements are executed.

Remember, you want to create a report that lists students who are enrolled in a specific course, according to data in the *Sasuser.All* data set, and you want to include a footnote in your report to indicate whether any student fees are unpaid. The following program uses SAS programming logic to determine which value is assigned to the macro variable **foot**. Then **foot** is referenced in the FOOTNOTE statement later in the program.

```sas
options symbolgen pagesize=30;
%let crsnum=3;
data revenue;
  set sasuser.all end=final;
  where course_number=&crsnum;
  total+1;
  if paid='Y' then paidup+1;
  if final then do;
    put total= paidup=; /* Write information to the log. */
    if paidup<total then do;
      %let foot=Some Fees Are Unpaid;
    end;
    else do;
      %let foot=All Students Have Paid;
    end;
  end;
run;

proc print data=revenue;
  var student_name student_company paid;
  title "Payment Status for Course &crsnum";
  footnote "&foot";
run;
```

In this example, the first `%LET` statement inside the DATA step is passed to the macro processor as soon as the word scanner encounters it. The macro processor then creates a macro variable named **foot** in the symbol table and assigns the value *Some Fees Are Unpaid* to the variable.

The word scanner then continues to read the program and passes the second `%LET` statement in the DATA step to the macro processor as well. This time, the macro processor reassigns the value *All Students Have Paid* to **foot** in the symbol table.

When the RUN statement in the DATA step is encountered, SAS recognizes that the step is complete, and executes it. Remember that at this point the DATA step no longer includes any of the `%LET` statements (which have already been executed by the macro processor). Because the `%LET` statements are *always* processed by the macro processor *before* the DATA step is executed, the value of **foot** will always be whatever the last `%LET` statement assigns.
Here is a representation of the program that is processed by the data step compiler as a result of the above code.

**Table 10.1  Code After Substitution**

```sas
data revenue;
  set sasuser.all end=final;
  where course_number=3;
  total+1;
  if paid='Y' then paidup+1;
  if final then do;
    put total= paidup=;
    if paidup<total then do;
      end;
    else do;
      end;
  end;
run;
proc print data=revenue;
  var student_name student_company paid;
  title "Payment Status for Course 3";
  footnote "All Students Have Paid";
run;
```

Let's look at a way to solve this problem.

**The SYMPUT Routine**

The DATA step provides functions and CALL routine that enable you to transfer information between an executing DATA step and the macro processor. You can use the SYMPUT routine to create a macro variable and to assign to that variable any value that is available in the DATA step.

General form, SYMPUT routine:

```
CALL SYMPUT(macro-variable,text);
```

where

- `macro-variable` is assigned the character value of `text`.
- `macro-variable` and `text` can each be specified as
  - a literal, enclosed in quotation marks
  - a DATA step variable
  - a DATA step expression.

**Note:** If `macro-variable` already exists, the value of `text` replaces the former value.

When you use the SYMPUT routine to create a macro variable in a DATA step, the macro variable is not actually created and assigned a value until the DATA step is executed. Therefore, you cannot successfully reference a macro variable that is created with the SYMPUT routine by preceding its name with an ampersand within the same DATA step in which it is created.
In the next few sections you will see several examples of how the SYMPUT routine can be used in different situations.

**Using SYMPUT with a Literal**

In the SYMPUT routine, you use a literal string for
- the first argument to specify an exact name for the name of the macro variable
- the second argument to specify the exact character value to assign to the macro variable.

To use a literal with the SYMPUT routine, you enclose the literal string in quotation marks.

```sas
CALL SYMPUT('macro-variable', 'text');
```

**Example**

Remember the previous example, in which you wanted to conditionally assign a value to the macro variable `foot` based on values that are generated during DATA step execution. You can use the SYMPUT routine with literal strings as both arguments in order to accomplish this.

```sas
options symbolgen pagesize=30;
%let crsnum=3;
data revenue;
  set sasuser.all end=final;
  where course_number=&crsnum;
  total+1;
  if paid='Y' then paidup+1;
  if final then do;
    if paidup<total then do;
      call symput('foot','Some Fees Are Unpaid');
    end;
    else do;
      call symput('foot','All Students Have Paid');
    end;
  end;
run;

proc print data=revenue;
  var student_name student_company paid;
  title "Payment Status for Course &crsnum";
  footnote "&foot";
run;
```

This time, the value assigned to `foot` is either *Some Fees Are Unpaid* or *All Students Have Paid*, depending on the value of the DATA step variable `Paidup`, because the value is assigned during the execution of the DATA step. When you submit this code, you get the following output.
Using SYMPUT with a DATA Step Variable

You can assign the value of a DATA step variable as the value for a macro variable by using the DATA step variable's name as the second argument to the SYMPUT routine.

To use a DATA step variable as the value for a macro variable in the SYMPUT routine, you place the name of the DATA step variable after the name of the macro variable, separated by a comma. You do not enclose the name of the DATA step variable in quotation marks.

```
CALL SYMPUT('macro-variable',DATA-step-variable);
```

This form of the SYMPUT routine creates the macro variable named `macro-variable` and assigns to it the current value of `DATA-step-variable`.

When you use a DATA step variable as the second argument,

- a maximum of 32767 characters can be assigned to the receiving macro variable.
- any leading or trailing blanks that are part of the DATA step variable's value are stored in the macro variable.
values of numeric variables are automatically converted to character values, using the BEST12. format.

**CAUTION:**
If you enclose the DATA step variable name in quotation marks, SAS interprets the name as a literal value rather than as a variable name, and the DATA step variable’s value is not resolved.

---

**Example**

Once again, suppose you want to create a report about students who are enrolled in a particular course. This time, suppose you want to add a title that contains the course title and the course number, and you want to include a footnote that summarizes how many students have paid their fees.

In this example, a DATA step variable named `paidup` records the number of students that have paid, and a DATA step variable named `total` records the total number of students who are registered for the class. Macro variables are created to record the values of `paidup`, the value of `total`, and the value of `Course_title`. These macro variables are referenced later in the program.

```
%let crsnum=3;
data revenue;
  set sasuser.all end=final;
  where course_number=&crsnum;
  total+1;
  if paid='Y' then paidup+1;
  if final then do;
    call symput('numpaid',paidup);
    call symput('numstu',total);
    call symput('crsname',course_title);
  end;
run;
proc print data=revenue noobs;
  var student_name student_company paid;
  title "Fee Status for &crsname (\&crsnum)";
  footnote "Note: &numpaid Paid out of &numstu Students";
run;
```

This time the footnote shows the correct information for how many students have paid.
Fee Status for Local Area Networks (#3)

<table>
<thead>
<tr>
<th>Student_Name</th>
<th>Student_Company</th>
<th>Paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bills, Ms. Paulette</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Chevarley, Ms. Arlene</td>
<td>Motor Communications</td>
<td>N</td>
</tr>
<tr>
<td>Clough, Ms. Patti</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>Crace, Mr. Ron</td>
<td>Von Crump Seafood</td>
<td>Y</td>
</tr>
<tr>
<td>Davis, Mr. Bruce</td>
<td>Semiconductors</td>
<td>Y</td>
</tr>
<tr>
<td>Elsins, Ms. Marisa F.</td>
<td>SSS Inc.</td>
<td>N</td>
</tr>
<tr>
<td>Gandy, Dr. David</td>
<td>Paralegal Assoc.</td>
<td>Y</td>
</tr>
<tr>
<td>Gash, Ms. Hedy</td>
<td>QA Information Systems Center</td>
<td>Y</td>
</tr>
<tr>
<td>Haubold, Ms. Ann</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Hudock, Ms. Cathy</td>
<td>So. Cal. Medical Center</td>
<td>Y</td>
</tr>
<tr>
<td>Kimble, Mr. John</td>
<td>Alfrone Chemical</td>
<td>N</td>
</tr>
<tr>
<td>Kochen, Mr. Dennis</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Larocque, Mr. Eric</td>
<td>Physicians IPA</td>
<td>Y</td>
</tr>
<tr>
<td>Licht, Mr. Bryan</td>
<td>SII</td>
<td>Y</td>
</tr>
<tr>
<td>McKnight, Ms. Maureen E.</td>
<td>Federated Bank</td>
<td>Y</td>
</tr>
<tr>
<td>Scannell, Ms. Robin</td>
<td>Amberly Corp.</td>
<td>N</td>
</tr>
<tr>
<td>Seitz, Mr. Adam</td>
<td>Lomax Services</td>
<td>Y</td>
</tr>
<tr>
<td>Smith, Ms. Jan</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>Sulzbach, Mr. Bill</td>
<td>Sailbest Ships</td>
<td>Y</td>
</tr>
<tr>
<td>Williams, Mr. Gene</td>
<td>Snowing Petroleum</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note: 14 Paid out of 20 Students

The SYMPUTX Routine

The SYMPUTX routine is very similar to the SYMPUT routine. In addition to creating a macro variable and assigning a value to it, the SYMPUTX routine also automatically removes leading and trailing blanks from both arguments. The SYMPUTX routine is available beginning in SAS 9.
General form, SYMPUTX routine:

\[
\text{CALL SYMPUTX}(\text{macro-variable,expression});
\]

where

- \text{macro-variable} is assigned the character value of \text{expression}, and any leading or trailing blanks are removed from both \text{macro-variable} and \text{expression}.

- \text{macro-variable} and \text{expression} can each be specified as
  - a literal, enclosed in quotation marks
  - a DATA step variable
  - a DATA step expression.

\textbf{Note:} If \text{macro-variable} already exists, the value of \text{expression} replaces the former value.

\section*{Example}

Remember the example where you created a report about students who are enrolled in a particular course. This time, suppose you want the title to contain the course name and the course number, as well as the date on which the course was held. Also, you want the footnote to list the current amount of unpaid fees for the course.

In this example, three macro variables are created. The macro variable \text{crsnum} records the value of the DATA step variable \text{Course_title}. The macro variable \text{date} records the value of the DATA step variable \text{Begin_date} in MMDDYY10. format. Finally, the macro variable \text{due} uses the values of the DATA step variables \text{paidup}, \text{total}, and \text{fee} to record the current amount of unpaid fees in DOLLAR8. format. These macro variables are referenced later in the program in the title and footnote statements.

\begin{verbatim}
%let crsnum=3;

data revenue;
  set sasuser.all end=final;
  where course_number=&crsnum;
  total+1;
  if paid='Y' then paidup+1;
  if final then do;
    call symputx('crsname',course_title);
    call symputx('date',put(begin_date,mmddyy10.));
    call symputx('due',put(fee*(total-paidup),dollar8.));
  end;
run;

proc print data=revenue;
  var student_name student_company paid;
  title "Fee Status for &crsname (#&crsnum) Held &date";
  footnote "Note: &due in Unpaid Fees";
run;
\end{verbatim}
If you had run the last example using listing output rather than HTML output, you would have seen extra blanks in the title between the course title and the course number, as well as in the footnote.
Table 10.2  SAS Listing Output

<table>
<thead>
<tr>
<th>Student_Name</th>
<th>Student_Company</th>
<th>Paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bills, Ms. Paulette</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Chevarley, Ms. Arlene</td>
<td>Motor Communications</td>
<td>N</td>
</tr>
<tr>
<td>Clough, Ms. Patti</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>Crace, Mr. Ron</td>
<td>Von Crump Seafood</td>
<td>Y</td>
</tr>
<tr>
<td>Davis, Mr. Bruce</td>
<td>Semi;Conductor</td>
<td>Y</td>
</tr>
<tr>
<td>Elsins, Ms. Marisa F.</td>
<td>SSS Inc.</td>
<td>N</td>
</tr>
<tr>
<td>Gandy, Dr. David</td>
<td>Paralegal Assoc.</td>
<td>Y</td>
</tr>
<tr>
<td>Gash, Ms. Hedy</td>
<td>QA Information Systems Center</td>
<td>Y</td>
</tr>
<tr>
<td>Haubold, Ms. Ann</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Hudock, Ms. Cathy</td>
<td>So. Cal. Medical Center</td>
<td>Y</td>
</tr>
<tr>
<td>Kimble, Mr. John</td>
<td>Alforone Chemical</td>
<td>N</td>
</tr>
<tr>
<td>Kochen, Mr. Dennis</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Larocque, Mr. Bret</td>
<td>Physicians IPA</td>
<td>Y</td>
</tr>
<tr>
<td>Licht, Mr. Bryan</td>
<td>SII</td>
<td>Y</td>
</tr>
<tr>
<td>McKnight, Ms. Maureen E.</td>
<td>Federated Bank</td>
<td>Y</td>
</tr>
<tr>
<td>Scannell, Ms. Robin</td>
<td>Amberly Corp.</td>
<td>N</td>
</tr>
<tr>
<td>Seitz, Mr. Adam</td>
<td>Lomax Services</td>
<td>Y</td>
</tr>
<tr>
<td>Smith, Ms. Jan</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>Sulzbach, Mr. Bill</td>
<td>Sailbest Ships</td>
<td>Y</td>
</tr>
<tr>
<td>Williams, Mr. Gene</td>
<td>Snowing Petroleum</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note: 14 Paid out of 20 Students

You do not see these blanks if you are using HTML output, but they are still stored in the value of your macro variable.

Remember that when a DATA step variable is used as the second argument in a SYMPUT routine, any leading or trailing blanks that are part of the DATA step variable’s value are stored in the macro variable. Because the value of a macro variable is always a text string, numeric variables are automatically converted using the BEST12. format, and blanks are stored as part of the macro variable’s value. In order to avoid including extra blanks, you need to use a DATA step function to remove them.

In these situations you can use DATA step functions before the SYMPUT routine executes, in order to

- left-align character strings that have been created by numeric-to-character conversions
- remove extraneous leading and trailing blanks.

Often you will want to combine several DATA step functions in order to create a DATA step expression as the second argument of the SYMPUT routine.

**CALL SYMPUT(macro-variable,expression);**

*Note:* A DATA step expression can be any combination of DATA step functions, DATA step variables, constants, and logical or arithmetic operators that resolves to a character or numeric constant.

When you use a DATA step expression as the second argument, its current value is evaluated according to the following rules:

- Numeric expressions are automatically converted to character constants using the BEST12. format.
- The resulting value can be up to 32767 characters long.
- Any leading or trailing blanks that are part of the expression are stored in the macro variable.
Example

In order to remove the extra blanks from the title and footnote of the previous example, you can use DATA step functions. To remove leading blanks from the macro variables `numstu` and `numpaid`, you can use the LEFT function to left-justify values and the TRIM function to remove any trailing blanks. To remove trailing blanks from `crsname`, you only need to use the TRIM function, because `crsname` was created from a character variable.

```sas
%let crsnum=3;
data revenue;
set sasuser.all end=final;
where course_number=&crsnum;
total+1;
if paid='Y' then paidup+1;
if final then do;
call symput('numpaid',trim(left(paidup)));
call symput('numstu',trim(left(total)));
call symput('crsname',trim(course_title));
end;
run;

proc print data=revenue noobs;
var student_name student_company paid;
title "Fee Status for &crsname (#&crsnum);"
footnote "Note: &numpaid Paid out of &numstu Students"
run;
```
Table 10.3  SAS Listing Output

<table>
<thead>
<tr>
<th>NAME</th>
<th>COMPANY</th>
<th>PAID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bills, Ms. Paulette</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Chevarley, Ms. Arlene</td>
<td>Motor Communications</td>
<td>N</td>
</tr>
<tr>
<td>Clough, Ms. Patti</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>Crace, Mr. Ron</td>
<td>Von Crump Seafood</td>
<td>Y</td>
</tr>
<tr>
<td>Davis, Mr. Bruce</td>
<td>Semi;Conductor</td>
<td>Y</td>
</tr>
<tr>
<td>Elsins, Ms. Marisa F.</td>
<td>SSS Inc.</td>
<td>N</td>
</tr>
<tr>
<td>Gandy, Dr. David</td>
<td>Paralegal Assoc.</td>
<td>Y</td>
</tr>
<tr>
<td>Gash, Ms. Hedy</td>
<td>QA Information Systems Center</td>
<td>Y</td>
</tr>
<tr>
<td>Haubold, Ms. Ann</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Hudock, Ms. Cathy</td>
<td>So. Cal. Medical Center</td>
<td>Y</td>
</tr>
<tr>
<td>Kimble, Mr. John</td>
<td>Alforone Chemical</td>
<td>N</td>
</tr>
<tr>
<td>Kochen, Mr. Dennis</td>
<td>Reston Railway</td>
<td>Y</td>
</tr>
<tr>
<td>Larocque, Mr. Bret</td>
<td>Physicians IPA</td>
<td>Y</td>
</tr>
<tr>
<td>Licht, Mr. Bryan</td>
<td>SII</td>
<td>Y</td>
</tr>
<tr>
<td>McKnight, Ms. Maureen E.</td>
<td>Federated Bank</td>
<td>Y</td>
</tr>
<tr>
<td>Scannell, Ms. Robin</td>
<td>Amberly Corp.</td>
<td>N</td>
</tr>
<tr>
<td>Seitz, Mr. Adam</td>
<td>Lomax Services</td>
<td>Y</td>
</tr>
<tr>
<td>Smith, Ms. Jan</td>
<td>Reston Railway</td>
<td>N</td>
</tr>
<tr>
<td>Sulzbach, Mr. Bill</td>
<td>Sailbest Ships</td>
<td>Y</td>
</tr>
<tr>
<td>Williams, Mr. Gene</td>
<td>Snowing Petroleum</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note: 14 Paid out of 20 Students

**PUT Function**

Remember that the values of macro variables are always character strings. You have seen that in the DATA step the SYMPUT routine will perform automatic numeric-to-character conversion on any numeric value that you attempt to assign to a macro variable. Messages are written to the SAS log to alert you that automatic conversion has occurred. Remember that the SYMPUT routine automatically uses the BEST12. format for the conversion.

Sometimes you might want to have explicit control over the numeric-to-character conversion. The PUT function returns a character string that is formed by writing a value with a specified format.

You can use the PUT function to
- perform explicit numeric-to-character conversions
- format the result of a numeric expression.
General form, PUT function:

\textbf{PUT}(source,format.)

where

\textit{source}

is a constant, a variable, or an expression (numeric or character).

\textit{format.}

is any SAS format or user-defined format, which determines

- the length of the resulting string
- whether the string is right- or left-aligned.

\textit{source} and \textit{format.}

must be the same type (numeric or character).

---

\textbf{Example}

Suppose you want to create a report that shows the amount of fees that are unpaid for a specific course. In the following example, you use the SYMPUT routine to format the value of the numeric variable \textbf{Begin date} with the MMDDYY10. format and assign that value to the macro variable \textbf{date}. Then you also use another call to the SYMPUT routine to format the result of an expression involving \textbf{Fee}, \textbf{total}, and \textbf{paidup} as a dollar amount and assign that value to the macro variable \textbf{due}.

\begin{verbatim}
%let crsnum=3;

data revenue;
  set sasuser.all end=final;
  where course_number=&crsnum;
  total+1;
  if paid='Y' then paidup+1;
  if final then do;
    call symput('crsname',trim(course_title));
    call symput('date',put(begin_date,mmddyy10.));
    call symput('due',
      trim(left(put(fee*(total-paidup),dollar8.))));
  end;
run;

You can use the macro variables \textbf{date} and \textbf{due} in a PROC PRINT step to create your report. The values of these macro variables appear in the report with the formatting that you assigned to them when you created them.

\begin{verbatim}
proc print data=revenue;
  var student_name student_company paid;
  title "Fee Status for &crsname (#&crsnum) Held &date";
  footnote "Note: &due in Unpaid Fees";
run;
\end{verbatim}
\end{verbatim}
Creating Multiple Macro Variables with CALL SYMPUT

Sometimes you might want to create multiple macro variables within one DATA step. For example, suppose you want to write a program that will list all of the scheduled dates for a particular course, using a macro variable to record the title of the course.

```sas
%let crsid=C005;
data _null_
   set sasuser.courses;
   where course_code="&crsid";
   call symput('title',trim(course_title));
run;

proc print data=sasuser.schedule noobs label;
   where course_code="&crsid";
   var location begin_date teacher;
   title1 "Schedule for &title";
```
options nodate nonumber;
run;

In this example, the value of the data set variable `Course_title` for the course whose `Course_code` is `C005` is assigned as a value for the macro variable `title`. In order to create a listing for a different course, you would need to change the `%LET` statement and resubmit the DATA step to assign a new value to `title`. Then you would need to resubmit the PROC PRINT step. Although you would need to resubmit both the DATA step and the PROC PRINT step, these two steps would be identical to the steps that you submitted for the first report. This is an extremely inefficient program.

```sas
%let crsid=C004;
data _null_;    
set sasuser.courses;    
where course_code="&crsid";    
call symput(‘title’,trim(course_title));    
run;

proc print data=sasuser.schedule noobs label;    
where course_code="&crsid";    
var location begin_date teacher;    
title1 “Schedule for &title”;    
options nodate nonumber;    
run;
```

Instead of executing separate DATA steps to update the same macro variable, you can create related macro variables in one DATA step. To create multiple macro variables, you use the SYMPUT routine with DATA step expressions for both arguments.

**General form, SYMPUT routine with DATA step expressions:**

```sas
CALL SYMPUT(expression1,expression2);
```

where

- `expression1` evaluates to a character value that is a valid macro variable name. This value should change each time you want to create another macro variable.
- `expression2` is the value that you want to assign to a specific macro variable.

**Example**

In this example, you use one call to the SYMPUT routine in order to create one macro variable for each value of the DATA step variable `Course_code` and to assign the corresponding value of `Course_title` to each macro variable. That is, for each observation in `Sasuser.Courses`, the macro processor will create a new macro variable. The new macro variable will have the same name as the value of the data set variable `Course_code` for that observation. The value of the new macro variable will be the value of the data set variable `Course_title` for that observation.

```sas
data _null_;    
set sasuser.courses;    
call symput(course_code, trim(course_title));
```
run;
%put _user_

The SAS log shows that six observations were read from the data set Sasuser.Courses and that six global macro variables were created and were assigned values.

Table 10.4  SAS Log

```
2   data _null_;  
3     set sasuser.courses;  
4     call symput(course_code, trim(course_title));  
5   run;  

NOTE: There were 6 observations read from the dataset SASUSER.COURSES.  
NOTE: DATA statement used:  
    real time 0.52 seconds  
    cpu time 0.13 seconds  

7   %put _user_;  
GLOBAL C006 Computer Aided Design  
GLOBAL C001 Basic Telecommunications  
GLOBAL C002 Structured Query Language  
GLOBAL C003 Local Area Networks  
GLOBAL C004 Database Design  
GLOBAL C005 Artificial Intelligence  
```

You can then use these new macro variables to print listings of information for various courses, using only one DATA step, as follows:

```sas
data _null_;  
  set sasuser.courses;  
  call symput(course_code,trim(course_title));  
run;

%let crsid=C005;  
proc print data=sasuser.schedule noobs label;  
  where course_code="&crsid";  
  var location begin_date teacher;  
  title1 "Schedule for &c005";  
run;

%let crsid=C002;  
proc print data=sasuser.schedule noobs label;  
  where course_code="&crsid";  
  var location begin_date teacher;  
  title1 "Schedule for &c002";  
run;
```

This is the output from the first PROC PRINT step.
Referencing Macro Variables Indirectly

**Introduction**

In the last example, you saw how to use the SYMPUT routine to create a series of macro variables whose names are based on the values of `Course_code`. However, you still needed to modify the TITLE statement in each PROC PRINT step in order to print output for each course.

Suppose you want to write a PROC PRINT step that you can reuse without any modification to print information about each course. You can do this by using an *indirect reference* in the TITLE statement.

```sas
data _null_;  
set sasuser.courses;  
call symput(course_code,trim(course_title));  
runc;

%let crsid=C002;  
proc print data=sasuser.schedule noobs label;  
  where course_code="&crsid";  
  var location begin_date teacher;  
  title1 "Schedule for ???";  
runc;
```

In the example above, the macro variable `c002` (as created by the SYMPUT routine) has a value of *Structured Query Language*. Therefore, the TITLE statement should reference a macro variable that will resolve to *Structured Query Language*. Remember...
that you want this reference to be flexible enough to apply to any of the macro variables that the SYMPUT routine creates, such as C003 or C004, by changing only the %LET statement.

To obtain the value *Structured Query Language*, you need to indirectly reference the macro variable C002 through a reference to the macro variable crsid. If the value of the macro variable crsid is C002, then you need to proceed in several steps:

1. Resolve the macro variable crsid to the value C002.
2. Attach an ampersand (&) to the front of the resolved value in order to create a new reference (&C002).
3. Resolve the resulting macro variable reference to the value *Structured Query Language*.

This sequence seems to imply that you should use the reference &&crsid to convert the value of the macro variable crsid to the corresponding course description. However, the Forward Re-Scan rule indicates that this is not the correct solution. Let’s look at this rule.

**The Forward Re-Scan Rule**

The Forward Re-Scan rule can be summarized as follows:

- When multiple ampersands or percent signs precede a name token, the macro processor resolves two ampersands (&&) to one ampersand (&), and re-scans the reference.

- To re-scan a reference, the macro processor scans and resolves tokens from left to right from the point where multiple ampersands or percent signs are coded, until no more triggers can be resolved.

According to the Forward Re-Scan rule, you need to use *three ampersands* in front of a macro variable name when its value matches the name of a second macro variable. This indirect reference resolves to the value of the second macro variable.

**Example**

Suppose you want to use the macro variable crsid to indirectly reference the macro variable C002.

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>C001</td>
</tr>
<tr>
<td>C002</td>
</tr>
<tr>
<td>C003</td>
</tr>
<tr>
<td>C004</td>
</tr>
<tr>
<td>C005</td>
</tr>
<tr>
<td>C006</td>
</tr>
<tr>
<td>CRSID</td>
</tr>
</tbody>
</table>

The following table shows several different references along with their resolved values.
By preceding a macro variable reference with two ampersands, you delay the resolution of the reference until the second scan. The first time the reference is scanned, only the double ampersands will be resolved (to one ampersand). In order to create an indirect reference (a reference whose value is a reference to a different macro variable), you must use three ampersands. Therefore, to use an indirect reference that resolves to *Structured Query Language*, the original reference must be `&&&crsid`.

### Example

You can use indirect referencing to improve the last example. By using an indirect reference to the macro variable whose name is the same as the current value of the macro variable `crsid`, you can write a PROC PRINT step that you can reuse without modification in order to print a report for each different course.

```sas
options symbolgen;
   data _null_;
      set sasuser.courses;
      call symput(course_code, trim(course_title));
   run;

%let crsid=C005;
proc print data=sasuser.schedule noobs label;
   where course_code="&crsid";
   var location begin_date teacher;
   title1 "Schedule for &&&crsid";
run;

%let crsid=C002;
proc print data=sasuser.schedule noobs label;
   where course_code="&crsid";
   var location begin_date teacher;
   title1 "Schedule for &&&crsid";
run;
```

The SAS log shows the steps that lead to the resolution of these macro variables for each PROC PRINT step.
Table 10.5  SAS Log

43 options symbolgen;
44 data _null_
45 set sasuser.courses;
46 call symput(course_code, trim(course_title));
47 run;
NOTE: There were 6 observations read from the dataset
   SASUSER.COURSES.
NOTE: DATA statement used:
   real time 0.07 seconds
   cpu time 0.05 seconds

48
49 %let crsid=C005;
50 proc print data=sasuser.schedule noobs label;
51   where course_code="&crsid";
SYMBOLGEN: Macro variable CRSID resolves to C005
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable CRSID resolves to C005
SYMBOLGEN: Macro variable C005 resolves to Artificial
   Intelligence
52   title1 "Schedule for &crsid";
53 run;
NOTE: There were 3 observations read from the dataset
   SASUSER.SCHEDULE.
   WHERE course_code='C005';
NOTE: PROCEDURE PRINT used:
   real time 0.09 seconds
   cpu time 0.04 seconds

55
56 %let crsid=C002;
57 proc print data=sasuser.schedule noobs label;
58   where course_code="&crsid";
SYMBOLGEN: Macro variable CRSID resolves to C002
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable CRSID resolves to C002
SYMBOLGEN: Macro variable C002 resolves to Structured
   Query Language
59   title1 "Schedule for &crsid";
60 run;
NOTE: There were 3 observations read from the dataset
   SASUSER.SCHEDULE.
   WHERE course_code='C002';
NOTE: PROCEDURE PRINT used:
   real time 0.06 seconds
   cpu time 0.04 seconds

This is the output from the first PROC PRINT step.
This is the output from the second PROC PRINT step.

Note that the PROC PRINT steps that produced these reports were identical. Only the %LET statement that precedes each PROC PRINT step and the resolved values of the macro variables changed.

Indirect referencing is especially useful when you are working with a series of related macro variables. In Chapter 9, “Introducing Macro Variables,” on page 287, you learned how to combine multiple macro variable references in order to build new tokens. You can combine indirect macro variable references with other macro variable references as well. That is, you can use two ampersands in a reference when the value of one macro variable matches part of the name of a second macro variable.

Example

You can create a series of macro variables, teach1 to teachn, each containing the name of the instructor who is assigned to a specific course.

```sas
options symbolgen;
data _null_;
  set sasuser.schedule;
  call symput('teach'||left(course_number),
    trim(teacher));
run;
```

Note: The concatenation operator || combines text. In the example above, the literal string teach is concatenated to the text that results from left-aligning the resolved value of the variable course_number.
Then, you can reference one of these variables when a course number is designated. If you designate a course number in a `%LET` statement, you can use multiple ampersands in order to create a reference to the `teach%n` macro variable that corresponds to the current course number.

```sas
%let crs=3;
proc print data=sasuser.register noobs;
  where course_number=&crs;
  var student_name paid;
  title1 "Roster for Course &crs";
  title2 "Taught by &&teach&crs";
run;
```

The SAS log shows the steps that lead to the resolution of the reference `&&teach&crs`.

**Table 10.6  SAS LOG**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>%let crs=3;</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>proc print data=sasuser.register noobs;</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>where course_number=&amp;crs;</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>var student_name paid;</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>title1 &quot;Roster for Course &amp;crs&quot;;</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>title2 &quot;Taught by &amp;&amp;teach&amp;crs&quot;;</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>run;</td>
<td></td>
</tr>
</tbody>
</table>

This is the output from the example.
Earlier you learned how to use the SYMPUT routine to create macro variables in a DATA step. You are also familiar with using a macro variable reference such as \texttt{&macvar} to obtain the value of a macro variable \textit{before} a DATA step executes. Now, suppose you want to obtain the value of a macro variable \textit{during} DATA step execution. You can obtain a macro variable’s value during DATA step execution by using the SYMGET function. The SYMGET function returns the value of an existing macro variable.

**General form, SYMGET function:**

\texttt{SYMGET(macro-variable)}

where

\textit{macro-variable}

\textbullet~ can be specified as one of the following:

\begin{itemize}
\item a macro variable name, enclosed in quotation marks
\item a DATA step variable name whose value is the name of a macro variable
\item a DATA step character expression whose value is the name of a macro variable
\end{itemize}
Example

You can use the SYMGET function to obtain the value of a different macro variable for each iteration of a DATA step. In this example, the data set variable Teacher is assigned the value of the macro variable teachn for each observation in the Sasuser.Register data set, where n is the value of the data set variable Course_number for that observation.

Note: This example assumes that a macro variable named teachn has already been created for each observation in Sasuser.Register.

```
data teachers;
  set sasuser.register;
  length Teacher $ 20;
  teacher=symget('teach'||left(course_number));
run;
```

```
proc print data=teachers;
  var student_name course_number teacher;
  title1 "Teacher for Each Registered Student";
run;
```

Part of the SAS output that this program creates is shown below. Notice that the new data set Teachers contains a variable named Teacher and that the values of this variable are the same as the values of the macro variables teach1-teach3 in the global symbol table above.
Creating Macro Variables During PROC SQL Step Execution

The INTO Clause and the NOPRINT Option

You've seen how to create macro variables during DATA step execution. You can also create or update macro variables during the execution of a PROC SQL step. Remember that the SELECT statement in a PROC SQL step retrieves and displays data. The INTO clause in a SELECT statement enables you to create or update macro variables.

When you create or update macro variables during execution of a PROC SQL step, you might not want any output to be displayed. The PRINT | NOPRINT option specifies whether a SELECT statement's results are displayed in output. PRINT is the default setting.
General form, PROC SQL with the NOPRINT option and the INTO clause:

```
PROC SQL NOPRINT;
   SELECT column1<,column2,...>
      INTO :macro-variable-1<,macro-variable-2,...>
   FROM table-1 | view-1
   <WHERE expression>
   <other clauses>;
QUIT;
```

where

- `column1, column2,...` specifies one or more columns of the SQL table specified by `table-1 | view-1`.
- `:macro-variable-1, :macro-variable-2,...` names the macro variables to create.
- `expression` produces a value that is used to subset the data.
- `other clauses` are other valid clauses that group, subset, or order the data.

**Note:** Macro variable names are preceded by a colon.△

**Note:** For more information about PROC SQL, see the SAS documentation.△

This form of the INTO clause does not trim leading or trailing blanks. Also, the INTO clause cannot be used when you create a table or a view.

**Example**

You can create a macro variable named `totalfee` that contains the total of all course fees, and use this macro variable in a later step. You use the NOPRINT option to suppress the output from the PROC SQL step.

```
proc sql noprint;
   select sum(fee) format=dollar10. into :totalfee
       from sasuser.all;
quit;
%let totalfee=&totalfee;
proc means data=sasuser.all sum maxdec=0;
   class course_title;
   var fee;
   title "Grand Total for All Courses Is &totalfee";
run;
```

**Note:** This form of the INTO clause does not trim leading or trailing blanks, but the %LET statement removes any leading or trailing blanks that are stored in the value of `totalfee`.△

The output from this PROC MEANS step shows the sum of all course fees in the DOLLAR10. format.
Creating Variables with the INTO Clause

Earlier you learned how to create a series of related macro variables during execution of the DATA step by using the SYMPUT routine. Sometimes you might want to create a series of related macro variables during execution of a PROC SQL step. You can use the INTO clause to create one new macro variable for each row in the result of the SELECT statement.

General form, SELECT statement with the INTO clause for a range of macro variables:

```sql
PROC SQL NOPRINT;
SELECT column1
   INTO :macro-variable-1 - :macro-variable-n
   FROM table-1 | view-1
   <WHERE expression>
   <other clauses>;
QUIT;
```

where

- `column1` specifies the column of the SQL table specified by `table-1 | view-1`.
- `:macro-variable-1 - :macro-variable-n` names the macro variables to create.
- `expression` produces a value that is used to subset the data.
- `other clauses` are other valid clauses that group, subset, or order the data.
Example

You can create a series of macro variables that contain the course code, location, and starting date of the first three courses that are scheduled in 2002. In this example, the macro variables `crsid1-crсид3` are assigned values of the data set variable `Course_code` from each of the first three rows of the PROC SQL result:

```
proc sql;
    select course_code, location, begin_date format=mmddyy10.
    into :crsid1-:crsid3,
    :place1-:place3,
    :date1-:date3
    from sasuser.schedule
    where year(begin_date)=2002
    order by begin_date;
quit;
```

This is the result of the PROC SQL step.

<table>
<thead>
<tr>
<th>Course Code</th>
<th>Location</th>
<th>Begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>C003</td>
<td>Dallas</td>
<td>01/07/2002</td>
</tr>
<tr>
<td>C004</td>
<td>Boston</td>
<td>01/21/2002</td>
</tr>
<tr>
<td>C005</td>
<td>Seattle</td>
<td>02/25/2002</td>
</tr>
<tr>
<td>C006</td>
<td>Dallas</td>
<td>03/25/2002</td>
</tr>
</tbody>
</table>

This is a representation of the symbol table after this PROC SQL step has run.

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRSID1 C003</td>
</tr>
<tr>
<td>CRSID2 C004</td>
</tr>
<tr>
<td>CRSID3 C005</td>
</tr>
<tr>
<td>PLACE1 Dallas</td>
</tr>
<tr>
<td>PLACE2 Boston</td>
</tr>
<tr>
<td>PLACE3 Seattle</td>
</tr>
<tr>
<td>DATE1 01/07/2002</td>
</tr>
<tr>
<td>DATE2 01/21/2002</td>
</tr>
<tr>
<td>DATE3 02/25/2002</td>
</tr>
</tbody>
</table>

If you do not know how many macro variables will be created, you can issue a query to determine how many macro variables are needed and to create a macro variable to store that number. You can then run the query, using the macro variable as the suffix of the final macro variable in each series of macro variables.
Example

Suppose you want to create ranges of macro variables that contain the course code, location, and starting date of all courses that are scheduled in 2002. You can issue a query to determine how many courses are listed in the Sasuser.Schedule data set. You can use the INTO clause to assign the resulting number to a macro variable named numrows. Then you can combine a reference to numrows with the text “crsid” in order to create the final macro variable in each series of macro variables that you create in a later SELECT statement.

Note: Remember that this form of the INTO clause does not trim leading or trailing blanks, but the %LET statement does. In the following program, the statement %let numrows=&numrows is included in order to remove leading and/or trailing blanks in the value.

```sas
proc sql noprint;
   select count(*) into :numrows
      from sasuser.schedule
      where year(begin_date)=2002;
   %let numrows=&numrows;
   %put There are &numrows courses in 2002;
   select course_code, location,
      begin_date format=mmddyy10.
      into :crsid1-:crsid&numrows,
      :place1-:place&numrows,
      :date1-:date&numrows
      from sasuser.schedule
      where year(begin_date)=2002
      order by begin_date;
   %put _user_
quit;
```

The SAS log shows that numrows is assigned a value of 4 in the first SELECT statement. The %PUT statement at the end of the program shows the names and values of all the macro variables that are created in the second SELECT statement.
### Table 10.7  SAS Log

| 20 | proc sql noprint; |
| 21 | select count(*) |
| 22 | into :numrows |
| 23 | from sasuser.schedule |
| 24 | where year(begin_date)=2002; |
| 25 | %let numrows=&numrows; |
| 26 | %put There are &numrows courses in 2002; |
| 27 | select course_code, location, |
| 28 | begin_date format=mmddyy10. |
| 29 | into :crsid1-:crsid&numrows, |
| 30 | :place1-:place&numrows, |
| 31 | :date1-:date&numrows |
| 32 | from sasuser.schedule |
| 33 | where year(begin_date)=2002 |
| 34 | order by begin_date; |
| 35 | %put _user_; |
| GLOBAL SQLOBS 4 |
| GLOBAL CRSID2 C004 |
| GLOBAL SQLOOPS 22 |
| GLOBAL CRSID3 C005 |
| GLOBAL DATE4 03/25/2002 |
| GLOBAL PLACE1 Dallas |
| GLOBAL CRSID1 C003 |
| GLOBAL PLACE2 Boston |
| GLOBAL PLACE3 Seattle |
| GLOBAL DATE1 01/07/2002 |
| GLOBAL CRSID4 C006 |
| GLOBAL TOTFEE $354,380 |
| GLOBAL DATE2 01/21/2002 |
| GLOBAL DATE3 02/25/2002 |
| GLOBAL SQLRC 0 |
| GLOBAL NUMROWS 4 |
| GLOBAL PLACE4 Dallas |

---

**Creating a Delimited List of Values**

Sometimes, during execution of a PROC SQL step, you might want to create one macro variable that will hold all values of a certain data set variable. You can use an alternate form of the INTO clause in order to take all of the values of a column (variable) and concatenate them into the value of one macro variable.
General form, SELECT statement with INTO clause for combining values into one macro variable:

```
PROC SQL NOPRINT;
   SELECT column1
       INTO :macro-variable-1
       SEPARATED BY 'delimiter1'
       FROM table-1 | view-1
       <WHERE expression>
       <other clauses>;
QUIT;
```

where

- `column1` specifies the column of the SQL table specified by `table-1 | view-1`.
- `:macro-variable-1` names the macro variable to create.
- `delimiter1` is enclosed in quotation marks and specifies the character that will be used as a delimiter in the value of the macro variable.
- `expression` produces a value that is used to subset the data.
- `other clauses` are other valid clauses that group, subset, or order the data.

**Example**

You can use the SQL procedure to create one macro variable named `sites` that contains the names of all training centers that appear in the `Sasuser.Schedule` data set. The names will be separated by blanks.

```
proc sql noprint;
   select distinct location into :sites separated by ' ' from sasuser.schedule;
quit;
```

Here is a representation of the macro variable `sites` as it is stored in the global symbol table after this PROC SQL step has run.

<table>
<thead>
<tr>
<th>Sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Dallas</td>
</tr>
<tr>
<td>Seattle</td>
</tr>
</tbody>
</table>

Now you can use the new macro variable in a title.

```
proc means data=sasuser.all sum maxdec=0;
   var fee;
   title1 'Total Revenue';
   title2 "from Course Sites: &sites";
run;
```
This is the output from the PROC MEANS step.

**Total Revenue**
*from Course Sites: Boston Dallas Seattle*

**The MEANS Procedure**

<table>
<thead>
<tr>
<th>Analysis Variable</th>
<th>Fee</th>
<th>Course Fee</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum</td>
<td>343515</td>
</tr>
</tbody>
</table>

**Working with PROC SQL Views**

When you submit a PROC SQL step, the PROC SQL program code is placed into the input stack, and word scanning is performed for macro triggers in the same process as in other SAS programs.

In the following code, the macro variable reference &crsid is resolved during the creation of the PROC SQL view, resulting in a constant value whenever the view is used. For example, if the value of crsid is C003 when this code is submitted, the view Subcrsid will always be based on the course code C003.

```
proc sql;
create view subcrsid as
  select student_name, student_company, paid
  from sasuser.all
  where course_code="&crsid";
quit;
```

A better approach would be to use the SYMGET function to enable the view to look up the macro variable value. In the following example, the view Subcrsid is based on whatever value crsid has when the view is used:

```
proc sql;
create view subcrsid as
  select student_name, student_company, paid
  from sasuser.all
  where course_code= symget('crsid');
quit;
```

%let crsid=C003;
proc print data=subcrsid noobs;
  title "Status of Students in Course Code &crsid";
run;

%let crsid=C004;
proc print data=subcrsid noobs;
  title "Status of Students in Course Code &crsid";
run;

PROC SQL does *not* perform automatic data conversion. You must use the INPUT function to convert the macro variable value to numeric if it is compared to a numeric variable.
The following code performs a query that is based on the numeric equivalent of the current value of the macro variable `crsnum`. The INPUT function is necessary in this WHERE statement because the value of the data set variable `Course_number` is numeric, but `crsnum` has a character value because it is a macro variable.

```sas
proc sql;
create view subcnum as
select student_name, student_company, paid
from sasuser.all
where course_number=input(symget('crsnum'),2.);
quit;

%let crsnum=4;
proc print data=subcnum noobs;
   title "Status of Students in Course Number &crsnum";
run;
```

### Using Macro Variables in SCL Programs

SAS Component Language (SCL) programs are placed into the input stack, and word scanning is performed for macro triggers in the same process as in other SAS programs. Macro variable references that are outside of SUBMIT blocks are resolved prior to execution. Therefore, in the following example, a constant value will be compared to the SCL variable `Wage` during SCL execution:

```sas
MAIN:
   erroroff wage;
   if wage gt &max then erroron wage;
   return;
```

Any text within a SUBMIT block is assumed to be SAS code and is therefore ignored by the SCL compiler when the SCL program is compiled. Macro variable references within SUBMIT blocks are not resolved until the SUBMIT block executes and the SAS code within the SUBMIT block is tokenized.

When a SUBMIT block executes, SAS attempts to resolve a macro variable reference (`&name`) to a corresponding SCL variable. If there is no corresponding SCL variable, the reference is passed to the macro processor for lookup in the global symbol table. You can force a reference (`&name`) within a SUBMIT block to be passed as a macro variable reference by preceding the name with two ampersands (`&&name`).

Also, there are several functions and routines that enable SCL programs and the macro facility to exchange information at execution time. Let’s take a look at these functions and routines.

You have already learned how to use the SYMPUT routine and the SYMGET function in a DATA step. Both the SYMPUT routine and the SYMGET function can be used in SCL programs. The syntax for each is exactly the same as it is in the DATA step.

Additionally, both the SYMPUT routine and the SYMGET function have numeric equivalents for use in SCL programs.

### The SYMPUTN Routine

The SYMPUTN routine enables you to create a macro variable during execution of an SCL program and to assign a numeric value to it.
General form, SYMPUTN routine:

\[
\text{CALL SYMPUTN('macro-variable', value);} \\
\]

where

- **macro-variable** is the name of a global macro variable enclosed in single quotation marks with no ampersand. Alternatively, it is the name of an SCL variable (not enclosed in quotation marks) whose value is the name of a global macro variable.

- **value** is the numeric value that is assigned to *macro-variable*, which can be a number or the name of a numeric SCL variable.

**Example**

Suppose the SCL variable `unitvar` has a value of `unit` and the SCL variable `unitnum` has a numeric value of 200. To create a macro variable whose name is the value of `unitvar` (in this case, `unit`) and assign a value equal to the value of the SCL variable `unitnum` (in this case, 200) you submit the following statement within a SUBMIT block:

```
call symputn(unitvar, unitnum);
```

Similarly, to create a macro variable named `unitvar` and assign a numeric value of 500 to it, you submit the following statement within a SUBMIT block.

```
call symputn('unitvar', 500);
```

**The SYMGETN Function**

The SYMGETN function enables you to obtain the numeric value of a macro variable during execution of an SCL program.

General form, SYMGETN function:

\[
SCL\text{-variable} = \text{SYMGETN('macro-variable')} ; \\
\]

where

- **SCL-variable** is the name of a numeric SCL variable to which the value of *macro-variable* is assigned.

- **macro-variable** is the name of a global macro variable enclosed in single quotation marks with no ampersand. Alternatively, it is the name of an SCL variable (not enclosed in quotation marks) whose value is the name of a global macro variable.
Example

Suppose the SCL variable \texttt{unitvar} has a value of \texttt{unit}, the macro variable \texttt{unit} has a value of \texttt{200}, and the macro variable \texttt{unitvar} has a value of \texttt{500}. The first statement below creates an SCL variable named \texttt{unitnum} and assigns to it a value of \texttt{200}. The second statement creates an SCL variable named \texttt{unit} and assigns it a value of \texttt{500}.

\begin{verbatim}
unitnum=symgetn(unitvar);
unit=symgetn('unitvar');
\end{verbatim}

\textit{Note:} For more information about using macro variables in SCL, see the SAS documentation for the macro language.
Summary

This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Creating a Macro Variable During DATA Step Execution

When you create or update a macro variable with the %LET statement, all macro processing takes place before the execution of the DATA step. The SYMPUT routine enables you to create or update macro variables during DATA step execution. Depending on how the arguments are coded, you can create either a single macro variable or multiple macro variables. You can use the SYMPUT routine with literal strings to create a macro variable and to assign either an exact name or an exact text value to it. You can use the SYMPUT routine with a DATA step variable to assign the value of that DATA step variable to a macro variable.

You can use the SYMPUTX routine to create or update a macro variable during DATA step execution, and to automatically strip leading and trailing blanks from the macro variable name and value. You can also use a DATA step expression as an argument to the SYMPUT routine in order to apply DATA step functions to a value before you assign that value to a macro variable. The PUT function is often useful in conjunction with the SYMPUT and SYMPUTX routines.

Creating Multiple Macro Variables During DATA Step Execution

You can use the SYMPUT or SYMPUTX routine with two DATA step expressions to create a series of related macro variables within one DATA step.

Referencing Macro Variables Indirectly

Sometimes, it is useful to use indirect references to macro variables. For example, you might want to use a macro variable to construct the name of another macro variable. You can reference a macro variable indirectly by preceding the macro variable name with two or more ampersands.

Obtaining Macro Variable Values During DATA Step Execution

The SYMGET function is used by both the DATA step and the SQL procedure to obtain the value of a macro variable during execution. You can use the SYMGET function to assign a macro variable value to a data step variable.

Creating Macro Variables During PROC SQL Step Execution

You can access the macro facility in a PROC SQL step by using the INTO clause in the SELECT statement. Various forms of the INTO clause enable you to create a series of macro variables, a varying number of macro variables, or a single macro variable that records a value that is created by concatenating the unique values of an SQL variable. You can use the NOPRINT option to prevent a PROC SQL step from creating output.
Working with PROC SQL Views

When you submit a PROC SQL step, the PROC SQL program code is placed into the input stack, and word scanning is performed for macro triggers in the same process as in other SAS programs.

Using Macro Variables in SCL Programs

SAS Component Language (SCL) also has routines and functions that assign values to macro variables and that obtain values from a macro symbol table. The SYMPUT routine and the SYMGET function can be used in an SCL program in the same way that they can be used in a DATA step program. Also, the SYMPUTN routine can be used to create macro variables and to assign numeric values to those variables during the execution of an SCL program. The SYMGETN function can be used to obtain the numeric value of a macro variable during the execution of an SCL program.

Syntax

CALL SYMPUT(macro-variable,text);

PUT(source,format.)

CALL SYMPUT(expression1,expression2);

CALL SYMPUTN('macro-variable', value);

CALL SYMPUTX(macro-variable,text);

SYMGET(macro-variable)

SYMGETN('macro-variable');

PROC SQL NOPRINT;
  SELECT column1,<column2,...>
    INTO :macro-variable-1<,:macro-variable-2,...>
    FROM table-1 | view-1
    <WHERE expression>
    <other clauses>;
QUIT;

PROC SQL NOPRINT;
  SELECT column1
    INTO :macro-variable-1 - :macro-variable-n
    FROM table-1 | view-1
    <WHERE expression>
    <other clauses>;
QUIT;
PROC SQL NOPRINT;
SELECT column1
    INTO :macro-variable-1
    SEPARATED BY 'delimiter1'
FROM table-1 | view-1
WHERE expression
{others clauses};
QUIT;

Sample Programs

Using CALL SYMPUT to Create Macro Variables

```sas
options symbolgen pagesize=30;
%let crsnum=3;
data revenue;
    set sasuser.all end=final;
    where course_number=&crsnum;
    total+1;
    if paid='Y' then paidup+1;
    if final then do;
        if paidup<total then do;
            call symput('foot','Some Fees Are Unpaid');
        end;
        else do;
            call symput('foot','All Students Have Paid');
        end;
    end;
run;
proc print data=revenue;
    var student_name student_company paid;
    title "Payment Status for Course &crsnum";
    footnote "$foot";
run;
```

Referencing Macro Variables Indirectly

```sas
options symbolgen;
data _null_; 
    set sasuser.courses;
    call symput(course_code, trim(course_title));
run;
%let crsid=C005;
proc print data=sasuser.schedule noobs label;
    where course_code="&crsid";
    var location begin_date teacher;
    title1 "Schedule for &crsid";
run;
%let crsid=C002;
```
proc print data=sasuser.schedule noobs label;
  where course_code="&crsid";
  var location begin_date teacher;
  title1 "Schedule for &crsid";
run;

Using SYMGET to Obtain Macro Variable Values

data teachers;
  set sasuser.register;
  length Teacher $ 20;
  teacher=symget('teach'||left(course_number));
run;

proc print data=teachers;
  var student_name course_number teacher;
  title1 "Teacher for Each Registered Student";
run;

Creating Macro Variables with the INTO Clause

proc sql noprint;
  select course_code, location, begin_date format=mmddyy10.
    into :crsid1-:crsid3,
        :place1-:place3,
        :date1-:date3
  from sasuser.schedule
  where year(begin_date)=2002
    order by begin_date;
quit;

Points to Remember

- The SYMPUT routine can be used to create or update macro variables during DATA step execution.
- The values of macro variables are always character values. In the DATA step, SYMPUT performs automatic numeric to character conversion on any numeric value that you attempt to assign to a macro variable.
- The SYMGET function can be used to obtain the value of a macro variable during the execution of a DATA step, a PROC SQL step, or an SCL program.
- The INTO clause can be used in the SELECT statement to create or update macro variables during execution of a PROC SQL step.
- The SYMPUT and SYMPUTN routines can be used to create or update macro variables during the execution of an SCL program.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.
1 Which of the following is false?
   a A %LET statement causes the macro processor to create a macro variable before the program is compiled.
   b To create a macro variable that is based on data calculated by the DATA step, you use the SYMPUT function.
   c Macro functions are always processed during the execution of the DATA step.
   d Macro variable references in a DATA step are always resolved prior to DATA step execution.

2 Which of the following correctly creates a macro variable named region and assigns to it a value that is based on the value of the data set variable Location?
   a data new;
      set sasuser.all;
      if location='Boston' then do;
         call symput('region', 'East');
      end;
      else do;
         call symput('region', 'West');
      end;
   run;
   b data new;
      set sasuser.all;
      if location='Boston' then do;
         %let region=East;
      end;
      else
         %let region=West;
      end;
   run;
   c data new;
      set sasuser.all;
      if location='Boston' then do;
         call symput(region, "East");
      end;
      else
         call symput(region, "West");
      end;
   run;
   d data new;
      set sasuser.all;
      if location='Boston' then do;
         symput(region, East);
      end;
      else
         symput(region, West);
      end;
   run;
3. The SYMPUT routine cannot
   a. be used to assign a data set variable as a value to a macro variable.
   b. create a series of macro variables in one DATA step.
   c. automatically convert a numeric value to a character value when used to
      assign a value to a macro variable in a DATA step.
   d. be used to assign a numeric value to a macro variable in an SCL program.

4. Which of the following programs correctly creates a series of macro variables
   whose names are values of the data set variable `course_code`, then indirectly
   references one of those macro variables in a later step?

   a. data _null_
      set sasuser.courses;
      call symput(course_code, trim(course_title));
      %let crsid=C005;
      proc print data=sasuser.schedule noobs label;
      where course_code="&crsid";
      var location begin_date teacher;
      title1 "Schedule for &c005";
      run;

   b. data _null_
      set sasuser.courses;
      call symput(course_code, trim(course_title));
      %let crsid=C005;
      proc print data=sasuser.schedule noobs label;
      where course_code="&crsid";
      var location begin_date teacher;
      title1 "Schedule for &&&crsid";
      run;

   c. data _null_
      set sasuser.courses;
      call symput('course_code', trim(course_title));
      %let crsid=C005;
      proc print data=sasuser.schedule noobs label;
      where course_code="&crsid";
      var location begin_date teacher;
      title1 "Schedule for &&&crsid";
      run;

   d. data _null_
      set sasuser.courses;
      call symget(course_code, trim(course_title));
      %let crsid=C005;
      proc print data=sasuser.schedule noobs label;
      where course_code="&crsid";
      var location begin_date teacher;
      title1 "Schedule for &&&crsid";
      run;
5. Which of the following statements about the resolution of macro variable references is false?
   a. Two ampersands resolve to one ampersand.
   b. If more than four consecutive ampersands precede a name token, the macro processor generates an error message.
   c. Re-scanning continues until there are no remaining macro triggers that the macro processor can resolve.
   d. The macro processor always re-scans a name token that is preceded by multiple ampersands or by multiple percent signs.

6. In which of the following situations would you use SYMGET rather than a macro variable reference (&macvar)?
   a. to create a DATA step variable from a macro variable value during the execution of the DATA step
   b. to include a macro variable reference in a PROC SQL view
   c. to access the value of a macro variable during the execution of an SCL program
   d. all of the above

7. Which of the following correctly creates a macro variable in a PROC SQL step?
   a. call symput(daily_fee, put(fee/days, dollar8.));
   b. %let daily_fee=put(fee/days, dollar8.)
   c. select fee/days format=dollar8.
      into :daily_fee from sasuser.all;
   d. select fee/days format=dollar8.
      into daily_fee from sasuser.all;

8. According to the global symbol table shown here, what value will a reference to &&teach&crs resolve to?
   a. &TEACH3
   b. TEACH3
   c. Forest, Mr. Peter
   d. none of the above

9. Which of the following statements correctly creates a DATA step variable named Price and assigns to it the value of the macro variable daily_fee during DATA step execution?
   a. price=daily_fee;
   b. price=symget(daily_fee);
   c. price=symget(&daily_fee);
   d. price=symget("daily_fee");
10 Which of the following is false?

a. The SYMPUT routine can be used to create a macro variable during execution of the DATA step or during execution of an SCL program.

b. In the DATA step, the SYMPUT routine automatically converts to a character value any numeric value that you attempt to assign as the value of a macro variable.

c. PROC SQL automatically converts to a numeric value any macro variable value that you attempt to compare to a numeric value.

d. In an SCL program, the SYMPUTN routine can be used to assign a numeric value to a macro variable.
Example 393
The MPRINTNEST Option 395
Example 395
The MLOGICNEST Option 396
Example 396
Processing Statements Conditionally 397
Conditional Execution 397
%IF-%THEN Compared to IF-THEN 398
Example 399
Example 400
Macro Execution with Conditional Processing 401
Example 401
Example 402
Conditional Processing of Parts of Statements 403
Example 403
Case Sensitivity in Macro Comparisons 405
Example 405
Processing Statements Iteratively 406
Example 407
Example 408
Generating Complete Steps 409
Example 409
Using Arithmetic and Logical Expressions 411
The %EVAL Function 411
Examples 411
Example 413
Automatic Evaluation 413
Summary 414
Text Summary 414
Basic Concepts 414
Developing and Debugging Macros 414
Using Macro Parameters 414
Understanding Symbol Tables 414
Processing Statements Conditionally 415
Processing Statements Iteratively 415
Using Arithmetic and Logical Expressions 415
Syntax 415
Sample Programs 416
Defining a Basic Macro 416
Defining a Macro with Positional Parameters 416
Defining a Macro with Keyword Parameters 417
Defining a Macro with Mixed Parameters 417
Using the %IF-%THEN Statement 417
Using the Iterative %DO Statement 417
Points to Remember 418
Quiz 418
Overview

Introduction
Like macro variables, macro programs (also known as macros) enable you to substitute text into your SAS programs. Macros are different from macro variables because they can use conditional logic to make decisions about the text that you substitute into your programs. Using macros can help make your SAS programs more dynamic and reusable.

For example, suppose you submit a SAS program every day to create registration listings for courses that are to be held later in the current month. Then, suppose that every Friday you also submit a SAS program to create a summary of revenue that has been generated so far in the current month. By using a macro, you can automate the process so that only one SAS program is required. This program will always submit the daily report and will conditionally submit the weekly report if it is Friday. Furthermore, you could create and store a macro that would automate this process, and the only code you would need to submit each day is this:

%reports

Objectives
In this chapter, you learn to

- define and call simple macros
- describe the basic actions that the macro processor performs during macro compilation and execution
- use system options for macro debugging
- interpret error messages and warning messages that the macro processor generates
- define and call macros that include parameters
- describe the difference between positional parameters and keyword parameters
- explain the difference between the global symbol table and local symbol tables
- describe how the macro processor determines which symbol table to use
- describe the concept of nested macros and the hierarchy of symbol tables
- conditionally process code within a macro program
- iteratively process code within a macro program.

Prerequisites
Before beginning this chapter, you should complete the following chapters:

- Chapter 9, “Introducing Macro Variables,” on page 287
- Chapter 10, “Processing Macro Variables at Execution Time,” on page 325.

Basic Concepts

Defining a Macro
In order to create a macro program, you must first define it. You begin a macro definition with a %MACRO statement, and you end the definition with a %MEND statement.
General form, `%MACRO` statement and `%MEND` statement:

```sas
%MACRO macro-name;
    text
%MEND <macro-name>;
```

where

- `macro-name` names the macro. The value of `macro-name` can be any valid SAS name that is not a reserved word in the SAS macro facility.

- `text` can be:
  - constant text, possibly including SAS data set names, SAS variable names, or SAS statements
  - macro variables, macro functions, or macro program statements
  - any combination of the above.

Tip: You might want to include `macro-name` in the `%MEND` statement in order to make your program more readable. However, the inclusion of `macro-name` in the `%MEND` statement is entirely optional.

### Example

This program creates a macro named `Prtlast` that will print the most recently created data set. (Remember that the automatic macro variable `SYSLAST` stores the name of the most recently created data set.)

```sas
%macro prtlast;
    proc print data=&syslast (obs=5);
    title "Listing of &syslast data set";
    run;
%mend;
```

### Compiling a Macro

In order to use this macro later in your SAS programs, you must first compile it by submitting the macro definition, as follows:

```sas
%macro prtlast;
    proc print data=&syslast (obs=5);
    title "Listing of &syslast data set";
    run;
%mend;
```

When you submit this code, the word scanner divides the macro into tokens and sends the tokens to the macro processor for compilation. The macro processor

- checks all macro language statements for syntax errors (non-macro language statements are not checked until the macro is executed).
- writes error messages to the SAS log and creates a dummy (non-executable) macro if any syntax errors are found in the macro language statements.
stores all compiled macro language statements and constant text in a SAS catalog entry if no syntax errors are found in the macro language statements. By default, a catalog named Work.Sasmacr is opened, and a catalog entry named Macro-nameMacro is created.

That is, if there are no syntax errors in the macro language statements within the macro, the text between the %MACRO statement and the %MEND statement will be stored under the name Prtlast for execution at a later time.

Note: You can also store a compiled macro in a permanent SAS catalog. You can learn how to do this in Chapter 12, “Storing Macro Programs,” on page 423.

---

**The MCOMPILENOTE Option**

The MCOMPILENOTE option is available beginning in SAS 9. MCOMPILENOTE will cause a note to be issued to the SAS log when a macro has completed compilation.

<table>
<thead>
<tr>
<th>General form, MCOMPILENOTE option:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIONS MCOMPILENOTE= NONE</td>
</tr>
<tr>
<td>where the option can take one of the three values listed and</td>
</tr>
<tr>
<td>NONE</td>
</tr>
<tr>
<td>is the default value, which specifies that no notes are issued to the log.</td>
</tr>
<tr>
<td>NOAUTOCALL</td>
</tr>
<tr>
<td>specifies that a note is issued to the log for completed macro compilations for all macros except autocall macros.</td>
</tr>
<tr>
<td>ALL</td>
</tr>
<tr>
<td>specifies that a note is issued to the log for all completed macro compilations.</td>
</tr>
</tbody>
</table>

Note: You can learn more about autocall macros in Chapter 12, “Storing Macro Programs,” on page 423.

---

Example

A macro might actually compile and still contain errors. If there are any errors, an ERROR message will be written to the SAS log in addition to the note. Here is an example of the note that is written to the log when a macro compiles without errors:

```sas
options mcompilenote=all;
%macro mymacro;
%mend mymacro;
```
Calling a Macro

After the macro is successfully compiled, you can use it in your SAS programs for the duration of your SAS session without resubmitting the macro definition. Just as you must reference macro variables in order to access them in your code, you must call a macro program in order to execute it within your SAS program.

A macro call
- is specified by placing a percent sign (%) before the name of the macro
- can be made anywhere in a program except within the data lines of a DATALINES statement (similar to a macro variable reference)
- requires no semicolon because it is not a SAS statement.

To execute the macro Prtlast you would call the macro as follows:

%prtlast

CAUTION:
A semicolon after a macro call might insert an inappropriate semicolon into the resulting program, leading to errors during compilation or execution. Δ

Macros come in three types, depending on how they are called: name style, command style, and statement style. Of the three, name style is the most efficient. This is because calls to name style macros always begin with a percent sign (%), which immediately tells the word scanner to pass the token to the macro processor. With the other two types, the word scanner does not know immediately whether the token should be sent to the macro processor or not. Therefore, time is wasted while the word scanner determines this. All of the macros in this chapter are name style macros.

Example

Suppose a SAS program consists of several program steps that create SAS data sets. Suppose that after each of these program steps you want to print out the data set that has been created. Remember that the macro Prtlast prints the most recently created data set. If Prtlast has been compiled, you can call it after each step in order to print each data set.

```sas
proc sort data=sasuser.courses out=courses;
   by course_code;
run;

%prtlast

proc sort data=sasuser.schedule out=schedule;
   by begin_date;
run;

%prtlast
```

Table 11.1  SAS Log

| 1 | options mcompilenote=all; |
| 2 | %macro mymacro;            |
| 3 | %mend mymacro;             |

NOTE: The macro MYMACRO completed compilation without errors.
Creating and Using Macro Programs

Example 377

```sas
proc sort data=sasuser.students out=students;
    by student_name;
run;
%prtlast
```

Note: The example above is simply meant to show you how you can incorporate a macro into your SAS program. Although this is a valid use of the Prtlast macro, this might not be the best way to code this example. Since the Prtlast macro uses no conditional logic or macro programming statements and it makes no decisions, this example does not illustrate the full power of a macro program. In the rest of this chapter, you will see examples of macro programs that are more useful than this one.

Macro Execution

When you call a macro in your SAS program, the word scanner passes the macro call to the macro processor, because the percent sign that precedes the macro name is a macro trigger. When the macro processor receives `%macro-name`, it

1. searches the designated SAS catalog (Work.Sasmacr by default) for an entry named `Macro-name.Macro`
2. executes compiled macro language statements within `Macro-name`
3. sends any remaining text in `Macro-name` to the input stack for word scanning
4. suspends macro execution when the SAS compiler receives a global SAS statement or when it encounters a SAS step boundary
5. resumes execution of macro language statements after the SAS code executes.

Later in this chapter you will see detailed examples of macro execution. These examples will make more sense once you have learned how to write a more complex macro program than you have seen so far in this chapter.

For now, remember that the macro call is processed by the macro processor before any SAS language statements such as DATA steps are compiled or executed. During macro execution, the macro processor can communicate directly with

- both global and local symbol tables. For example, the macro processor can store macro variable values with a `%LET` statement and can resolve macro variable references.
- the input stack. For example, the macro processor can generate SAS code for tokenization by the word scanner.

During macro execution, the macro processor cannot communicate directly with SAS data sets in order to obtain or modify the values of DATA step variables.

Note: You will learn more about global and local symbol tables later in this chapter.

Example

Let’s look at an example of macro execution. Assume that the Prtlast macro has been compiled and that it has been stored in the Work.Sasmacr catalog.

1. First, you submit the macro call, as follows:
   ```sas
   %prtlast
   ```

2. When the word scanner encounters this call, it passes the call to the macro processor. The macro processor searches for the compiled macro in the catalog entry `Work.Sasmacr.Prtlast.Macro`. 
Catalog Entry

%macro prtlast;
  proc print data=&syslast(obs=5);
  title "Listing of &syslast data set";
  run;
%mend;

3 The macro processor begins executing compiled macro language statements. However, in this example, no compiled macro statements are included in the macro.

4 The macro processor places noncompiled items (SAS language statements) on the input stack, and pauses as the word scanner tokenizes the inserted text. In this example, the macro processor will place the PROC PRINT step on the input stack.

Input Stack

proc print data=&syslast(obs=5);
  title "Listing of &syslast data set";
run;

5 The word scanner passes these tokens to the compiler. When the word scanner encounters a macro variable reference such as &syslast, it passes the reference to the macro processor for resolution. The macro processor returns the macro variable value to the input stack and word scanning continues.

6 After all of the statements in the PROC PRINT step have been compiled, the PROC PRINT step is executed, and SAS creates output that includes only the first five observations of the most recently created data set.

7 Once the PROC PRINT step has been executed, the macro processor resumes execution of any remaining macro language statements in the macro (there are none in this example). The macro processor ends execution when it reaches the %MEND statement.

Assume that the most recently created data set is Work.Practice (which is a copy of Sasuser.Courses). Here is the output that is generated by calling the Prtlast macro.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Course_Code</th>
<th>Course_Title</th>
<th>Days</th>
<th>Fee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C001</td>
<td>Basic Telecommunications</td>
<td>3</td>
<td>$795</td>
</tr>
<tr>
<td>2</td>
<td>C002</td>
<td>Structured Query Language</td>
<td>4</td>
<td>$1150</td>
</tr>
<tr>
<td>3</td>
<td>C003</td>
<td>Local Area Networks</td>
<td>3</td>
<td>$650</td>
</tr>
<tr>
<td>4</td>
<td>C004</td>
<td>Database Design</td>
<td>2</td>
<td>$375</td>
</tr>
<tr>
<td>5</td>
<td>C005</td>
<td>Artificial Intelligence</td>
<td>2</td>
<td>$400</td>
</tr>
</tbody>
</table>

Here is an example of messages that are written to the SAS log when %prtlast is submitted, assuming that the most recently created data set is Work.Practice.
Table 11.2 SAS Log

```
37 %prtlast

NOTE: Writing HTML Body file: sashtm3.htm
NOTE: There were 5 observations read from the data set WORK.PRACTICE.
NOTE: PROCEDURE PRINT used (Total process time):
    real time 0.04 seconds
    cpu time 0.03 seconds
```

Notice that in this SAS log message, you see a note from PROC PRINT, but not the PROC PRINT code itself since the call to the macro does not display the text that is sent to the compiler.

This example depicts the processing and execution of a simple macro. Later in this chapter you will see how more-complex macros that include compiled macro language statements are handled by the macro processor.

Developing and Debugging Macros

### Monitoring Execution with System Options

In the last example, you saw that when you call a macro, the text that is sent to the compiler does not appear in the SAS log. But sometimes you might want to see this text. There are SAS system options that can display information about the execution of a macro in the SAS log. This can be especially helpful for debugging purposes.

### The MPRINT Option

When the MPRINT option is specified, the text that is sent to the SAS compiler as a result of macro execution is printed in the SAS log.

General form, MPRINT | NOMPRINT option:

```
OPTIONS MPRINT | NOMPRINT;
```

where

- **NOMPRINT**
  - is the default setting, and specifies that the text that is sent to the compiler when a macro executes is not printed in the SAS log.
- **MPRINT**
  - specifies that the text that is sent to the compiler when a macro executes is printed in the SAS log.

You might want to specify the MPRINT system option if
- you have a SAS syntax error or execution error
- you want to see the generated SAS code.
Example

Suppose you want to call the `Prtlast` macro and to use the MPRINT system option to show the SAS code that results from the macro execution.

Catalog Entry

```sas
%macro prtlast;
   proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
   run;
%end;
```

The following sample code creates a data set named `Sales`, specifies the MPRINT option, then references the `Prtlast` macro:

```sas
data sales;
   price_code=1;
run;
options mprint;
%prtlast
```

The messages that are written to the SAS log show the text that is sent to the compiler. Notice that the macro variable reference (`&SYSLAST`) is resolved to the value `Work.Sales` in the MPRINT messages that are written to the SAS log.

Table 11.3  SAS Log

<table>
<thead>
<tr>
<th>101</th>
<th>%prtlast</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPRINT(PRTLAST): proc print data=WORK.SALES (obs=5);</td>
<td></td>
</tr>
<tr>
<td>MPRINT(PRTLAST): title &quot;Listing of WORK.SALES&quot;;</td>
<td></td>
</tr>
<tr>
<td>MPRINT(PRTLAST): run;</td>
<td></td>
</tr>
<tr>
<td>NOTE: There were 1 observations read from the dataset WORK.SALES.</td>
<td></td>
</tr>
<tr>
<td>NOTE: PROCEDURE PRINT used:</td>
<td></td>
</tr>
<tr>
<td>real time</td>
<td>0.04 seconds</td>
</tr>
<tr>
<td>cpu time</td>
<td>0.04 seconds</td>
</tr>
</tbody>
</table>

The MLOGIC Option

Another system option that might be useful when you debug your programs is the MLOGIC option. The MLOGIC option prints messages that indicate macro actions that were taken during macro execution.

General form, MLOGIC | NOMLOGIC option:

```sas
OPTIONS MLOGIC | NOMLOGIC;
```

where

NOMLOGIC

is the default setting, and specifies that messages about macro actions are not printed to the SAS log during macro execution.

MLOGIC

specifies that messages about macro actions are printed to the log during macro execution.
When the MLOGIC system option is in effect, the information that is displayed in SAS log messages includes
- the beginning of macro execution
- the results of arithmetic and logical macro operations
- the end of macro execution.

Example

Suppose you want to repeat the previous example with only the MLOGIC system option in effect. This sample code creates a data set named Sales, sets the MLOGIC system option, then calls the Prtlast macro.

```sas
data sales;
    price_code=1;
run;
options nomprint mlogic;
%prtlast
```

When this code is submitted, the messages that are written to the SAS log show the beginning and the end of macro processing.

<table>
<thead>
<tr>
<th>107</th>
<th>%prtlast</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLOGIC(PRTLAST): Beginning execution.</td>
<td></td>
</tr>
<tr>
<td>NOTE: There were 1 observations read from the dataset WORK.SALES.</td>
<td></td>
</tr>
<tr>
<td>NOTE: PROCEDURE PRINT used:</td>
<td></td>
</tr>
<tr>
<td>real time 0.02 seconds</td>
<td></td>
</tr>
<tr>
<td>cpu time 0.02 seconds</td>
<td></td>
</tr>
<tr>
<td>MLOGIC(PRTLAST): Ending execution.</td>
<td></td>
</tr>
</tbody>
</table>

The MLOGIC option, along with the MPRINT option and the SYMBOLGEN option, is typically turned
- on for development and debugging purposes
- off when the application is in production mode.

Comments in Macro Programs

As with any other programs, your macro programs might benefit from comments. Comments can be especially helpful if you plan to save your macros permanently or to share them with other users. You can place comments within a macro definition by using the macro comment statement.

General form, macro comment statement:

```sas
%*comment;
```

where

```sas
comment
```

can be any message. Like other SAS statements, each macro comment statement ends with a semicolon.
Example

The following code uses macro comments to describe the functionality of the macro:

```
%macro printit;
    /* The value of &syslast will be substituted appropriately; */
    /* as long as a data set has been created during this session. */
    proc print data=&syslast(obs=5);
    /* Print only the first 5 observations */
    title "Last Created Data Set Is &syslast";
    run;
%mend;
```

Note: You can also use the comment symbols /* and */ inside a macro. When these symbols appear, the macro processor ignores the text within the comment.

Using Macro Parameters

You have seen the basic form for a macro definition. Your macros will often contain macro variables. To make your macros more dynamic, you could use the %LET statement to update the values of the macro variables that are used within the macros. However, parameter lists in your macro definitions enable you to update the macro variables within your macro programs more conveniently. A parameter list is an optional part of the %MACRO statement that names one or more macro variables whose values you specify when you call the macro.

Example

Suppose the compiled macro Printdsn contains references to the macro variables dsn (which records a data set name) and vars (which records a list of data set variables), as follows:

```
%macro printdsn;
    proc print data=&dsn;
    var &vars;
    title "Listing of %upcase(&dsn) data set";
    run;
%mend;
```

You could modify the behavior of Printdsn by changing the value of macro variable dsn or vars with a %LET statement before you call the macro. For example, you could substitute sasuser.courses for dsn and course_code course_title days for vars at macro execution, as follows:

```
%let dsn=sasuser.courses;
%let vars=course_code course_title days;
%printdsn
```

If the MPRINT system option is turned on when this code is submitted, the following messages are written to the SAS log. Notice that the values that you provided in the %LET statements have been substituted into the macro when it appears in the SAS log.
Then you could submit new %LET statements in order to change the value of `dsn` to `sasuser.schedule` and to change the value of `vars` to `course_code location begin_date` when the macro executes, as follows:

```sas
%let dsn=sasuser.schedule;
%let vars=course_code location begin_date;
%printdsn
```

The messages that are written to the SAS log when this code is submitted show that the new values have been substituted for the macro variable references in the macro.

You can make these macro variables easier to update by using parameters in the macro definition to create the macro variables. Then you can pass values to the macro variables each time you call the macro rather than using separate %LET statements. The next few sections will show you how to use various types of parameters to create macro variables.

### Macros That Include Positional Parameters

When you include positional parameters in a macro definition, a macro variable is automatically created for each parameter when you call the macro. To define macros that include positional parameters, you list the names of macro variables in the `%MACRO` statement of the macro definition. Positional parameters are so named because the order in which you specify them in a macro definition determines the order in which they are assigned values from the macro call. That is, when you call a macro
that includes positional parameters, you specify the values of the macro variables that are defined in the parameters in the same order in which they are defined.

General form, macro definition that includes positional parameters:

```plaintext
%MACRO macro-name(parameter-1<,...,parameter-n>);
  text
%MEND <macro-name>;
```

where

- `parameter-1<,...,parameter-n>`
  - specifies one or more positional parameters, separated by commas. You must supply each parameter with a name: you cannot use a text expression to generate it.

To call a macro that includes positional parameters, precede the name of the macro with a percent sign, and enclose the parameter values in parentheses. List the values in the same order in which the parameters are listed in the macro definition, and separate them with commas, as follows:

```plaintext
%macro-name(value-1<,...,value-n>)
```

The values listed in a macro call

- can be null values, text, macro variable references, or macro calls
- are assigned to the parameter variables using a one-to-one correspondence.

**Example**

You can use positional parameters to create the macro variables `dsn` and `vars` in the `Printdsn` macro definition, as follows:

```plaintext
%macro printdsn(dsn,vars);
  proc print data=&dsn;
    var &vars;
    title "Listing of %upcase(&dsn) data set";
  run;
%mend;
```

In this case, when you call the `Printdsn` macro you assign values to the macro variables that are created in the parameters. In the following example, the value `sasuser.courses` is assigned to the macro variable `dsn`, and the value `course_code course_title days` is assigned to the macro variable `vars`. Notice that the value for `dsn` is listed first and the value for `vars` is listed second, since this is the order in which they are listed in the macro definition.

```plaintext
%printdsn(sasuser.courses,course_code course_title days)
```

*Note:* To substitute a null value for one or more positional parameters, use commas as placeholders for the omitted values, as follows:

```plaintext
%printdsn(,course_code course_title days)
```
Macros That Include Keyword Parameters

You can also include *keyword parameters* in a macro definition. Like positional parameters, keyword parameters create macro variables. However, when you use keyword parameters to create macro variables, you list both the name and the value of each macro variable in the macro definition.

Keyword parameters can be listed in any order. Whatever value you assign to each parameter (or variable) in the `%MACRO` statement becomes its default value. Null values are allowed.

General form, macro definition that includes keyword parameters:

```sas
%MACRO macro-name(keyword-1=<value-1>,...,keyword-n=<value-n>);
    text
%MEND <macro-name>;
```

where

- `keyword-1=<value-1>,...,keyword-n=<value-n>` names one or more macro parameters followed by equal signs. Optionally, you can specify default values after the equal signs. If you omit a default value, the keyword parameter has a null value.

When you *call* a macro whose definition includes keyword parameters, you specify both the keyword and the value for each parameter, in any order. If you omit a keyword parameter from the macro call, the keyword variable retains its default value, as follows:

```sas
%macro-name(keyword-1=value-1<,...,keyword-n=value-n>)
```

Example

You can use keyword parameters to create the macro variables `dsn` and `vars` in the `Printdsn` macro. This example assigns a default value of `sasuser.courses` to the macro variable `dsn` and assigns a default value of `course_code course_title days` to the macro variable `vars`:

```sas
%macro printdsn(dsn=sasuser.courses,vars=course_code course_title days);
    proc print data=&dsn;
    var &vars;
    title "Listing of %upcase(&dsn) data set";
    run;
%mend;
```

To invoke the `Printdsn` macro with a value of `sasuser.schedule` for `dsn` and a value of `teacher course_title begin_date` for `vars`, you would issue the following call:

```sas
%printdsn(dsn=sasuser.schedule, vars=teacher course_code begin_date)
```

To call the `Printdsn` macro with default values for the parameters (`sasuser.courses` as the value for `dsn` and `course_code course_title days` as the value for `vars`), you could issue the following call:

```sas
%printdsn()
```
Note: To call the macro `Printdsn` with default values for the parameters, you could also issue a macro call that specified these values explicitly, as follows:

```
%printdsn(dsn=sasuser.courses,vars=course_code
course_title days)
```

 Macros That Include Mixed Parameter Lists

You can also include a parameter list that contains both positional and keyword parameters in your macro definitions. All positional parameter variables in the `%MACRO` statement must be listed before any keyword parameter variable is listed.

General form, macro definition that includes mixed parameters:

```
%MACRO macro-name(parameter-1<,...,parameter-n>,
  keyword-1=value-1<,...,keyword-n=value-n>);
  text
%MEND;
```

where

```
parameter-1<,...,parameter-n>
  is listed before keyword-1=value-1<,...,keyword-n=value-n>.
```

Similarly, when you call a macro that includes a mixed parameter list, you must list the positional values before any keyword values, as follows:

```
%macro-name(value-1<,...,value-n>,
  keyword-1=value-1<,...,keyword-n=value-n>)
```

Example

You can use a combination of positional and keyword parameters to create the macro variables in the `Printdsn` macro definition. This code uses a positional parameter to create the macro variable `dsn`, and a keyword parameter to create the macro variable `vars`:

```
%macro printdsn(dsn, vars=course_title course_code days);
  proc print data=&dsn;
    var &vars;
    title "Listing of %upcase(&dsn) data set";
  run;
%mend;
```

The following call to the `Printdsn` macro assigns the value `sasuser.schedule` to the macro variable `dsn` and assigns the value `teacher location begin_date` to the macro variable `vars`. Notice that the value for `dsn` is listed first, since `dsn` is the positional parameter.

```
%printdsn(sasuser.schedule, vars=teacher location begin_date)
```
Now, suppose you want to execute the `Printdsn` macro, assigning the default value `course_title course_code days` to the macro variable `vars` and assigning the value `sasuser.courses` to the macro variable `dsn`. You could issue the following call:

```
%printdsn(sasuser.courses)
```

Because this call omits the keyword parameter (`vars`), the default value for that parameter is used.

**Macros That Include the PARMBUFF Option**

You can use the PARMBUFF option in a macro definition to create a macro that can accept a *varying number of parameters* at each invocation. The PARMBUFF option assigns the entire list of parameter values in a macro call, including the parentheses in a name-style invocation, as the value of the automatic macro variable `SYSPBUFF`.

<table>
<thead>
<tr>
<th>General form, macro definition with the PARMBUFF option:</th>
</tr>
</thead>
<tbody>
<tr>
<td>`%%MACRO macro-name /PARMBUFF;</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>%MEND;</td>
</tr>
<tr>
<td>where</td>
</tr>
<tr>
<td>text</td>
</tr>
<tr>
<td>contains a reference to the automatic macro variable <code>SYSPBUFF</code>.</td>
</tr>
</tbody>
</table>

**Example**

The following macro definition creates a macro named `Printz`. `Printz` uses a varying number of parameters and the automatic macro variable `SYSPBUFF` to display the parameters that are specified in the macro call. The macro also uses a conditional loop to print the data sets that are named as parameters, and the `%EVAL` function to increment the macro variable `num`. You will learn more about the `%EVAL` function later in this chapter.

```
%macro printz /parmbuff;
   %put Syspbuff contains: &syspbuff;
   %let num=1;
   %let dsname=%scan(&syspbuff,&num);
   %do %while(&dsname ne);
      proc print data=sasuser.&dsname;
      run;
   %let num=%eval(&num+1);
   %let dsname=%scan(&syspbuff,&num);
   %end;
%mend printz;
```

If you submit a call to the macro that includes two parameters, the `Printz` macro writes the following line to the SAS log and causes two data sets to be printed:

```
%printz(courses, schedule)
```
Table 11.7 SAS Log

| Syspbuff contains: (courses,schedule) |

If you submit a call to the macro that includes one parameter, the Printz macro writes the following line to the SAS log and causes one data set to be printed:

```sas
%printz(courses)
```

Table 11.8 SAS Log

| Syspbuff contains: (courses) |

*Note:* If the macro definition includes both a set of parameters and the PARMBUFF option, the macro invocation causes the parameters to receive values and the entire invocation list of values to be assigned to `SYSPBUFF`. △

---

### Understanding Symbol Tables

#### The Global Symbol Table

You are already somewhat familiar with the *global symbol table*. Remember that automatic macro variables are stored in the global symbol table. User-defined macro variables that you create with a `%LET` statement in open code (code that is outside of a macro definition) are also stored in the global symbol table.

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSDATE</td>
</tr>
<tr>
<td>SYSDAY</td>
</tr>
<tr>
<td>SYSVER</td>
</tr>
<tr>
<td>uservar1</td>
</tr>
<tr>
<td>uservar2</td>
</tr>
</tbody>
</table>

The global symbol table is created during the initialization of a SAS session and is deleted at the end of the session. Macro variables in the global symbol table

- are available anytime during the session
- can be created by a user
- have values that can be changed during the session (except for some automatic macro variables).

You can create a global macro variable with

- a `%LET` statement (used outside a macro definition)
- a DATA step that contains a SYMPUT routine
- a DATA step that contains a SYMPUTX routine (beginning in SAS 9)
- a SELECT statement that contains an INTO clause in PROC SQL
The %GLOBAL Statement

The %GLOBAL statement

- creates one or more macro variables in the global symbol table and assigns null values to them
- can be used either inside or outside a macro definition
- has no effect on variables that are already in the global symbol table.

General form, %GLOBAL statement:

```
%GLOBAL macro-variable-1 <...macro-variable-n>;
```

where

- `macro-variable`
  - is either the name of a macro variable or a text expression that generates a macro variable name.

Example

To create a global macro variable inside a macro definition, you can use the the %GLOBAL statement. The %GLOBAL statement in the following example creates two global macro variables, `dsn` and `vars`. The %LET statements assign values to the new global macro variables, as follows:

```
%macro printdsn;
  %global dsn vars;
  %let dsn=sasuser.courses;
  %let vars=course_title course_code days;
  proc print data=&dsn;
    var &vars;
    title "Listing of &dsn data set";
    run;
  %mend;

%printdsn
```

Note: You use the %SYMDEL statement to delete a macro variable from the global symbol table during a SAS session. To remove the macro variable `dsn` from the global symbol table, you submit the following statement:

```
%symdel dsn;
```

The Local Symbol Table

A local symbol table is created when a macro that includes a parameter list is called or when a request is made to create a local variable during macro execution. The local
symbol table is deleted when the macro finishes execution. That is, the local symbol table exists only while the macro executes.

<table>
<thead>
<tr>
<th>Local Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter1</td>
</tr>
<tr>
<td>parameter2</td>
</tr>
<tr>
<td>uservar1</td>
</tr>
<tr>
<td>uservar2</td>
</tr>
</tbody>
</table>

The local symbol table contains macro variables that can be
☐ created and initialized at macro invocation (that is, by parameters)
☐ created or updated during macro execution
☐ referenced anywhere within the macro.

You can create local macro variables with
☐ parameters in a macro definition
☐ a %LET statement within a macro definition
☐ a DATA step that contains a SYMPUT routine within a macro definition
☐ a DATA step that contains a SYMPUTX routine within a macro definition (beginning in SAS 9)
☐ a SELECT statement that contains an INTO clause in PROC SQL within a macro definition
☐ a %LOCAL statement.

Note: The SYMPUT routine and the SYMPUTX routine can only create a local macro variable if a local symbol table already exists. If no local symbol table exists when the SYMPUT routine or SYMPUTX routine executes, it will create a global macro variable. Δ

You have already learned about using parameters in macro definitions. You should also already be familiar with the %LET statement, the SYMPUT routine, and the INTO clause. Let’s take a closer look at using the %LOCAL statement.

---

The %LOCAL Statement

The %LOCAL statement
☐ can appear only inside a macro definition
☐ creates one or more macro variables in the local symbol table and assigns null values to them
☐ has no effect on variables that are already in the local symbol table.

A local symbol table is not created until a request is made to create a local variable. Macros that do not create local variables do not have a local table. Remember, the SYMPUT routine and the SYMPUTX routine can create local variables only if the local table already exists.

Since local symbol tables exist separately from the global symbol table, it is possible to have a local macro variable and a global macro variable that have the same name and different values.
Creating and Using Macro Programs  △  Rules for Creating and Updating Variables  391

Example

In this example, the first `%LET` statement creates a global macro variable named `dsn` and assigns a value of `sasuser.courses` to it.

The `%LOCAL` statement within the macro definition creates a local macro variable named `dsn`, and the `%LET` statement within the macro definition assigns a value of `sasuser.register` to the local variable `dsn`.

The `%PUT` statement within the macro definition will write the value of the local variable `dsn` to the SAS log, whereas the `%PUT` statement that follows the macro definition will write the value of the global variable `dsn` to the SAS log:

```sas
%let dsn=sasuser.courses;

%macro printdsn;
  %local dsn;
  %let dsn=sasuser.register;
  %put The value of DSN inside Printdsn is &dsn;
%mend;

%printdsn
%put The value of DSN outside Printdsn is &dsn;
```

When you submit this code, the following statements are written to the SAS log.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>199</td>
<td>%let dsn=sasuser.courses;</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>%macro printdsn;</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>%local dsn;</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>%let dsn=sasuser.register;</td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>%put The value of DSN inside Printdsn is &amp;dsn;</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>%mend;</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>%printdsn</td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>%put The value of DSN outside Printdsn is &amp;dsn;</td>
<td></td>
</tr>
</tbody>
</table>

Table 11.9  SAS Log

The value of DSN inside Printdsn is sasuser.register
The value of DSN outside Printdsn is sasuser.courses

Rules for Creating and Updating Variables

When the macro processor receives a request to create or update a macro variable during macro execution, the macro processor follows certain rules. Let’s look at those rules.

Suppose the macro processor receives a `%LET` statement during a macro call, as follows:

```sas
%let macvar=value;
```

The macro processor will take the following steps:

1. The macro processor checks to see whether the macro variable `macvar` already exists in the local symbol table. If so, the macro processor updates `macvar` in the local symbol table with the value `value`. If `macvar` does not exist in the local table, the macro processor goes on to step 2.

2. The macro processor checks to see whether the macro variable `macvar` already exists in the global symbol table. If so, the macro processor updates `macvar` in the
global symbol table with the value `value`. If `macvar` does not exist in the global symbol table, the macro processor goes on to step 3.

3 The macro processor creates a macro variable named `macvar` in the local symbol table and assigns a value of `value` to it.

```
<table>
<thead>
<tr>
<th>Does <code>macvar</code> already exist in the local symbol table?</th>
<th>Yes</th>
<th>Update <code>macvar</code> in the local symbol table with the value <code>value</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does <code>macvar</code> already exist in the global symbol table?</td>
<td>Yes</td>
<td>Update <code>macvar</code> in the global symbol table with the value <code>value</code>.</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Create <code>macvar</code> in the local symbol table and assign a value of <code>value</code> to it.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Similarly, suppose the macro processor receives the following reference during a macro call:

```
&macvar
```

The macro processor will take the following steps:

1 The macro processor checks to see whether the macro variable `macvar` exists in the local symbol table. If so, the macro processor retrieves the value of `macvar` from the local symbol table. If `macvar` does not exist in the local table, the macro processor goes on to step 2.

2 The macro processor checks to see whether the macro variable `macvar` exists in the global symbol table. If so, the macro processor retrieves the value of `macvar` from the global symbol table. If `macvar` does not exist in the global symbol table, the macro processor goes on to step 3.

3 The macro processor returns the tokens to the word scanner. A warning message is written to the SAS log to indicate that the reference was not resolved.

```
<table>
<thead>
<tr>
<th>Does <code>macvar</code> exist in the local symbol table?</th>
<th>Yes</th>
<th>Retrieve the value of <code>macvar</code> from the local symbol table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does <code>macvar</code> exist in the global symbol table?</td>
<td>Yes</td>
<td>Retrieve the value of <code>macvar</code> from the global symbol table.</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Return the tokens to the word scanner. Issue a warning message to the SAS log to indicate that the reference was not resolved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*Note:* Remember that if the macro processor receives either a `%LET` statement or a macro variable reference (`&macvar`) in open code, it will check only the *global* symbol table for existence of the macro variable. If a macro program is not currently executing, a local symbol table does not currently exist. □
Multiple Local Symbol Tables

Multiple local symbol tables can exist concurrently during macro execution if you have nested macros. That is, if you define a macro program that calls another macro program, and if both macros create local symbol tables, then two local symbol tables will exist while the second macro executes.

Example

Suppose the following two macros, Outer and Inner, have been compiled. The macro named Outer creates a local macro variable named variX and assigns a value of one to it. Then Outer calls another macro program named Inner. The macro named Inner creates a local macro variable named variY and assigns the value of variX to it.

```sas
%macro outer;
  %local variX;
  %let variX=one;
  %inner
%mend outer;

%macro inner;
  %local variY;
  %let variY=&variX;
%mend inner;
```

Let’s examine what happens to the symbol tables when you submit the following code:

```sas
%let variX=zero;
%outer
```

1 The macro processor receives $let variX=zero;$. It checks the global symbol table for a macro variable named variX. There is none, so the macro processor creates variX and assigns a value of zero to it.

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>variX</td>
</tr>
<tr>
<td>zero</td>
</tr>
</tbody>
</table>

2 The macro processor receives $outer$. The macro processor retrieves the macro Outer from Work.Sasmacr, then begins executing it.

3 The macro processor encounters $local variY;$. It creates a local symbol table. The macro processor creates the macro variable variX in this local table and assigns a null value to it. This does not affect the macro variable variX that is stored in the global symbol table.

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
<th>Outer Local Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>variX</td>
<td>variX</td>
</tr>
<tr>
<td>zero</td>
<td></td>
</tr>
</tbody>
</table>

4 The macro processor encounters $let variX=one;$. The macro processor checks the local symbol table for variX and assigns a value of one to it.
5 The macro processor receives %inner. It retrieves the macro Inner from Work.Sasmacr, then begins executing it.

6 The macro processor encounters %local variY;. It creates a local symbol table. The macro processor creates a macro variable variY in this table and assigns a null value to it. There are now two local symbol tables in existence.

7 The macro processor encounters %let variY=&variX;. It checks the most recently created local table for variX. There is no such macro variable in that symbol table, so the macro processor then checks the other local symbol table. It retrieves the value one from that symbol table and substitutes the value into the %LET statement. Then the macro processor checks the most recently created local symbol table for a macro variable named variY. When it finds this macro variable, it assigns the value one to it.

8 The Inner macro finishes executing, and the local symbol table that was created within this macro is deleted. There is now only one local symbol table in existence.

9 The Outer macro finishes executing, and the local symbol table that was created within this macro is deleted. There are now no local symbol tables in existence. The global symbol table has not been changed since variX was created and was assigned a value of zero.

As you can see, each macro program in the example above has its own local symbol table that exists as long as the macro executes. When a macro finishes executing, its local symbol table and all of the local macro variables that are contained in that table are erased. The global symbol table and all of the global macro variables that are contained in it remain.
The MPRINTNEST Option

The MPRINTNEST option is available beginning in SAS 9. MPRINTNEST allows the macro nesting information to be written to the SAS log in the MPRINT output. This has no effect on the MPRINT output that is sent to an external file.

General form, MPRINTNEST option:

    OPTIONS MPRINTNEST | NOMPRINTNEST;

where

- MPRINTNEST specifies that macro nesting information is written in the MPRINT output in the SAS log.
- NOMPRINTNEST specifies that macro nesting information is not written in the MPRINT output in the SAS log.

The setting of the MPRINTNEST option does not imply the setting of MPRINT. You must set both MPRINT and MPRINTNEST in order for output with the nesting information to be written to the SAS log.

Example

Suppose that you have defined three nested macros, as follows:

```sas
%macro outer;
    data _null_;
    %inner
    run;
%mend outer;

%macro inner;
    put %inrmost;
%mend inner;

%macro inrmost;
    'This is the text of the PUT statement'
%mend inrmost;
```

The SAS log below shows the messages that are written when you set both the MPRINT and MPRINTNEST options and submit a call to the Outer macro, as follows:

```sas
options mprint mprintnest;
%outer
```
The MLOGICNEST Option

The MLOGICNEST option is available beginning in SAS 9. MLOGICNEST allows the macro nesting information to be displayed in the MLOGIC output in the SAS log. The setting of MLOGICNEST does not affect the output of any currently executing macro.

General form, MLOGICNEST option:

```
OPTIONS MLOGICNEST | NOMLOGICNEST;
```

where

- **MLOGICNEST** specifies that macro nesting information is written in the MLOGIC output in the SAS log.
- **NOMLOGICNEST** specifies that macro nesting information is not written in the MLOGIC output in the SAS log.

The setting of MLOGICNEST does not imply the setting of MLOGIC. You must set both MLOGIC and MLOGICNEST in order for output with nesting information to be written to the SAS log.

Example

Suppose that you have defined three nested macros, as follows:

```
%macro outer;
  %put THIS IS OUTER;
  %inner
%mend outer;

%macro inner;
  %put THIS IS INNER;
  %inrmost
%mend inner;

%macro inrmost;
  %put THIS IS INRMOST;
%mend inrmost;
```

The SAS log below shows the messages that are written when you set both the MLOGIC and MLOGICNEST options and submit a call to the *Outer* macro, as follows:
options mlogic mlogicnest;
%outer

Table 11.11 SAS Log

| MLOGIC(OUTER): Beginning execution.         |
| MLOGIC(OUTER): %PUT THIS IS OUTER          |
| THIS IS OUTER                              |
| MLOGIC(OUTER.INNER): Beginning execution.  |
| MLOGIC(OUTER.INNER): %PUT THIS IS INNER    |
| THIS IS INNER                              |
| MLOGIC(OUTER.INNER.INRMOST): Beginning execution. |
| MLOGIC(OUTER.INNER.INRMOST): %PUT THIS IS INRMOST |
| THIS IS INRMOST                            |
| MLOGIC(OUTER.INNER.INRMOST): Ending execution. |
| MLOGIC(OUTER.INNER): Ending execution.     |
| MLOGIC(OUTER): Ending execution.           |

Processing Statements Conditionally

Conditional Execution

You can use macros to control conditional execution of statements. Remember the example from the beginning of this chapter where you wanted to run a daily report to create registration listings for courses to be held later in the month. Remember that you also wanted to run a weekly report each Friday to create a summary of revenue that has been generated so far in the current month. You can accomplish these tasks with one program if you use conditional execution to determine whether the second report should be run.

You can perform conditional execution at the macro level with %IF-%THEN and %ELSE statements.

General form, %IF-%THEN and %ELSE statements:

```
%IF expression %THEN text;
<%ELSE text;>
```

where

- **expression**
  - can be any valid macro expression that resolves to an integer.

- **text**
  - can be specified as
    - constant text
    - a text expression
    - a macro variable reference, a macro call, or a macro program statement.

If **expression** resolves to zero, then it is false and the %THEN text is not processed (the optional %ELSE text is processed instead). If it resolves to any integer other than
zero, then the expression is true and the %THEN text is processed. If it resolves to null or to any noninteger value, an error message is issued.

The %ELSE statement is optional. However, the macro language does not contain a subsetting %IF statement. Thus, you cannot use %IF without %THEN.

**%IF-%THEN Compared to IF-THEN**

Although they look similar, the %IF-%THEN/%ELSE statement and the IF-THEN/ELSE statement belong to two different languages. Most of the same rules that apply to the DATA step IF-THEN/ELSE statement also apply to the %IF-%THEN/%ELSE statement. However, there are several important differences between the macro %IF-%THEN statement and the DATA step IF-THEN statement.

<table>
<thead>
<tr>
<th>%IF-%THEN...</th>
<th>IF-THEN...</th>
</tr>
</thead>
<tbody>
<tr>
<td>is used only in a macro program.</td>
<td>is used only in a DATA step program.</td>
</tr>
<tr>
<td>executes during macro execution.</td>
<td>executes during DATA step execution.</td>
</tr>
<tr>
<td>uses macro variables in logical expressions and cannot refer to DATA step variables in logical expressions.</td>
<td>uses DATA step variables in logical expressions.</td>
</tr>
<tr>
<td>determines what text should be copied to the input stack.</td>
<td>determines what DATA step statement(s) should be executed. When inside a macro definition, it is copied to the input stack as text.</td>
</tr>
</tbody>
</table>

Simple %DO and %END statements often appear in conjunction with %IF-%THEN/%ELSE statements in order to designate a section of the macro to be processed depending on whether the %IF condition is true or false. Use %DO and %END statements following %THEN or %ELSE in order to conditionally place text that contains multiple statements onto the input stack. Each %DO statement must be paired with an %END statement.

**General form, %DO-%END with %IF-%THEN and %ELSE statements:**

```
%IF expression %THEN %DO;
    text and/or macro language statements
%END;
%ELSE %DO;
    text and/or macro language statements
%END;
```

where

```
text and/or macro language statements
```

is either constant text, a text expression, and/or a macro statement.

**Note:** The statements %IF-%THEN, %ELSE, %DO, and %END are macro language statements that can be used only inside a macro program. △
Example

You can control text that is copied to the input stack with the %IF-%THEN while controlling DATA step logic with IF-THEN. In this example, the value of the macro variable status determines which variables will be included in the new data set. The value of the data set variable Location determines the value of the new data set variable Totalfee.

```sas
%macro choice(status);
  data fees;
  set sasuser.all;
  %if &status=PAID %then %do;
    where paid='Y';
    keep student_name course_code begin_date totalfee;
  %end;
  %else %do;
    where paid='N';
    keep student_name course_code begin_date totalfee latechg;
    latechg=fee*.10;
  %end;
  /* add local surcharge */
  if location='Boston' then totalfee=fee*1.06;
  else if location='Seattle' then totalfee=fee*1.025;
  else if location='Dallas' then totalfee=fee*1.05;
  run;
%mend choice;
```

If the MPRINT and MLOGIC system options are both set, the SAS log will display messages showing the text that is sent to the compiler. For example, suppose you submit the following macro call:

```sas
options mprint mlogic;
%choice(PAID)
```

The following messages will be written to the log. Notice that the MLOGIC option shows the evaluation of the expression in the %IF statement, but it does not show the evaluation of the expression in the IF statement.

Table 11.12  SAS Log

```
160  %choice(PAID)
    MLOGIC(CHOICE): Beginning execution.
    MLOGIC(CHOICE): Parameter STATUS has value PAID
    MPRINT(CHOICE): data fees;
    MPRINT(CHOICE): set sasuser.all;
    MLOGIC(CHOICE): %IF condition &status=PAID is TRUE
    MPRINT(CHOICE): where paid='Y';
    MPRINT(CHOICE): keep student_name course_code begin_date totalfee;
    MPRINT(CHOICE): if location='Boston' then totalfee=fee*1.06;
    MPRINT(CHOICE): else if location='Seattle' then totalfee=fee*1.025;
    MPRINT(CHOICE): else if location='Dallas' then totalfee=fee*1.05;
    MPRINT(CHOICE): run;
```

Suppose you submit the following macro call:

```sas
options mprint mlogic;
%choice(OWED)
```
The following messages will be sent to the SAS log. Notice that the text that is written to the input stack is different this time.

**Table 11.13** SAS Log

<table>
<thead>
<tr>
<th>Line</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>161</td>
<td>%choice(OWED)</td>
</tr>
<tr>
<td>162</td>
<td>MLOGIC(CHOICE): Beginning execution.</td>
</tr>
<tr>
<td>163</td>
<td>MLOGIC(CHOICE): Parameter STATUS has value OWED</td>
</tr>
<tr>
<td>164</td>
<td>MPRINT(CHOICE): data fees;</td>
</tr>
<tr>
<td>165</td>
<td>MPRINT(CHOICE): set sasuser.all;</td>
</tr>
<tr>
<td>166</td>
<td>MPRINT(CHOICE): %IF condition &amp;status=PAID is FALSE</td>
</tr>
<tr>
<td>167</td>
<td>MPRINT(CHOICE): where paid='N';</td>
</tr>
<tr>
<td>168</td>
<td>MPRINT(CHOICE): keep student_name course_code begin_date totalfee</td>
</tr>
<tr>
<td></td>
<td>latechg;</td>
</tr>
<tr>
<td>169</td>
<td>MPRINT(CHOICE): latechg=fee*.10;</td>
</tr>
<tr>
<td>170</td>
<td>MPRINT(CHOICE): if location='Boston' then totalfee=fee*1.06;</td>
</tr>
<tr>
<td>171</td>
<td>MPRINT(CHOICE): else if location='Seattle' then totalfee=fee*1.025;</td>
</tr>
<tr>
<td>172</td>
<td>MPRINT(CHOICE): else if location='Dallas' then totalfee=fee*1.05;</td>
</tr>
<tr>
<td>173</td>
<td>MPRINT(CHOICE): run;</td>
</tr>
</tbody>
</table>

Earlier you learned the process that occurs when a macro program is compiled. Now that you have seen more-complex macro programs, let’s look at this process again.

Remember that during macro compilation, macro statements are checked for syntax errors. If a macro definition contains macro statement syntax errors, error messages are written to the SAS log, and a non-executable (dummy) macro is created.

**Example**

Suppose you attempt to compile a macro that contains a syntax error. For example, the following program is missing a percent sign in the %IF-%THEN statement:

```sas
%macro printit;
  %if &syslast ne _NULL_ then %do;
    proc print data=_last_(obs=5);
    title "Last Created Data Set Is &syslast";
    run;
  %end;
%mend;
```

When you submit this macro definition, the macro processor checks the %IF-%THEN statement and the %DO and %END statements for syntax errors. Since there is a syntax error in the %IF-%THEN statement, the following error messages are written to the SAS log.
Macro Execution with Conditional Processing

Now let’s take another look at the execution of a macro. Earlier you learned that when the macro processor receives `%macro-name`, it executes compiled macro language statements such as `%IF-%THEN`. The values of macro variables that are used within the `%IF` logical expression are resolved during macro execution. The `%IF` logical expression is automatically evaluated.

**Example**

Suppose the Printit macro has been compiled and has been stored in the Work.Sasmacr catalog.

1. First, you submit a call to the Printit macro, as follows:

   ```sas
   %printit
   ```


   ```sas
   %macro printit;
   %if &syslast ne _NULL_ %then %do;
   proc print data=_last_(obs=5);
   title "Last Created Data Set Is &syslast";
   run;
   %end;
   %mend;
   ```

3. The macro processor begins to execute compiled macro language statements from Printit (that is, the `%IF-%THEN` statement). Because the `%IF` expression is true, the `%DO` block is processed.

4. The macro processor places the text that follows the `%DO` statement (that is, the PROC PRINT step) on the input stack.

   ```sas
   proc print data=_last_(obs=5);
   title "Last Created Data Set Is &syslast";
   run;
   ```

5. Word scanning proceeds as usual on the PROC PRINT step. When a macro trigger such as `&syslast` is encountered, the macro reference is passed to the macro.
processor for resolution. The macro processor returns resolved values to the input stack.

6 After the word scanner sends all of the tokens from the PROC PRINT step to the compiler, and the RUN statement is encountered, the PROC PRINT step executes.

7 Macro execution pauses while the PROC PRINT step executes, and macro execution stops when the %MEND statement is encountered.

It is possible to conditionally insert individual statements into the input stack, even in the middle of a step.

Example

Suppose you want to generate a report of enrollment at each training center as listed in the data set Sasuser.All. You can specify your macro program so that if a specific course is requested, the macro will insert a WHERE ALSO statement in order to restrict the report to that course. This example also customizes the second title line based on whether or not a course was selected, as follows:

```sas
%macro attend(crs,start=01jan2001,stop=31dec2001);
   %let start=%upcase(&start);
   %let stop=%upcase(&stop);
   proc freq data=sasuser.all;
      where begin_date between "&start"d and "&stop"d;
      table location / nocum;
      title "Enrollment from &start to &stop";
      %if &crs= %then %do;
         title2 "for all Courses";
      %end;
      %else %do;
         title2 "for Course &crs only";
         where also course_code="&crs";
      %end;
   run;
%mend;

Note: In the program above, the %IF statement %if &crs= is true when crs has a value of null. △

Suppose you submit the following call, which specifies a specific course:

%attend(C003)

This call results in the following output. Notice that the second title has been written according to the %ELSE %DO statement in the macro.
Creating and Using Macro Programs

Enrollment from 01JAN2001 to 31DEC2001
for Course C003 only

The FREQ Procedure

<table>
<thead>
<tr>
<th>Location</th>
<th>Frequency</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>20</td>
<td>40.00</td>
</tr>
<tr>
<td>Seattle</td>
<td>30</td>
<td>60.00</td>
</tr>
</tbody>
</table>

Now suppose you submit the following call, which specifies a start date but does not specify a course:

```
%attend(start=01jul2001)
```

This call results in the following output. Notice that in this output, the second title line is written according to the %IF-%THEN statement in the macro.

Enrollment from 01JUL2001 to 31DEC2001
for all Courses

The FREQ Procedure

<table>
<thead>
<tr>
<th>Location</th>
<th>Frequency</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>56</td>
<td>34.57</td>
</tr>
<tr>
<td>Dallas</td>
<td>23</td>
<td>14.20</td>
</tr>
<tr>
<td>Seattle</td>
<td>63</td>
<td>51.23</td>
</tr>
</tbody>
</table>

Conditional Processing of Parts of Statements

The text that is processed as the result of conditional logic can be a small part of a SAS statement. This makes it possible to conditionally insert text into the middle of a statement.

Example

Suppose you want to print a table of frequency counts from a SAS data set. You can generate either a one-way table or a two-way table, based on the value of a macro parameter. This example creates a one-way table if only the `cols` parameter is specified in the call. It creates a two-way table if the `rows` parameter is also specified.

```
%macro counts (cols=_all_,rows=,dsn=&syslast);
    title "Frequency Counts for %upcase(&dsn) data set";
```
Suppose you submit the following call, which specifies both `cols` and `rows`:

```sas
%counts(dsn=sasuser.all, cols=paid, rows=course_number)
```

Part of the resulting output from this call is shown below. Notice that the macro has created a two-way table.

### Frequency Counts for SASUSER.ALL data set

#### The FREQ Procedure

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Percent</th>
<th>Row Pct</th>
<th>Col Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course_Number (Course Number)</td>
<td>N</td>
<td>Y</td>
<td>Total</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>1.38</td>
<td>3.92</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26.09</td>
<td>73.91</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.61</td>
<td>5.20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>1.84</td>
<td>3.69</td>
<td></td>
</tr>
<tr>
<td></td>
<td>33.33</td>
<td>66.67</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.48</td>
<td>4.89</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>1.38</td>
<td>3.23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30.00</td>
<td>70.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.61</td>
<td>4.25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>1.61</td>
<td>4.61</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25.93</td>
<td>74.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.54</td>
<td>6.12</td>
<td></td>
</tr>
</tbody>
</table>

Now suppose you submit the following call, which specifies `cols` but does not specify `rows`:

```sas
%counts(dsn=sasuser.all, cols=paid)
```

The output that results from this call is shown below. Notice that this time the macro has created a one-way table.
Case Sensitivity in Macro Comparisons

Remember that comparisons that are made in %IF expressions are case sensitive.

Example

If you construct your %IF statement using incorrect case in any program text, the statement will never be true. For example, the %IF statement below will always be false because _null_ is specified in lowercase but is always stored in SAS in uppercase:

```
%macro prtlast;
  %if &syslast=_null_ %then %do;
    %put No data sets created yet.;
  %end;
  %else %do;
    proc print;
    title "Last Created Data Set is &syslast";
    run;
  %end;
%mend;
```

Suppose SYSLAST has a value of _NULL_ when you submit this example. The following messages are written to the SAS log.
Tip: The %UPCASE function is often useful when you construct %IF statements. For more information about the %UPCASE function, see Chapter 9, “Introducing Macro Variables,” on page 287.

### Processing Statements Iteratively

Many macro applications require iterative processing. With the iterative %DO statement you can repeatedly

- execute macro programming code
- generate SAS code.
General form, iterative %DO statement with %END statement:

```
%DO index-variable=start %TO stop <%BY increment>;
   text
%END;
```

where

- `index-variable` is either the name of a macro variable or a text expression that generates a macro variable name.
- `start` and `stop` specify either integers or macro expressions that generate integers to control how many times the portion of the macro between the iterative %DO and %END statements is processed.
- `increment` specifies either an integer (other than 0) or a macro expression that generates an integer to be added to the value of the index variable in each iteration of the loop. By default, `increment` is 1.
- `text` can be:
  - constant text, possibly including SAS data set names, SAS variable names, or SAS statements
  - macro variables, macro functions, or macro program statements
  - any combination of the above.

%DO and %END statements are valid only inside a macro definition. The `index-variable` is created in the local symbol table if it does not appear in any existing symbol table.

The iterative %DO statement evaluates the value of the index variable at the beginning of each loop. The loop stops processing when the index variable has a value that is one increment beyond the range of the `start` and `stop` values.

---

**Example**

You can use a macro loop to create and display a series of macro variables.

This example creates a series of macro variables named `teach1-teachn`, one for each observation in the `Sasuser.Schedule` data set, and assigns teacher names to them as values. Then the `Putloop` macro uses a %DO statement and a %END statement to create a loop that writes these macro variables and their values to the SAS log, as follows:

```sas
data _null_;  
   set sasuser.schedule end=no_more;  
   call symput('teach'||left(_n_), (trim(teacher)));  
   if no_more then call symput('count',_n_);  
run;

%macro putloop;  
   %local i;  
   %do i=1 %to &count;  
      %put TEACH%i is &&teach%i;
```

```
Tip: It is a good idea to specifically declare the index variable of a macro loop as a local variable to avoid the possibility of accidentally changing the value of a macro variable that has the same name in other symbol tables.

When the Putloop macro is executed, no code is sent to the compiler, because the %PUT statements are executed by the macro processor. The following messages are written to the SAS log.

<table>
<thead>
<tr>
<th>Table 11.16</th>
<th>SAS Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEACH1 is Hallis, Dr. George</td>
<td></td>
</tr>
<tr>
<td>TEACH2 is Wickam, Dr. Alice</td>
<td></td>
</tr>
<tr>
<td>TEACH3 is Forest, Mr. Peter</td>
<td></td>
</tr>
<tr>
<td>TEACH4 is Tally, Ms. Julia</td>
<td></td>
</tr>
<tr>
<td>TEACH5 is Hallis, Dr. George</td>
<td></td>
</tr>
<tr>
<td>TEACH6 is Berthan, Ms. Judy</td>
<td></td>
</tr>
<tr>
<td>TEACH7 is Hallis, Dr. George</td>
<td></td>
</tr>
<tr>
<td>TEACH8 is Wickam, Dr. Alice</td>
<td></td>
</tr>
<tr>
<td>TEACH9 is Forest, Mr. Peter</td>
<td></td>
</tr>
<tr>
<td>TEACH10 is Tally, Ms. Julia</td>
<td></td>
</tr>
<tr>
<td>TEACH11 is Tally, Ms. Julia</td>
<td></td>
</tr>
<tr>
<td>TEACH12 is Berthan, Ms. Judy</td>
<td></td>
</tr>
<tr>
<td>TEACH13 is Hallis, Dr. George</td>
<td></td>
</tr>
<tr>
<td>TEACH14 is Wickam, Dr. Alice</td>
<td></td>
</tr>
<tr>
<td>TEACH15 is Forest, Mr. Peter</td>
<td></td>
</tr>
<tr>
<td>TEACH16 is Tally, Ms. Julia</td>
<td></td>
</tr>
<tr>
<td>TEACH17 is Hallis, Dr. George</td>
<td></td>
</tr>
<tr>
<td>TEACH18 is Berthan, Ms. Judy</td>
<td></td>
</tr>
</tbody>
</table>

You can also use a macro loop to generate statements that can be placed inside a SAS program step.

Example

The following macro generates a series of statements within a DATA step. On each iteration, the macro writes a message to the SAS log that puts the current value of the index variable into HEX6. format.

```sas
%macro hex(start=1, stop=10, incr=1);
   %local i;
   data _null_;
   %do i=&start %to &stop %by &incr;
      value=&i;
      put "Hexadecimal form of &i is " value hex6.;
   %end;
   run;
%mend hex;
```

Suppose you submit the following call:

```sas
options mprint mlogic symbolgen;
%hex(start=20,stop=30,incr=2)
```

Note: The HEX6. format converts a number to hexadecimal format. △
Some of the messages that are written to the SAS log when *Hex* executes are shown below. Notice that according to the MLOGIC messages, the loop stops processing when the value of the index variable is 32 (which is one increment beyond the value that is specified for *Stop*).

Table 11.17 SAS Log

| MLOGIC(HEX): %DO loop index variable I is now 30; loop will iterate again. |
|-------------|------------------|
| SYMBOLGEN: Macro variable I resolves to 30 |
| MPRINT(HEX): value=30; |
| SYMBOLGEN: Macro variable I resolves to 30 |
| MPRINT(HEX): put "Hexadecimal form of 30 is " value hex6.; |
| MLOGIC(HEX): %DO loop index variable I is now 32; loop will not iterate again. |
| MPRINT(HEX): run; |

Hexadecimal form of 20 is 000014
Hexadecimal form of 22 is 000016
Hexadecimal form of 24 is 000018
Hexadecimal form of 26 is 00001A
Hexadecimal form of 28 is 00001C
Hexadecimal form of 30 is 00001E

NOTE: DATA statement used:
- real time 0.06 seconds
- cpu time 0.06 seconds

MLOGIC(HEX): Ending execution.

Generating Complete Steps

You can use the iterative %DO statement to build macro loops that create complete SAS steps.

Example

Suppose course offerings for several years are stored in a series of external files that are named by year, such as *Raw1999.dat* and *Raw2000.dat*. All the files have the same record layout. Suppose you want to read each file into a separate SAS data set.

The following macro uses a %DO statement to create a loop that creates a data set from each of the specified external files:

```sas
%macro readraw(first=1999,last=2005);
  %local year;
  %do year=&first %to &last;
    data year&year;
      infile "raw&year..dat";
      input course_code $4.
        location $15.
      begin_date date9.
      teacher $25.;
    run;
    proc print data=year&year;
      title "Scheduled classes for &year";
    format begin_date date9.;
  %end;
%mend readraw;
```

```sas
%readraw(first=1999,last=2005);
```
Suppose you submit the following call to the Readraw macro:

```
%readraw(first=2000,last=2002)
```

The macro creates three data sets named Year2000, Year2001, and Year2002. The output that SAS creates is shown below. Remember that in order for this program to run properly, the raw data files must be named appropriately, and they must be stored in the location that the program specifies.

### Scheduled classes for 2000

<table>
<thead>
<tr>
<th>Obs</th>
<th>course_code</th>
<th>location</th>
<th>begin_date</th>
<th>teacher</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C001</td>
<td>Seattle</td>
<td>23OCT2000</td>
<td>Hallis, Dr. George</td>
</tr>
<tr>
<td>2</td>
<td>C002</td>
<td>Dallas</td>
<td>04DEC2000</td>
<td>Wickam, Dr. Alice</td>
</tr>
</tbody>
</table>

### Scheduled classes for 2001

<table>
<thead>
<tr>
<th>Obs</th>
<th>course_code</th>
<th>location</th>
<th>begin_date</th>
<th>teacher</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C003</td>
<td>Boston</td>
<td>08JAN2001</td>
<td>Forest, Mr. Peter</td>
</tr>
<tr>
<td>2</td>
<td>C004</td>
<td>Seattle</td>
<td>22JAN2001</td>
<td>Tally, Ms. Julia</td>
</tr>
<tr>
<td>3</td>
<td>C005</td>
<td>Dallas</td>
<td>26FEB2001</td>
<td>Hallis, Dr. George</td>
</tr>
<tr>
<td>4</td>
<td>C006</td>
<td>Boston</td>
<td>02APR2001</td>
<td>Berthan, Ms. Judy</td>
</tr>
<tr>
<td>5</td>
<td>C001</td>
<td>Dallas</td>
<td>21MAY2001</td>
<td>Hallis, Dr. George</td>
</tr>
<tr>
<td>6</td>
<td>C002</td>
<td>Boston</td>
<td>11JUN2001</td>
<td>Wickam, Dr. Alice</td>
</tr>
<tr>
<td>7</td>
<td>C003</td>
<td>Seattle</td>
<td>16JUL2001</td>
<td>Forest, Mr. Peter</td>
</tr>
<tr>
<td>8</td>
<td>C004</td>
<td>Dallas</td>
<td>13AUG2001</td>
<td>Tally, Ms. Julia</td>
</tr>
<tr>
<td>9</td>
<td>C005</td>
<td>Boston</td>
<td>17SEP2001</td>
<td>Tally, Ms. Julia</td>
</tr>
<tr>
<td>10</td>
<td>C006</td>
<td>Seattle</td>
<td>01OCT2001</td>
<td>Berthan, Ms. Judy</td>
</tr>
<tr>
<td>11</td>
<td>C001</td>
<td>Boston</td>
<td>12NOV2001</td>
<td>Hallis, Dr. George</td>
</tr>
<tr>
<td>12</td>
<td>C002</td>
<td>Seattle</td>
<td>03DEC2001</td>
<td>Wickam, Dr. Alice</td>
</tr>
</tbody>
</table>
Creating and Using Macro Programs

Examples

Using Arithmetic and Logical Expressions

The %EVAL Function

The %EVAL function evaluates integer arithmetic or logical expressions. Logical expressions and arithmetic expressions are sequences of operators and operands forming sets of instructions that are evaluated to produce a result.

- An arithmetic expression contains an arithmetic operator.
- A logical expression contains a logical operator.

General form, %EVAL function:

```
%EVAL(arithmetic or logical expression);
```

The %EVAL function

- translates integer strings and hexadecimal strings to integers
- translates tokens representing arithmetic, comparison, and logical operators to macro-level operators
- performs arithmetic and logical operations.

For arithmetic expressions, if an operation results in a non-integer value, %EVAL truncates the value to an integer. Also, %EVAL returns a null value and issues an error message when non-integer values are used in arithmetic expressions.

%EVAL evaluates logical expressions and returns a value to indicate if the expression is true or false. A value of 0 indicates that the expression is false, and a value of 1 or any other numeric value indicates that the expression is true.

The %EVAL function does not convert the following to numeric values:

- numeric strings that contain a period or E-notation
- SAS date and time constants.

Let’s look at some examples.

Examples

The following table shows several examples of arithmetic and logical expressions, as well as the results that %EVAL produces when it evaluates these expressions.

<table>
<thead>
<tr>
<th>Obs</th>
<th>course_code</th>
<th>location</th>
<th>begin_date</th>
<th>teacher</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C003</td>
<td>Dallas</td>
<td>07JAN2002</td>
<td>Forest, Mr. Peter</td>
</tr>
<tr>
<td>2</td>
<td>C004</td>
<td>Boston</td>
<td>21JAN2002</td>
<td>Tally, Ms. Julia</td>
</tr>
<tr>
<td>3</td>
<td>C005</td>
<td>Seattle</td>
<td>25FEB2002</td>
<td>Hallis, Dr. George</td>
</tr>
<tr>
<td>4</td>
<td>C006</td>
<td>Dallas</td>
<td>25MAR2002</td>
<td>Berthan, Ms. Judy</td>
</tr>
</tbody>
</table>
If you submit these statements...

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%put value=%eval(10 lt 2);</code></td>
<td><code>value=0</code></td>
</tr>
<tr>
<td><code>%put value=10+2;</code></td>
<td><code>value=10+2</code></td>
</tr>
<tr>
<td><code>%put value=%eval(10+2);</code></td>
<td><code>value=12</code></td>
</tr>
<tr>
<td><code>%let counter=2;</code></td>
<td></td>
</tr>
<tr>
<td><code>%let counter=%eval(&amp;counter+1);</code></td>
<td><code>counter=3</code></td>
</tr>
<tr>
<td><code>%let counter=&amp;counter;</code></td>
<td></td>
</tr>
<tr>
<td><code>%let numer=2;</code></td>
<td></td>
</tr>
<tr>
<td><code>%let denom=8;</code></td>
<td></td>
</tr>
<tr>
<td><code>%put value=%eval(&amp;numer/&amp;denom);</code></td>
<td><code>value=0</code></td>
</tr>
<tr>
<td><code>%let numer=2;</code></td>
<td></td>
</tr>
<tr>
<td><code>%let demon=8;</code></td>
<td></td>
</tr>
<tr>
<td><code>%put value=%eval(&amp;numer/&amp;denom*&amp;denom);</code></td>
<td><code>value=0</code></td>
</tr>
<tr>
<td><code>%put value=%eval(&amp;denom*&amp;numer/&amp;denom);</code></td>
<td><code>value=2</code></td>
</tr>
<tr>
<td><code>%let real=2.4;</code></td>
<td></td>
</tr>
<tr>
<td><code>%let int=8;</code></td>
<td></td>
</tr>
<tr>
<td><code>%put value=%eval(&amp;real+%int);</code></td>
<td><code>value=0</code></td>
</tr>
</tbody>
</table>

In the last example above, the decimal value of the `real` variable causes an error message to be written to the SAS log, as shown here.

Table 11.18  SAS Log

1  `%let real=2.4;`  
2  `%let int=8;`  
3  `%put value=%eval(&real+%int);`  

ERROR: A character operand was found in the `%EVAL` function or `%IF` condition where a numeric operand is required. The condition was: `2.4+8`  

Because `%EVAL` does not convert a value that contains a period to a number, the operands are evaluated as character operands. You have seen that the `%EVAL` function generates ERROR messages in the log when it encounters an expression that contains non-integer values. In order to avoid these ERROR messages, you can use the `%SYSEVALF` function. The `%SYSEVALF` function evaluates arithmetic and logical expressions using floating-point arithmetic.

General form, `%SYSEVALF` function:

`%SYSEVALF(expression<, conversion-type>);`

where

- `expression` is an arithmetic or logical expression to evaluate.
- `conversion-type` optionally converts the value returned by `%SYSEVALF` to the type of value specified. Conversion-type can be BOOLEAN, CEIL, FLOOR, or INTEGER.
The `%SYSEVALF` function performs floating-point arithmetic and returns a value that is formatted using the BEST32 format. The result of the evaluation is always text.

**Example**

The macro in the following example performs all types of conversions for values in the `%SYSEVALF` function:

```sas
%macro figureit(a,b);
  %let y=%sysevalf(&a+&b);
  %put The result with SYSEVALF is: &y;
  %put BOOLEAN conversion: %sysevalf(&a +&b, boolean);
  %put CEIL conversion: %sysevalf(&a +&b, ceil);
  %put FLOOR conversion: %sysevalf(&a +&b, floor);
  %put INTEGER conversion: %sysevalf(&a +&b, integer);
%mend figureit;

%figureit(100,1.59)
```

Executing this program writes the following lines to the SAS log.

Table 11.19  SAS Log

<table>
<thead>
<tr>
<th>Conversion Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>The result with SYSEVALF is:</td>
<td>101.59</td>
</tr>
<tr>
<td>BOOLEAN conversion:</td>
<td>1</td>
</tr>
<tr>
<td>CEIL conversion:</td>
<td>102</td>
</tr>
<tr>
<td>FLOOR conversion:</td>
<td>101</td>
</tr>
<tr>
<td>INTEGER conversion:</td>
<td>101</td>
</tr>
</tbody>
</table>

**Automatic Evaluation**

 `%SYSEVALF` is the only macro function that can evaluate logical expressions that contain floating point or missing values. Specifying a conversion type can prevent problems when `%SYSEVALF` returns missing or floating point values to macro expressions or macro variables that are used in other macro expressions that require an integer value.

Keep in mind that any macro language function or statement that requires a numeric or logical expression automatically invokes the `%EVAL` function. This includes the `%SCAN` function, the `%SUBSTR` function, the `%IF-%THEN` statement, and more.
Summary

This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Basic Concepts

A macro program is created with a macro definition, which consists of a %MACRO statement and a %MEND statement. The %MACRO statement also provides a name for the macro. Any combination of macro language statements and SAS language statements can be placed in a macro definition. The macro definition must be compiled before it is available for execution. The MCOMPILENOTE option will cause a note to be issued to the SAS log when a macro has completed compilation. To execute a name style macro, you submit a call to the macro by preceding the macro name with a percent sign.

Developing and Debugging Macros

Two system options, MLOGIC and MPRINT, are useful for macro development and debugging. The MLOGIC option writes messages that trace macro execution to the SAS log. The MPRINT option prints the text that is sent to the compiler after all macro resolution has taken place. The SYMBOLGEN option and macro comments are also useful for macro development and debugging.

Using Macro Parameters

You can use parameter lists in your macro definition in order to make your macros more flexible and easier to adapt. Parameters can be either positional or keyword. You can also use mixed parameter lists that contain both positional and keyword parameters. Parameters define macro variables that can take on different values when you call the macro, including null values. You can use the PARMBUFF option in conjunction with the automatic macro variable SYSPBUFF to define a macro that accepts a varying number of parameters each time you call it.

Understanding Symbol Tables

When a macro executes, it sometimes creates its own temporary symbol table, called a local symbol table. The local symbol table exists in addition to the global symbol table. If a macro creates or resolves macro variables, a local symbol table might be used. In order to fully control macro behavior, you must understand the basic rules that the macro processor uses to determine which symbol table to access under specific circumstances. Statements such as %GLOBAL and %LOCAL enable you to explicitly define where macro variables are stored. The %SYMDEL statement enables you to delete a macro variable from the global symbol table during a SAS session.

You can call a macro within a macro definition. That is, you can nest macros. When a nested macro is called, multiple local symbol tables can exist. The MPRINTNEST and MLOGICNEST options provide nesting information in the messages that are written to the SAS log for the MPRINT and MLOGIC options.
Processing Statements Conditionally

Conditional processing is available with the %IF-%THEN/%ELSE statements. These statements control what action the macro processor takes when an expression evaluates to true or to false. The action could be the execution of other macro programming statements or the placement of text onto the input stack. If the code that is used to describe this action includes multiple statements, you must enclose this code between a %DO statement and a %END statement.

It is possible to conditionally place whole SAS steps, whole SAS statements, or parts of SAS statements onto the input stack.

Processing Statements Iteratively

To perform repetitive actions, you can use %DO loops. You can use iterative processing to generate complete SAS steps, individual statements, or data-dependent steps.

Using Arithmetic and Logical Expressions

You use the %EVAL function to evaluate arithmetic or logical expressions that do not contain any non-integer or missing values. Macro language functions and statements that require a numeric or logical expression automatically use the %EVAL function. You use the %SYSEVALF function to evaluate arithmetic or logical expressions that contain non-integer or missing values.

Syntax

```
%MACRO macro-name;
    text
%MEND <macro-name>;

OPTIONS MCOMPILENOTE= NONE | NOAUTOCALL | ALL;
OPTIONS MPRINT | NOPRINT;
OPTIONS MLOGIC | NOMLOGIC;
OPTIONS MLOGICNEST | NOMLOGICNEST;
OPTIONS MPRIINTNEST | NOMPRINTNEST;

/*comment; 

%MACRO macro-name(parameter-1<,...,parameter-n>);
    text
%MEND <macro-name>;

%MACRO macro-name/ PARMBUFF;
    text
%MEND;
```
Sample Programs

Defining a Basic Macro

%MACRO prtlast;
   proc print data=&syslast (obs=5);
   title "Listing of &syslast data set";
   run;
%MEND;

Defining a Macro with Positional Parameters

%MACRO printdsn(dsn,vars);
   proc print data=&dsn;
%MEND;
Defining a Macro with Keyword Parameters

%macro printdsn(dsn=sasuser.courses, vars=course_code course_title days);
proc print data=&dsn;
  var &vars;
  title "Listing of %upcase(&dsn) data set";
run;
%mend;

Defining a Macro with Mixed Parameters

%macro printdsn(dsn, vars=course_title course_code days);
proc print data=&dsn;
  var &vars;
  title "Listing of %upcase(&dsn) data set";
run;
%mend;

Using the %IF-%THEN Statement

%macro choice(status);
data fees;
  set sasuser.all;
  %if &status=PAID %then %do;
    where paid='Y';
    keep student_name course_code begin_date totalfee;
  %end;
  %else %do;
    where paid='N';
    keep student_name course_code begin_date totalfee latechg;
    latechg=fee*1.10;
  %end;
  /* add local surcharge */
  if location='Boston' then totalfee=fee*1.06;
  else if location='Seattle' then totalfee=fee*1.025;
  else if location='Dallas' then totalfee=fee*1.05;
run;
%mend choice;

Using the Iterative %DO Statement

%macro hex(start=1, stop=10, incr=1);
  %local i;
  data _null_;%do i=&start %to &stop %by &incr;
    value=&i;
    put
      "Hexadecimal form of &i is " value hex6.;
Points to Remember

- Macro programs are defined by using a %MACRO statement and a %MEND statement.
- The MPRINT, MLOGIC, and SYMBOLGEN system options can be useful for developing and debugging macro programs.
- Parameters can make your macro programs more flexible by creating local macro variables whose values can be updated by the macro call.
- You can use the %IF-%THEN statement to conditionally process whole SAS steps, SAS statements, or parts of statements.
- You can use the iterative %DO statement to create macro loops that can process repetitive tasks.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which of the following is false?
   - a. A %MACRO statement must always be paired with a %MEND statement.
   - b. A macro definition can include macro variable references, but it cannot include SAS language statements.
   - c. Only macro language statements are checked for syntax errors when the macro is compiled.
   - d. Compiled macros are stored in a temporary SAS catalog by default.

2. Which of the following examples correctly defines a macro named `Print` that includes parameters named `vars` and `total`?
   - a. `%macro print(vars, total);
      proc print data=classes;
      var vars;
      sum total;
      run;
%end;
%end hex;
run;
%mend hex;
options mprint mlogic symbolgen;
%hex(start=20, stop=30, incr=2)

   - b. `%macro print('vars', 'total');
      proc print data=classes;
      var &vars;
      sum &total;
      run;
%end;
%end hex;
run;
%mend hex;

   - c. `%macro print(vars, total);
      proc print data=classes;
      var &vars;
      sum &total;
      run;
%end;
%end hex;
run;
%mend hex;`


3 Which of the following correctly references the macro named Printdsn as shown here:

```sas
%macro printdsn(dsn, vars);
  %if &vars= %then %do;
    proc print data=&dsn;
    title "Full Listing of %upcase(&dsn) data set";
    run;
  %end;
  %else %do;
    proc print data=&dsn;
    var &vars;
    title "Listing of %upcase(&dsn) data set";
    run;
  %end;
%mend;
```

a %printdsn(sasuser.courses, course_title days);
b %printdsn(dsn=sasuser.courses, vars=course_title days)
c %printdsn(sasuser.courses, course_title days)
d %printdsn(sasuser.courses, course_title, days)

4 If you use a mixed parameter list in your macro program definition, which of the following is false?

a You must list positional parameters before any keyword parameters.
b Values for both positional and keyword parameters are stored in a local symbol table.
c Default values for keyword parameters are the values that are assigned in the macro definition, whereas positional parameters have a default value of null.
d You can assign a null value to a keyword parameter in a call to the macro by omitting the parameter from the call.

5 Which of the following is false?

a A macro program is compiled when you submit the macro definition.
b A macro program is executed when you call it (%macro-name).
c A macro program is stored in a SAS catalog entry only after it is executed.
d A macro program is available for execution throughout the SAS session in which it is compiled.

6 When you use an %IF-%THEN statement in your macro program,

a you must place %DO and %END statements around code that describes the conditional action, if that code contains multiple statements.
b the %ELSE statement is optional.
c you cannot refer to DATA step variables in the logical expression of the %IF statement.
d all of the above.
7 Which of the following can be placed onto the input stack?
   a only whole steps.
   b only whole steps or whole statements.
   c only whole statements or pieces of text within a statement.
   d whole steps, whole statements, or pieces of text within statements.

8 Which of the following will create a macro variable named class in a local symbol table?
   a data _null_
      set sasuser.courses;
      %let class=course_title;
      run;
   b data _null_
      set sasuser.courses;
      call symput('class', course_title);
      run;
   c %macro sample(dsn);
      %local class;
      %let class=course_title;
      data _null_
      set &dsn;
      run;
      %mend;
   d %global class;
      %macro sample(dsn);
      %let class=course_title;
      data _null_
      set &dsn;
      run;
      %mend;

9 Which of the following examples correctly defines the macro program Hex?
   a %macro hex(start=1, stop=10, incr=1);
      %local i;
      data _null_
      %do i=&start to &stop by &incr;
      value=&i;
      put "Hexadecimal form of &i is " value hex6.;
      %end;
      run;
      %mend hex;
   b %macro hex(start=1, stop=10, incr=1);
      %local i;
      data _null_
      %do i=&start %to &stop %by &incr;
      value=&i;
      put "Hexadecimal form of &i is " value hex6.;
      %end;
      run;
      %mend hex;
   c %macro hex(start=1, stop=10, incr=1);
      %local i;
```
data _null_;  
%do i=&start to &stop by &incr;  
   value=&i;  
   put "Hexadecimal form of &i is " value hex6.;  
run;  
%mend hex;  

%macro hex(start=1, stop=10, incr=1);  
   %local i;  
   data _null_;  
   %do i=&start to &stop by &incr;  
      value=&i;  
      put "Hexadecimal form of &i is " value hex6.;  
   %end  
run;  
%mend hex;  
```

10. When you submit a call to a compiled macro, what happens?

   a. First, the macro processor checks all macro programming statements in the macro for syntax errors. Then the macro processor executes all statements in the macro.

   b. The macro processor executes compiled macro programming statements. Then any SAS programming language statements are executed by the macro processor.

   c. First, all compiled macro programming statements are executed by the macro processor. After all macro statements have been processed, any SAS language statements are passed back to the input stack in order to be passed to the compiler and then executed.

   d. The macro processor executes compiled macro statements. If any SAS language statements are encountered, they are passed back to the input stack. The macro processor pauses while those statements are passed to the compiler and then executed. Then the macro processor continues to repeat these steps until it reaches the %MEND statement.
Overview 424

Introduction 424
Objectives 424
Prerequisites 424

Understanding Session-Compiled Macros 424
Storing Macro Definitions in External Files 425
Example 426
Storing Macro Definitions in Catalog SOURCE Entries 427
Example 428

The CATALOG Procedure 428
Example 428

The CATALOG Access Method 429
Example 429
Example 430

Using the Autocall Facility 431
Creating an Autocall Library 432
Example 432
Default Autocall Library 432
Example 433
Accessing Autocall Macros 433
Example 434

Using Stored Compiled Macros 435
The Stored Compiled Macro Facility 435
Creating a Stored Compiled Macro 436
Using the SOURCE Option 437
Example 438
Accessing Stored Compiled Macros 438
Example 439
Accessing Stored Macro Code 439
Example 440

Summary 442

Text Summary 442
Understanding Session-Compiled Macros 442
Storing Macro Definitions in External Files 442
Storing Macro Definitions in Catalog SOURCE Entries 442
Using the Autocall Facility 442
Using Stored Compiled Macros 442

Syntax 443
Sample Programs 443

Compiling an Externally Stored Macro Definition with the %INCLUDE Statement 443
Listing the Contents of a Catalog 443
Overview

Introduction

One of the most useful aspects of macro programming is the ability to reuse your macro programs. In Chapter 11, “Creating and Using Macro Programs,” on page 371, you learned that compiled macros are stored in a temporary SAS catalog by default and are available for execution anytime during the current SAS session. You also learned that macros stored in this temporary SAS catalog are deleted at the end of the SAS session.

You might want to store your macros permanently so that you can reuse them in later SAS sessions or share them with others. There are several ways of storing your macro programs permanently and of making them accessible during a SAS session. The methods that you will learn in this chapter are

- the %INCLUDE statement
- the autocall macro facility
- permanently stored compiled macros.

Objectives

In this chapter, you learn to

- use the %INCLUDE statement to make macros available to a SAS program
- use the autocall macro facility to make macros available to a SAS program
- use SAS system options with the autocall facility
- create and use permanently stored compiled macros.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 9, “Introducing Macro Variables,” on page 287
- Chapter 10, “Processing Macro Variables at Execution Time,” on page 325

Understanding Session-Compiled Macros

In Chapter 11, “Creating and Using Macro Programs,” on page 371, you learned that you can submit a macro definition in order to compile a macro. For example, when you submit the macro definition shown here, the macro processor compiles the macro _Prtlast_

```sas
%macro _prtlast;
  %if &syslast ne _NULL_ %then %do;
```
By default, the Prtlast macro is stored in a temporary SAS catalog as Work.Sasmacro.Prtlast.Macro. Macros that are stored in this temporary SAS catalog are known as session-compiled macros. Once a macro has been compiled, it can be invoked from a SAS program as shown here:

```sas
proc sort data=sasuser.courses out=bydays;
    by days;
run;
%prtlast
```

Session-compiled macros are available for execution during the SAS session in which they are compiled. They are deleted at the end of the session. But suppose you want to save your macros so that they are not deleted at the end of the SAS session. The rest of this chapter looks at methods of storing macros permanently.

---

**Storing Macro Definitions in External Files**

One way to store macro programs permanently is to save them to an external file. You can then use the %INCLUDE statement to insert the statements that are stored in the external file into a program. If the external file contains a macro definition, the macro is compiled when the %INCLUDE statement is submitted. Then the macro can be called again later in the same program, or anytime later in the current SAS session.

General form, %INCLUDE statement:

```sas
%INCLUDE file-specification </SOURCE2>;
```

where

- `file-specification` describes the location of the file that contains the SAS code to be inserted.
- `SOURCE2` causes the SAS statements that are inserted into the program to be displayed in the SAS log. If `SOURCE2` is not specified in the %INCLUDE statement, then the setting of the SAS system option `SOURCE2` controls whether the inserted code is displayed.

By storing your macro program externally and using the %INCLUDE statement, you gain several advantages over using session-compiled macros.

- The source code for the macro definition does not need to be part of your program.
- A single copy of a macro definition can be shared by many programs.
- Macro definitions in external files are easily viewed and edited with any text editor.
- No special SAS system options are required in order to access a macro definition that is stored in an external file.
Example

You can compile a macro by using the `%INCLUDE` statement to insert its definition into a program. Then you can call the macro in order to execute it. Suppose the following macro definition is stored in the external file C:sasfiles\prtlast.sas:

```
%macro prtlast;
   %if &syslast ne _NULL_ %then %do;
   proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
   run;
%end;
%else
   %put No data set has been created yet.;
%mend;
```

You could submit the following code to access, compile, and execute the `Prtlast` macro. The PROC SORT step is included in this example in order to create a data set that the `Prtlast` macro can print.

```
%include 'c:sasfiles\prtlast.sas' /source2;
proc sort data=sasuser.courses out=bydays;
   by days;
run;
%prtlast
```

Note: The location and names of external files will be specific to your operating environment.

The following messages are written to the SAS log when this code is submitted. Notice that the macro definition is written to the log because `SOURCE2` was specified in the `%INCLUDE` statement.
Table 12.1  SAS Log

```
NOTE: %INCLUDE (level 1) file prtlast.sas is file
     C:sasfiles\prtlast.sas.
31 +%macro prtlast;
32 + %if &syslast ne _NULL_ %then %do;
33 + proc print data=&syslast(obs=5);
34 + title 'Listing of &syslast data set';
35 + run;
36 + %end;
37 + %else
38 + %put No data set has been created yet.;
39 + %mend;
NOTE: %INCLUDE (level 1) ending.
40 41 proc sort data=sasuser.courses out=bydays;
42 by days;
43 run;
NOTE: There were 6 observations read from the dataset
SASUSER.COURSES.
NOTE: The data set WORK.BYDAYS has 6 observations and
4 variables.
NOTE: PROCEDURE SORT used:
     real time 0.04 seconds
     cpu time 0.04 seconds
44 45 %prtlast
NOTE: There were 5 observations read from the dataset
WORK.BYDAYS.
NOTE: PROCEDURE PRINT used:
     real time 1.07 seconds
     cpu time 0.26 seconds
```

Here is the output that the code generates.

**Listing of WORK.BYDAYS data set**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Course_Code</th>
<th>Course_Title</th>
<th>Days</th>
<th>Fee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C004</td>
<td>Database Design</td>
<td>2</td>
<td>$375</td>
</tr>
<tr>
<td>2</td>
<td>C005</td>
<td>Artificial Intelligence</td>
<td>2</td>
<td>$400</td>
</tr>
<tr>
<td>3</td>
<td>C001</td>
<td>Basic Telecommunications</td>
<td>3</td>
<td>$795</td>
</tr>
<tr>
<td>4</td>
<td>C003</td>
<td>Local Area Networks</td>
<td>3</td>
<td>$660</td>
</tr>
<tr>
<td>5</td>
<td>C002</td>
<td>Structured Query Language</td>
<td>4</td>
<td>$1150</td>
</tr>
</tbody>
</table>

**Storing Macro Definitions in Catalog SOURCE Entries**

Another way of permanently storing macros is to store a macro definition in a
**SOURCE entry** in a SAS catalog. If you decide to store your macro programs in a SAS
catalog, you must store each macro program in a separate SOURCE entry. It is a good
idea to give each SOURCE entry the same name as the macro program that is stored
within it. For example, a macro named Printit would be stored in a SOURCE entry
that is also named Printit.
Note: SAS catalogs are members of SAS data libraries that store program source code and other types of content.

To store a macro definition as a SOURCE entry in a SAS catalog, you use the Save As Object window.

Example

To save the Printit macro definition to the Sasuser.Mymacs catalog, perform these steps:

1. Select File ▶ Save As Object. In the Save As Object window, select the Sasuser library.
2. If the Sasuser.Mymacs catalog does not already exist, you need to create it. You can either select the Create New Catalog icon or right-click the Save As Object window and select New in order to open the New Catalog window. Enter Mymacs as the name for the new catalog and click OK.
3. Enter Printit in the Entry Name field. Make sure that the Entry Type is set to SOURCE entry (SOURCE), then click Save.

Tip: If you use the Program Editor, you could also use the SAVE command to save your macro definition as a catalog SOURCE entry. To use the SAVE command, you enter save libref.catalog.entry.source in the command line where libref.catalog.entry is the libref, the catalog name, and the entry name.

The CATALOG Procedure

If you store your macros in a SAS catalog, you might want to view the contents of a particular catalog to see the macros you have stored there. You can use the Explorer window to view the contents of a SAS catalog by navigating to the catalog and double clicking it. You can also use the CATALOG procedure to list the contents of a SAS catalog. The CONTENTS statement of the CATALOG procedure lists the contents of a catalog in the procedure output.

General form, CATALOG procedure with CONTENTS statement:

```
PROC CATALOG CATALOG=libref.catalog;
   CONTENTS;
QUIT;
```

where

- `libref.catalog` is a valid two-level catalog name.

Note: CAT= is an alias for CATALOG.

Example

You can use PROC CATALOG to view all of the macros that are stored in the temporary Work.Sasmacr catalog, as follows:
proc catalog cat=work.sasmacr;
   contents;
   title "Default Storage of SAS Macros";
quit;

This PROC CATALOG step produces results that are similar to the output shown below. The macros that are actually listed will be the macros that have been compiled during the current SAS session.

### Default Storage of SAS Macros

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Create Date</th>
<th>Modified Date</th>
</tr>
</thead>
</table>

### The CATALOG Access Method

If you store a macro definition in a SOURCE entry of a SAS catalog, you can use the CATALOG access method in a FILENAME statement in conjunction with the %INCLUDE statement to insert the macro definition into a SAS program.

```sas
FILENAME fileref CATALOG 'libref.catalog.entry-name.entry-type';
%INCLUDE fileref;
```

where

- `fileref` is a valid fileref.
- `libref.catalog.entry-name.entry-type` is a four-level SAS catalog entry name.
- `entry-type` is SOURCE.

### Example

Suppose you have stored the following macro definition as a SOURCE entry in the SAS catalog `Sasuser.Mymacs`:

```sas
%macro prtlast;
   %if &syslast ne _NULL_ %then %do;
      proc print data=&syslast(obs=5);
      title "Listing of &syslast data set";
```
You can use the CATALOG access method along with the `%INCLUDE` statement to compile the macro `Prtlast`. Then you can reference the macro later in the program.

```sas
filename prtlast catalog 'sasuser.mymacs.prtlast.source';
#include prtlast;
proc sort data=sasuser.courses out=bydays;
   by days;
run;
#include prtlast
```

You can also use the CATALOG access method to reference multiple SOURCE entries as long as the entries are stored in the same SAS catalog.

General form, CATALOG access method to reference multiple SOURCE entries:

```sas
FILENAME fileref CATALOG 'libref.catalog';
#include fileref(entry-1);
#include fileref(entry-2);
```

where

- `fileref` is a valid fileref.
- `libref.catalog` is a two-level catalog name.
- `entry-1` and `entry-2` are names of SOURCE entries in `library.catalog`.

---

**Example**

Suppose you have two macros, named `Prtlast` and `Sortlast`, that are stored in a SAS catalog.

Catalog Entry: **Sasuser.Mymacs.Prtlast.Source**

```sas
%macro prtlast;
   %if &syslast ne _NULL_ %then %do;
      proc print data=&syslast(obs=5);
         title "Listing of &syslast data set";
      run;
   %end; %else
   %put No data set has been created yet.; %mend;
```

Catalog Entry: **Sasuser.Mymacs.Sortlast.Source**

```sas
%macro sortlast(sortby);
   %if &syslast ne _NULL_ %then %do;
```
You can use the CATALOG access method in conjunction with the %INCLUDE statement to compile both macros. Then you can call the macros later in the program. In this example, assume that the macros have the same names as the SOURCE entries in which they are stored:

```sas
filename prtsort catalog 'sasuser.mymacs';
%include prtsort(prtlast) / source2;
%include prtsort(sortlast) / source2;
```

data current(keep=student_name course_title begin_date location);
  set sasuser.all;
  if year(begin_date)=2001;
    diff=year(today())-year(begin_date);
    begin_date=begin_date+(365*diff);
  run;

%sortlast(begin_date)
%prtlast

This code produces the following output:

<table>
<thead>
<tr>
<th>Obs</th>
<th>Student_Name</th>
<th>Location</th>
<th>Begin_Date</th>
<th>Course_Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bills, Ms. Paulette</td>
<td>Boston</td>
<td>08JAN2002</td>
<td>Local Area Networks</td>
</tr>
<tr>
<td>2</td>
<td>Chevarley, Ms. Adene</td>
<td>Boston</td>
<td>08JAN2002</td>
<td>Local Area Networks</td>
</tr>
<tr>
<td>3</td>
<td>Clough, Ms. Fatti</td>
<td>Boston</td>
<td>08JAN2002</td>
<td>Local Area Networks</td>
</tr>
<tr>
<td>4</td>
<td>Crace, Mr. Ron</td>
<td>Boston</td>
<td>08JAN2002</td>
<td>Local Area Networks</td>
</tr>
<tr>
<td>5</td>
<td>Dams, Mr. Bruce</td>
<td>Boston</td>
<td>08JAN2002</td>
<td>Local Area Networks</td>
</tr>
</tbody>
</table>

Using the Autocall Facility

You can make macros accessible to your SAS session or program by using the autocall facility to search predefined source libraries for macro definitions. These predefined source libraries are known as autocall libraries. You can store your macro definitions permanently in an autocall library, and you can set up multiple autocall libraries.

When you use this approach, you do not need to compile the macro in order to make it available for execution. That is, if the macro definition is stored in an autocall library, then you do not need to submit or include the macro definition before you submit a call to the macro.

Suppose you have stored a file that contains a macro definition in your autocall library. When you submit a call to that macro
the macro processor searches the autocall library for the macro
- the macro is compiled and stored as it would be if you had submitted it (that is, the compiled macro is stored in the default location of Work.Sasmacr)
- the macro is executed.

Once it has been compiled, the macro can be executed as needed throughout the same SAS session. At the end of the SAS session, the compiled macro is deleted from the Work.Sasmacr catalog, but the source code remains in the autocall library.

Creating an Autocall Library

An autocall library can be either
- a directory that contains source files
- a partitioned data set (PDS)
- a SAS catalog.

The method for creating an autocall library depends on the operating environment that you are using.

To create an autocall library in a directory-based operating system such as Windows, UNIX, or OpenVMS, create a directory in which to store macro definitions. Each macro definition in this directory will be a separate file that has the extension .sas and that has the same name as the macro that it contains.

Example

Suppose you want to save the macro Prtlast in an autocall library. In a directory-based operating system, the first step is to create a directory that will hold your macro source files. You can use the Save As window to create the directory, and to save the macro definition in that directory. With the Prtlast definition in an active code editing window, select File ▶ Save As. In the Save As window, navigate to the location where you want to create your autocall library. Select New Folder, enter the directory name, and click OK. Then enter Prtlast as the filename, make sure the file type is .sas, and click Save.

Tip: You could also use the FILE command to save your macro definition in an autocall library. To use the FILE command, you enter file ‘<path>external-file-name’ in the command line.

Default Autocall Library

SAS provides several macros in a default autocall library for you. Some of the macros in the autocall library that SAS provides are listed here.

<table>
<thead>
<tr>
<th>Macro Syntax</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%LOWCASE(argument)</td>
<td>converts letters in its argument from uppercase to lowercase</td>
</tr>
<tr>
<td>%QLOWCASE(argument)</td>
<td>converts letters in its argument from uppercase to lowercase, and returns a result that masks special characters and mnemonic operators</td>
</tr>
<tr>
<td>%LEFT(argument)</td>
<td>removes leading blanks from the argument</td>
</tr>
<tr>
<td>%TRIM(argument)</td>
<td>removes trailing blanks from the argument</td>
</tr>
</tbody>
</table>
Storing Macro Programs

Accessing Autocall Macros

<table>
<thead>
<tr>
<th>Macro Syntax</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%CMPRES(argument)</td>
<td>removes multiple blanks from the argument</td>
</tr>
<tr>
<td>%DATATYP(argument)</td>
<td>returns the string NUMERIC or CHAR, depending on whether the argument is an integer or a character string</td>
</tr>
</tbody>
</table>

You might be familiar with SAS functions such as TRIM and LEFT. The macros that SAS supplies look like macro functions, but they are in fact macros. One of the useful things about these macros is that in addition to using them in your SAS programs, you can see their source code.

Example

The macro definition for the Lowcase macro is shown below. Notice that the comments that are included in this macro provide information about using the macro. All of the macros that SAS provides in the autocall library include explanatory comments so that they will be easy for you to understand and use.

```sas
%macro lowcase(string);
%******************************************************;
%* MACRO: LOWCASE *
%* USAGE: 1) %lowcase(argument) *
%* DESCRIPTION: *
%* This macro returns the argument passed to it unchanged except that all upper-case alphabetic characters are changed to their lower-case equivalents. *
%* E.g.: %let macvar=%lowcase(SAS Institute Inc.); *
%* The variable macvar gets the value "sas institute inc." *
%* NOTES: *
%* Although the argument to the %UPCASE macro function may contain commas, the argument to %LOWCASE may not, unless they are quoted. *
%* Because %LOWCASE is a macro, not a function, it interprets a comma as the end of a parameter. *
%******************************************************;
%sysfunc(lowcase(%nrbquote(&string)))
%mend;
```

Accessing Autocall Macros

Remember that an autocall library is either a SAS catalog, an external directory, or a partitioned data set. This is true both for the default autocall library that SAS supplies and for autocall libraries that you create.

In order to access a macro definition that is stored in an autocall library, you must use two SAS system options, as follows:
The MAUTOSOURCE system option must be specified.
The SASAUTOS system option must be set to identify the location of the autocall library or libraries.

Both the MAUTOSOURCE and SASAUTOS system options can be set either at SAS invocation or with an OPTIONS statement during program execution.
The MAUTOSOURCE system option controls whether the autocall facility is available.

General form, MAUTOSOURCE system option:

```
OPTIONS MAUTOSOURCE | NOMAUTOSOURCE;
```

where

MAUTOSOURCE

is the default setting, and specifies that the autocall facility is available.

NOMAUTOSOURCE

specifies that the autocall facility is not available.

The SASAUTOS system option controls where the macro facility looks for autocall macros.

General form, SASAUTOS system option:

```
OPTIONS SASAUTOS=library-1;
OPTIONS SASAUTOS=(library-1,...,library-n);
```

where

the values of library-1 through library-n

are references to source libraries that contain macro definitions. To specify a source library you can

- use a fileref to refer to its location
- specify the pathname (enclosed in quotation marks) for the library.

Unless your system administrator has changed the default value for the SASAUTOS system option, its value is the fileref Sasautos, and that fileref points to the location where the default autocall library was created during installation. The Sasautos fileref can refer to multiple locations that are concatenated.

Generally, it is a good idea to concatenate any autocall libraries that you create yourself with the default autocall library in the value of the SASAUTOS system option. Otherwise, the new autocall library will replace the default or existing libraries in the value of SASAUTOS, and the autocall facility will have access to only the new autocall library.

Example

Suppose you want to access the Prtlast macro, which is stored in the autocall library C:\Mysasfiles. You also want to make sure that the default autocall library (which the
fileref *Sasautos* points to) is still available to the autocall facility. You would submit the following code:

```sas
options mautosource sasautos=('c:\mysasfiles',sasautos);
%prtlast
```

*Note:* The MAUTOLOCDISPLAY option is a Boolean option that is available beginning in SAS 9. When set, MAUTOLOCDISPLAY will cause a note to be issued to the SAS log indicating where the source code was obtained to compile an autocall macro. The note is similar to the information displayed when using the MLOGIC option. The default setting of this option is NOMAUTOLOCDISPLAY. △

When the autocall facility is in effect, if you invoke a macro that has not been previously compiled, the macro facility automatically

1. searches the autocall library (or each autocall library in turn if multiple libraries are identified in the SASAUTOS system option) for a member that has the same name as the invoked macro
2. brings the source statements into the current SAS session if the member is found
3. issues an error message if the member is not found
4. submits all statements in the member in order to compile the macro
5. stores the compiled macro in the temporary catalog *Work.Sasmacr*
6. calls the macro.

The autocall facility does not search for a macro in the autocall library if the macro has already been compiled during the current SAS session. In that case, the session-compiled macro is executed.

---

**Using Stored Compiled Macros**

---

**The Stored Compiled Macro Facility**

Remember that when a macro is compiled, it is stored in the temporary SAS catalog *Work.Sasmacr* by default. You can also store compiled macros in a permanent SAS catalog. Then you can use the *Stored Compiled Macro Facility* to access permanent SAS catalogs that contain compiled macros.

There are several advantages to using stored compiled macros:

- SAS does not need to compile a macro definition when a macro call is made.
- Session-compiled macros and the autocall facility are also available in the same session.
- Users cannot modify compiled macros.

Two SAS system options affect stored compiled macros: MSTORED and SASMSTORE. The MSTORED system option controls whether the Stored Compiled Macro Facility is available.
Creating a Stored Compiled Macro

To create a permanently stored compiled macro, you must
1 assign a libref to the SAS library in which the compiled macro will be stored
2 set the system options MSTORED and SASMSTORE=libref
3 use the STORE option in the %MACRO statement when you submit the macro definition.
General form, macro definition with STORE option:

```
%MACERO macro-name <(parameter-list)> /STORR
   <DES='description'>;
   text
%MEND <macro-name>;
```

where

- `description` is an optional 156-character description that appears in the catalog directory.
- `macro-name` names the macro.
- `parameter-list` names one or more local macro variables whose values you specify when you invoke the macro.
- `text` can be:
  - constant text, possibly including SAS data set names, SAS variable names, or SAS statements
  - macro variables, macro functions, or macro program statements
  - any combination of the above.

There are several restrictions on stored compiled macros.

- `Sasmacr` is the only catalog in which compiled macros can be stored. You can create a catalog named `Sasmacr` in any SAS library. You should not rename this catalog or its entries.
- You cannot copy stored compiled macros across operating systems. You must copy the source program and re-create the stored compiled macro.
- The source cannot be re-created from the compiled macro. You should retain the original source program. For convenience, you can store the source program in an autocall library. Alternatively, you can store the source program as a source entry in the same catalog as the compiled macro.

### Using the SOURCE Option

An alternative to saving your source program separately from the stored compiled macro is to use the SOURCE option in the `%MACRO` statement to combine and store the source of the compiled macro with the compiled macro code. The SOURCE option requires that the STORE option and the MSTORED option be set. The `%MACRO` statement below shows the correct syntax for using the SOURCE option.

```
%macro macro-name<(parameter list)> /STORE SOURCE;
```

The source code that is saved by the SOURCE option begins with the `%MACRO` keyword and ends with the semicolon following the `%MEND` statement.

**Tip:** The SOURCE option cannot be used on nested macro definitions.
Example

Suppose you want to store the *Words* macro in compiled form in a SAS library. This example shows the macro definition for *Words*. The macro takes a text string, divides it into words, and creates a series of macro variables to store each word.

Notice that both the STORE option and the SOURCE option are used in the macro definition so that *Words* will be permanently stored as a compiled macro and the macro source code will be stored with it, as follows:

```
libname macrolib 'c:\storedlib';
options mstored sasmstore=macrolib;

%macro words(text,root=w,delim=%str( ));
   %local i word;
   %let i=1;
   %let word=%scan(&text,&i,&delim);
   %do %while (&word ne );
      %global &root&i;
      %let &root&i=&word;
      %let i=%eval(&i+1);
      %let word=%scan(&text,&i,&delim);
   %end;
   %global &root.num;
   %let &root.num=%eval(&i-1);
%mend words;
```

If the *Sasmacr* catalog does not exist in the *Macrolib* library, it is automatically created. You can list the contents of the *Macrolib.Sasmacr* catalog to verify that the compiled macro was created, as follows:

```
proc catalog cat=macrolib.sasmacr;
   contents;
   title "Stored Compiled Macros";
quit;
```

Here is the output from the PROC CATALOG step if no other compiled macros are stored in *Macrolib.Sasmacr*.

### Stored Compiled Macros

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Create Date</th>
<th>Modified Date</th>
<th>Description</th>
</tr>
</thead>
</table>

### Accessing Stored Compiled Macros

In order to access a stored compiled macro, you must

1. assign a libref to the SAS library that contains a *Sasmacr* catalog in which the macro was stored
2. set the system options MSTORED and SASMSTORE=libref
3. call the macro.
Only one permanent catalog containing compiled macros can be accessed at any given time.

Example

The following program calls the Words macro. Assume that the Words macro was compiled and stored in an earlier SAS session.

    libname macrolib 'c:\storedlib';
    options mstored sasmstore=macrolib;

    %words(This is a test)
    %put Number of Words (wnum): &wnum;
    %put Word Number 1 (w1): &w1;
    %put Word Number 2 (w2): &w2;
    %put Word Number 3 (w3): &w3;
    %put Word Number 4 (w4): &w4;

Here is a portion of the messages that are written to the SAS log when this code is submitted.

Table 12.2 SAS Log

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>libname macrolib 'c:\storedlib';</td>
</tr>
<tr>
<td></td>
<td>NOTE: Libref MACROLIB was successfully assigned as follows:</td>
</tr>
<tr>
<td></td>
<td>Engine: V9</td>
</tr>
<tr>
<td></td>
<td>Physical Name: c:\storedlib</td>
</tr>
<tr>
<td>10</td>
<td>options mstored sasmstore=macrolib;</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>%words(This is a test)</td>
</tr>
<tr>
<td>13</td>
<td>%put Number of Words (wnum): &amp;wnum;</td>
</tr>
<tr>
<td>14</td>
<td>Word Number 1 (w1): This</td>
</tr>
<tr>
<td>15</td>
<td>Word Number 2 (w2): is</td>
</tr>
<tr>
<td>16</td>
<td>Word Number 3 (w3): a</td>
</tr>
<tr>
<td>17</td>
<td>Word Number 4 (w4): test</td>
</tr>
</tbody>
</table>

Accessing Stored Macro Code

If you use the SOURCE option of the %MACRO statement to store your macro source code along with the stored compiled macro, you can use the %COPY statement to access the stored source code.
General form, %COPY statement:

```
%COPY macro-name /SOURCE <other option(s)>;
```

where

*macro-name*

is the name of the macro whose source code will be accessed.

*SOURCE*

specifies that the source code of the macro will be copied to the output destination. If no output destination is specified, the source is written to the SAS log.

*other options*

include the following options:

- **LIBRARY=** specifies the libref of a SAS data library that contains a catalog of stored compiled SAS macros. If no library is specified, the libref specified by the SASMSTORE= option is used. The libref cannot be `Work`.
- **OUTFILE=** specifies the output destination of the %COPY statement. The value cannot be a fileref or an external file.

---

**Example**

Suppose you submitted the program below to create a stored compiled macro named *Words*.

```sas
libname macrolib 'c:\storedlib';
options mstored sasmstore=macrolib;

%macro words(text,root=w,delim=%str( ))/store source;
  %local i word;
  %let i=1;
  %let word=%scan(&text,&i,&delim);
  %do %while (&word ne );
    %global &root&i;
    %let &root&i=&word;
    %let i=%eval(&i+1);
    %let word=%scan(&text,&i,&delim);
  %end;
  %global &root.num;
  %let &root.num=%eval(&i-1);
%mend words;
```

The %COPY statement writes the source code for the *Words* macro to the SAS log, for example:

```sas
%copy words/source;
```

The partial SAS log below shows the source code of the *Words* macro.
Table 12.3  SAS Log

\[
\begin{verbatim}
17 %copy words/source;
%macro words(text,root=w,delim=%str( ))/store source;
    %local i word;
    %let i=1;
    %let word=%scan(&text,&i,&delim);
    %do %while (&word ne )
        %global &root&i;
        %let &root&i=&word;
        %let i=%eval(&i+1);
        %let word=%scan(&text,&i,&delim);
    %end;
    %global &root.num;
    %let &root.num=%eval(&i-1);
%mend words;
\end{verbatim}
\]

The Stored Compiled Macro Facility can be used in conjunction with the Autocall Facility and with session-compiled macros. When you submit a macro call such as `%words`, the macro processor searches for the macro `Words` as

1. an entry named `Words.Macro` in the temporary `Work.Sasmacr` catalog.
2. an entry named `Words.Macro` in the `Libref.Sasmacr` catalog. The `MSTORED` option must be specified, and the `SASMSTORE` option must have a value of `Libref`.
3. an autocall library member named `Words` that contains the macro definition for the macro `Words`. The `MAUTOSOURCE` option must be specified, and the value of the `SASAUTOS` option must point to the autocall library.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>No</strong></td>
<td></td>
<td>Requires MSTORED and SASMSTORE= system options.</td>
</tr>
<tr>
<td>Is there a member named <code>Words</code> in the autocall library?</td>
<td><strong>Yes</strong></td>
<td>Compile and execute WORDS autocall macro.</td>
</tr>
<tr>
<td><strong>No</strong></td>
<td></td>
<td>Requires MAUTOSOURCE and SASAUTOS= system options.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Issue this message to the SAS log: &quot;WARNING: Apparent invocation of macro WORDS not resolved.&quot;</td>
</tr>
</tbody>
</table>
Summary

This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Understanding Session-Compiled Macros

You can make a macro available to your SAS session by submitting the macro definition before calling the macro. This creates a session-compiled macro. Session-compiled macros are deleted from the temporary SAS catalog `Work.Sasmacr` at the end of the session.

Storing Macro Definitions in External Files

One way to store your macro definitions permanently is to save them in external files. You can make a macro definition that is stored in an external file accessible to your SAS programs by using the `%INCLUDE` statement.

Storing Macro Definitions in Catalog SOURCE Entries

You can also store your macro definitions permanently as SOURCE entries in SAS catalogs. You can use the catalog access method to make these macros accessible to your SAS programs. The PROC CATALOG statement enables you to view a list of the contents of a SAS catalog.

Using the Autocall Facility

You can permanently store macro definitions in source libraries called autocall libraries. SAS provides several macro definitions for you in a default autocall library. You can concatenate multiple autocall libraries. To access macros that are stored in an autocall library, you specify the SASAUTOS and MAUTOSOURCE system options.

Using Stored Compiled Macros

Another efficient way to make macros available to a program is to store them in compiled form in a SAS library. To store a compiled macro permanently, you must set two system options, MSTORED and SASMSTORE. Then you submit one or more macro definitions, using the STORE option in the `%MACRO` statement. The compiled macro is stored as a catalog entry in `Libref.Sasmacr`. The source program is not stored as part of the compiled macro. You should always maintain the original source program for each macro definition in case you need to redefine the macro. You can use the SOURCE option in the `%MACRO` statement to store the macro source code with the compiled macro. If you use the SOURCE option in the `%MACRO` statement, you can use the `%COPY` statement to access the macro source code later.
Syntax

%INCLUDE file-specification /SOURCE2;

FILENAME fileref
   CATALOG 'libref.catalog.entry-name.entry-type';
%INCLUDE fileref;

FILENAME fileref CATALOG 'libref.catalog';
%INCLUDE fileref(entry-1);
%INCLUDE fileref(entry-2);

PROC CATALOG CATALOG=libref.catalog;
   CONTENTS;
QUIT;

OPTIONS MAUTOSOURCE | NOMAUTOSOURCE;
OPTIONS SASAUTOS=library-1;
OPTIONS SASAUTOS=(library-1,...,library-n);
OPTIONS MSTORED | NOMSTORED;
OPTIONS SASMSTORE=libref;

%MACRO macro-name<(parameters-list)> /STORE SOURCE
   <DES='description'>;
   text
%MEND <macro-name>;

Sample Programs

Compiling an Externally Stored Macro Definition with the %INCLUDE Statement

%include 'c:sasfiles\prtlast.sas' / source2;

proc sort data=sasuser.courses out=bydays;
   by days;
run;

%prtlast

Listing the Contents of a Catalog

proc catalog cat=work.sasmacr;
   contents;
   title "Default Storage of SAS Macros";
Using the Catalog Access Method

```sas
filename prtlast catalog 'sasuser.mymacs.prtlast.source';
%include prtlast;
proc sort data=sasuser.courses out=bydays;
   by days;
run;
%prtlast
```

Accessing an Autocall Macro

```sas
options mautosource sasautos=(‘c:\mysasfiles’,sasautos);
%prtlast
```

Creating a Stored Compiled Macro

```sas
libname macrolib ‘c:\storedlib’;
options mstored sasmstore=macrolib;

%macro words(text,root=w,delim=%str( ))/store;
   %local i word;
   %let i=1;
   %let word=%scan(&text,&i,&delim);
   %do %while (&word ne );
      %global &root&i;
      %let &root&i=&word;
      %let i=%eval(&i+1);
      %let word=%scan(&text,&i,&delim);
   %end;
   %global &root.num;
   %let &root.num=%eval(&i-1);
%mend words;
```

Points to Remember

- You can make macros available to your programs in four ways: as session-compiled macros, with a %INCLUDE statement, through the autocall facility, or as stored compiled macros.
- If you use the autocall facility, you must specify the MAUTOSOURCE and SASAUTOS system options.
- If you use the stored compiled macro facility, you must specify the MSTORED and SASMSTORE system options.
- The point at which macro compilation occurs depends on which method you use to access the macro.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.
1. The `%INCLUDE` statement
   a. can be used to insert the contents of an external file into a program.
   b. will cause a macro definition that is stored in an external file to be compiled when the contents of that file are inserted into a program and submitted.
   c. can be specified with the `SOURCE2` option in order to write the contents of the external file that is inserted into a program to the SAS log.
   d. all of the above

2. If you store a macro definition in a SAS catalog `SOURCE` entry
   a. the macro definition can be submitted for compilation by using the `FILENAME` and `%INCLUDE` statements.
   b. you can use the `PROC CATALOG` statement to compile the macro.
   c. the `SOURCE` entry will be deleted at the end of the session.
   d. you do not need to compile the macro before you invoke it in a program.

3. Which of the following programs correctly sets the appropriate system options and calls the macro `Prtlast`? Assume that `Prtlast` is stored in an autocall library and that it has not been compiled during the current SAS session.
   a. `options mautosource; %prtlast`
   b. `libname mylib 'c:\mylib';
      filename macsrc catalog 'mylib.macs';
      %prtlast`
   c. `libname mylib 'c:\mylib';
      filename macsrc 'mylib.macs';
      options mautosource sasautos=(macsrc, sasautos);
      %prtlast`
   d. `libname mylib 'c:\mylib';
      options mautosource sasautos=mylib;
      %prtlast`

4. If you use the Stored Compiled Macro Facility,
   a. the macro processor does not compile a macro every time it is used.
   b. the only compiled macros that the Stored Compiled Macro Facility can access are those that are stored in the `Sasmacr` catalog.
   c. you need to specify the `MSTORED` and `SASMSTORE` system options.
   d. all of the above

5. Which of the following correctly creates a permanently stored compiled macro?
   a. `libname macrolib 'c:\mylib';
      options sasmstore;
      %macro prtlast; / store
      proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
      run;
      %mend;`
   b. `libname macrolib 'c:\mylib';
      options mstored sasmstore=macrolib;
      %macro prtlast / store;
      proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
      run;
      %mend;`
libname macrolib 'c:\mylib';
options mstored sasmstore=macrolib;
%macro prtlast;
  proc print data=&syslast (obs=5);
    title "Listing of &syslast data set";
    run;
%mend;

d libname macrolib 'c:\mylib';
%macro prtlast / store;
  proc print data=&syslast (obs=5);
    title "Listing of &syslast data set";
    run;
%mend;

6 When you submit the following code, what happens?

%macro prtlast;
  proc print data=&syslast (obs=5);
    title "Listing of &syslast data set";
    run;
%mend;

a A session-compiled macro named Prtlast is stored in Work.Sasmacr.
b A macro named Prtlast is stored in the autocall library.
c The Prtlast macro is stored as a stored compiled macro.
d The Prtlast macro is stored as a SOURCE entry in a permanent SAS catalog.

7 Why would you want to store your macros in external files?

a You could easily share your macros with others.
b You could edit your macros with any text editor.
c Your macros would be available for use in later SAS sessions.
d All of the above

8 What will the following PROC CATALOG step do?

proc catalog cat=mylib.sasmacr;
  contents;
quit;

a Copy the contents of the Sasmacr catalog to a temporary data set.
b List the contents of the Sasmacr catalog as output.
c Copy the contents of the output window to the Sasmacr catalog.
d None of the above

9 Which of the following is not true about stored compiled macros?

a Because these stored macros are compiled, you should save and maintain the source for the macro definitions in a different location.
b The Stored Compiled Macro Facility compiles and saves compiled macros in a permanent catalog, in a library that you specify.
c You do not need to specify any system options in order to use the Stored Compiled Macro Facility.
d You cannot move a stored compiled macro to another operating system.

10 Which of the following is not true?

a The autocall macro facility stores compiled SAS macros in a collection of external files called an autocall library.
b Autocall libraries can be concatenated together.
c One disadvantage of the autocall facility is that the first time you call an autocall macro in a SAS session, the macro processor must use system resources to compile it.

d The autocall facility can be used in conjunction with the Stored Compiled Macro Facility.
Advanced SAS Programming Techniques

Chapter 13. . . . . . . Creating Samples and Indexes  451
Chapter 14. . . . . . . Combining Data Vertically  481
Chapter 15. . . . . . . Combining Data Horizontally  513
Chapter 16. . . . . . . Using Lookup Tables to Match Data  559
Chapter 17. . . . . . . Formatting Data  603
Chapter 18. . . . . . . Modifying SAS Data Sets and Tracking Changes  633
CHAPTER 13
Creating Samples and Indexes

Overview 452
   Introduction 452
   Objectives 452
   Prerequisites 453
Creating a Systematic Sample from a Known Number of Observations 453
   Example 453
   Example 454
Creating a Systematic Sample from an Unknown Number of Observations 455
   Example 456
Creating a Random Sample with Replacement 456
   Using the RANUNI Function 456
   Example 457
   Using a Multiplier with the RANUNI Function 457
   Using the CEIL Function 457
   Example 458
   Example 458
Creating a Random Sample without Replacement 459
   Example 459
Using Indexes 460
   Types of Indexes 461
Creating Indexes in the DATA Step 462
   Examples 462
   Example 463
   Determining Whether SAS Is Using an Index 464
   Example 464
Managing Indexes with PROC DATASETS 464
   Example 465
Managing Indexes with PROC SQL 466
   Example 466
Documenting and Maintaining Indexes 467
   Example 468
   Example 469
   Copying Data Sets 469
   Examples 470
   Renaming Data Sets 471
   Example 471
   Renaming Variables 471
   Example 472
Summary 473
   Text Summary 473
   Creating a Systematic Sample from a Known Number of Observations 473
Overview

Introduction

Some of the SAS data sets that you work with might be quite large. Large data sets can take a relatively long time to process because, by default, SAS reads observations in a data set sequentially. For example, assume that your data set has five hundred observations. In order to read the five-hundredth observation, SAS first reads the observations numbered 1 through 499, and then reads observation number 500. Sometimes, you might want to make SAS access specific observations directly for greater speed and efficiency.

You will need to access specific observations directly when you want to create a representative sample of a large data set, which can be much easier to work with than the full data set. For example, if you are concerned about the accuracy of the data in a large data set, you could audit a small sample of the data in order to determine if a full audit is necessary. A representative sample is a subset of the full data set. The subset should contain observations that are taken from throughout the original data set so that the subset gives an accurate representation of the full data set. This chapter discusses two types of representative samples:

- systematic samples
- random samples.

Indexes can also make working with very large data sets easier. An index is a separate data structure that is associated with a data set, and that contains information about the specific location of observations in the data set according to the value of key variables. An index enables you to access a particular observation directly, without needing to read all of the observations that precede it in the data set. Indexes are useful in many instances, including WHERE and BY processing. This chapter discusses how to create and maintain both simple and composite indexes.

Objectives

In this chapter, you learn to
create a systematic sample from a known number of observations
create a systematic sample from an unknown number of observations
create a random sample with replacement
create a random sample without replacement
use indexes
create indexes in the DATA step
manage indexes with PROC DATASETS
manages indexes with PROC SQL
document and maintain indexes.

Prerequisites
Before beginning this chapter, you should complete the following chapter:
Chapter 1, “Performing Queries Using PROC SQL,” on page 3.

Creating a Systematic Sample from a Known Number of Observations

One type of representative sample that you might want to create is a systematic sample. A systematic sample contains observations that are chosen from the original data set at regular intervals. For example, a systematic sample could contain every hundredth observation of a very large data set.

To create a systematic sample from a data set that has a known number of observations, you use the POINT= option in the SET statement.

General form, SET statement with POINT= option:

```plaintext
SET data-set-name POINT= point-variable;
```

where

- `point-variable`
  - names a temporary numeric variable whose value is the observation number of the observation to be read
  - must be given a value before the execution of the SET statement
  - must be a variable and not a constant value.

The value of the variable that is named by the POINT= option should be an integer that is greater than zero and less than or equal to the number of observations in the SAS data set. SAS uses the value to point to a specific observation in the SET statement. You must assign this value within the program so that the POINT= variable has a value when the SET statement begins execution. Also, in order for SAS to read different observations into the sample, the value of the POINT= variable must change during execution of the DATA step.

Example

You can place the SET statement with the POINT= option inside a DO loop that will assign a different value to the POINT= variable on each iteration. In the following code
sample, the DO loop assigns a value to the variable pickit, which is used by the POINT= option to select every tenth observation from Sasuser.Sale2000. Notice that the following example is not a complete step.

```sas
do pickit=1 to 142 by 10;
   set sasuser.sale2000 point=pickit;
   output;
end;
```

Note: In general, samples are most useful when you are working with very large data sets. However, the sample data sets that are included in this chapter are relatively small. The basics of creating samples and indexes are the same no matter the size of the data set, and you can apply the techniques in this chapter to larger data sets.

By default, SAS reads a data set sequentially, beginning with the first observation. A DATA step stops processing when SAS reaches the end-of-file marker after reading the last observation in the data set.

The POINT= option uses direct-access read mode, which means that SAS reads only those observations that you direct it to read. In direct-access read mode, SAS does not detect the end-of-file marker. Therefore, when you use the POINT= option in a SET statement, you must use a STOP statement to prevent the DATA step from looping continuously.

The STOP statement causes SAS to stop processing the current DATA step immediately and resume processing statements after the end of the current DATA step.

General form, STOP statement:

```sas
STOP;
```

Example

The Sasuser.Revenue data set contains 142 observations. Suppose you want to select a ten-observation subset of the data set Sasuser.Revenue by reading every fifteenth observation. You can use the POINT= option in a SET statement inside a DO loop to create this sample.

```sas
data sasuser.subset;
    do pickit=1 to 142 by 15;
        set sasuser.revenue point=pickit;
        output;
    end;
    stop;
run;
```

```sas
proc print data=sasuser.subset;
run;
```

The program above creates the Sasuser.Subset data set, shown below.
Creating a Systematic Sample from an Unknown Number of Observations

Sometimes you might not know how many observations are in the original data set from which you want to create a systematic sample. In order to make a systematic sample, you need to know the total number of observations in the original data set so that you can choose observations that are evenly distributed from it.

You can use the NOBS= option in the SET statement to determine how many observations there are in a SAS data set.

General form, SET statement with NOBS= option:

```
SET SAS-data-set NOBS=variable;
```

where

```
variable
```

names a temporary numeric variable whose value is the number of observations in the input data set.

Note: If multiple data sets are listed in the SET statement, the value of the NOBS= variable is the total number of observations in all of the data sets that are listed.

The value of the NOBS= variable is assigned automatically during compilation when SAS reads the descriptor portion of the data file. Therefore, this value is available at any time during execution of the DATA step.

Note: The total that is used as a value for the NOBS= variable includes observations that have been marked for deletion but have not been physically removed from the data set.

You can use the NOBS= option in conjunction with the POINT= option to create a systematic sample of a data set if you do not know how many observations are in the data set.
Example

Suppose you want to create a systematic sample of the `Sasuser.Revenue` data set, and you do not know how many observations are in it. In the following example, `totobs` is assigned the value of the total number of observations in the data set `Sasuser.Revenue` during compilation. Then, `totobs` is used as the upper limit for the DO loop that controls how many observations are chosen for the systematic sample.

```sas
data sasuser.subset;
  do pickit=1 to totobs by 10;
    set sasuser.revenue point=pickit nobs=totobs;
    output;
  end;
  stop;
run;
```

The resulting `Sasuser.Subset` data set contains every tenth observation from the `Sasuser.Revenue` data set. When the program above is submitted, the DATA step iterates only once, and the DO loop iterates multiple times within the DATA step.

Creating a Random Sample with Replacement

Another type of representative sample that you might want to create is a random sample. A random sample contains observations that are chosen from the original data set on a random basis.

When you create a random sample with replacement, it is possible for one observation to appear in the sample multiple times. You can think of the original data set as a pool of possible observations that may be chosen for inclusion in the sample. For each observation in the sample data set, SAS chooses an observation randomly from the original pool, copies it to the sample data set, then replaces it in the pool.

Using the RANUNI Function

In order to create a random sample, you need to generate a random number. SAS provides several random number functions to generate random numbers from various distributions. One example of a random number function is the RANUNI function.

General form, RANUNI function:

```
RANUNI (seed)
```

where

- `seed` is a nonnegative integer with a value less than $2^{31} - 1$ (2,147,483,647).

The RANUNI function generates streams of random numbers from an initial starting point, called the `seed`. If you use a positive seed, you can always replicate the stream of random numbers by using the same DATA step. If you use 0 as the seed, the computer clock initializes the stream, and the stream of random numbers is not replicable.
The numbers that the RANUNI function returns are all between 0 and 1 (noninclusive). Examples of the type of number RANUNI returns include .01253689 and .95196500. If you use RANUNI in a DATA step that generates only one random number, RANUNI returns only the first number from the stream. If you use RANUNI in a DATA step that generates multiple numbers, such as in a DO loop, RANUNI will return a different random number each time the loop iterates.

Let's take a look at some examples.

**Example**

The following DATA step creates one observation with one variable named *varone* and assigns a random number to it. You can submit this DATA step multiple times, or in multiple SAS sessions, and *varone* will have the same value.

```sas
data random1;
  varone=ranuni(10);
run;
```

The following DATA step creates ten observations with one variable named *varone* and assigns a random number as a value for *varone*. You can submit this DATA step multiple times, or in multiple SAS sessions, and *varone* will have the same ten values.

```sas
data random2;
  do i=1 to 10 by 1;
    varone=ranuni(10);
    output;
  end;
run;
```

If you changed the seed value from 10 to a different value in either of the two DATA steps above, the values for *varone* would be different when you submitted the DATA step than it was when the seed value was 10. However, *varone* will have the same ten values each time you submit the DATA step with a constant seed value.

*Note:* For clarity and consistency in the sample programs, all remaining examples of the RANUNI function in this chapter will use a seed of 0.

**Using a Multiplier with the RANUNI Function**

By default, RANUNI generates numbers that are between 0 and 1. To increase the interval from which the random number is chosen, you use a multiplier on the RANUNI function. For example, if you want to generate a random number between 0 and 50, you use the following code:

```
ranuni(0)*50
```

You have seen that the RANUNI function generates a random number between 0 and 1. However, in order to create a random sample, you need to generate a random integer that will match one of the observation numbers in the original data set.

**Using the CEIL Function**

You can use the CEIL function in conjunction with the RANUNI function to generate a random integer.
Example

The following example creates a random integer between 1 and 50:

\[
\text{ceil}(\text{ranuni}(0) \times 50)
\]

Now that you have seen how to use the CEIL function in conjunction with the RANUNI function, let's take a look at how to use these functions in a DATA step to create a random sample. You use the CEIL and RANUNI functions together to generate a random integer that is assigned as a value for the variable to which the POINT= option points.

Example

In the following example, the CEIL and RANUNI functions are used together in the assignment statement for \textit{pickit}, which is the variable that is pointed to by the POINT= option. The NOBS= option assigns the total number of observations in the \textit{Sasuser.Revenue} data set as a value for the \textit{totobs} variable. The variable \textit{totobs} is then used as the multiplier in the CEIL function, so that for each iteration of the DO loop, every observation in \textit{Sasuser.Revenue} has an equal chance of being picked for inclusion in the sample.

```sas
data work.rsubset (drop=i sampsize);
   sampsize=10;
   do i=1 to sampsize;
      pickit=ceil(ranuni(0)*totobs);
      set sasuser.revenue point=pickit nobs=totobs;
      output;
   end;
   stop;
run;

proc print data=work.rsubset label;
   title 'A Random Sample with Replacement';
run;
```

Since the program uses a seed of 0 for the RANUNI function, the \textit{Work.Rsubset} data set will be different each time you submit this code. Here is an example of the possible output.
Creating a Random Sample without Replacement

You can also create a random sample without replacement. A sample without replacement cannot contain duplicate observations because after an observation is chosen from the original data set and output to the sample, it is programmatically excluded from being chosen again.

Example

You can use a DO WHILE loop to avoid replacement as you create your random sample. In the following example

- **Sasuser.Revenue** is the original data set.
- **sampsize** is the number of observations to read into the sample.
- **Work.Rsubset** is the data set that contains the random sample that you are creating.
- **obsleft** is the number of observations in the original data set that have not yet been considered for selection.
- **totobs** is the total number of observations in the original data set.
- **pickit** is the number of the observation to be read into the sample data set (if the RANUNI expression is true), and its starting value is 0.

With each iteration of the DO loop, **pickit** is incremented by 1. If the RANUNI expression is true, the observation that is indicated by the value of **pickit** is selected for the sample, and **sampsize** is decreased by 1. If the RANUNI expression is not true, the observation that is indicated by the value of **pickit** is not added to the sample. On each iteration of the loop, **obsleft** is decreased by 1 regardless of whether the observation is selected for the sample. The process ends when the value of **sampsize** is 0 and no additional observations are needed.

```
data work.rsubset(drop=obsleft sampsize);
sampsize=10;
```
Using Indexes

An index can help you quickly locate one or more particular observations that you want to read. An index is an optional file that you can create for a SAS data set in order to specify the location of observations based on values of one or more key variables. Indexes can provide direct access to observations in SAS data sets to:

- yield faster access to small subsets of observations for WHERE processing
- return observations in sorted order for BY processing
- perform table lookup operations

A Random Sample without Replacement

<table>
<thead>
<tr>
<th>Obs</th>
<th>Origin</th>
<th>Dest</th>
<th>FlightID</th>
<th>Date</th>
<th>Rev1st</th>
<th>RevBusiness</th>
<th>RevEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CBR</td>
<td>WLG</td>
<td>IA10500</td>
<td>28DEC1999</td>
<td>16092</td>
<td>21164</td>
<td>27324</td>
</tr>
<tr>
<td>2</td>
<td>CBR</td>
<td>WLG</td>
<td>IA10501</td>
<td>11DEC1999</td>
<td>13112</td>
<td>15260</td>
<td>23760</td>
</tr>
<tr>
<td>3</td>
<td>CCU</td>
<td>HKG</td>
<td>IA09901</td>
<td>25DEC1999</td>
<td>16272</td>
<td>19404</td>
<td>27225</td>
</tr>
<tr>
<td>4</td>
<td>CCU</td>
<td>SIN</td>
<td>IA09300</td>
<td>22DEC1999</td>
<td>18575</td>
<td>25350</td>
<td>29393</td>
</tr>
<tr>
<td>5</td>
<td>JED</td>
<td>LHR</td>
<td>IA08200</td>
<td>08DEC1999</td>
<td>23161</td>
<td>44820</td>
<td>51435</td>
</tr>
<tr>
<td>6</td>
<td>LHR</td>
<td>JNB</td>
<td>IA08301</td>
<td>09DEC1999</td>
<td>41940</td>
<td>76224</td>
<td>111466</td>
</tr>
<tr>
<td>7</td>
<td>LHR</td>
<td>RDU</td>
<td>IA02000</td>
<td>07DEC1999</td>
<td>20600</td>
<td>30520</td>
<td>73650</td>
</tr>
<tr>
<td>8</td>
<td>RDU</td>
<td>FRA</td>
<td>IA00301</td>
<td>03DEC1999</td>
<td>24654</td>
<td>31226</td>
<td>72540</td>
</tr>
<tr>
<td>9</td>
<td>RDU</td>
<td>LHR</td>
<td>IA00101</td>
<td>29DEC1999</td>
<td>20800</td>
<td>25070</td>
<td>75933</td>
</tr>
<tr>
<td>10</td>
<td>SFO</td>
<td>HNL</td>
<td>IA02900</td>
<td>26DEC1999</td>
<td>11856</td>
<td>16152</td>
<td>48216</td>
</tr>
</tbody>
</table>
Creating Samples and Indexes  △  Types of Indexes  461

- join observations
- modify observations.

Without an index, SAS accesses observations sequentially, in the order in which they are stored in a data set. For example, if you want to access the observation in the sample SAS data set shown below that has a value of Smith for the variable Name, SAS begins with the first observation and reads through each one until it reaches the observation that satisfies the condition.

### SAS Data Set

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Sex</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>09JAN2000</td>
<td>X</td>
<td>34</td>
</tr>
<tr>
<td>Baker</td>
<td>14OCT2001</td>
<td>X</td>
<td>54</td>
</tr>
<tr>
<td>Davis</td>
<td>30MAR2000</td>
<td>Y</td>
<td>49</td>
</tr>
<tr>
<td>Edwards</td>
<td>28JUN2002</td>
<td>X</td>
<td>52</td>
</tr>
<tr>
<td>Smith</td>
<td>15JAN2000</td>
<td>Y</td>
<td>62</td>
</tr>
<tr>
<td>Yates</td>
<td>04AUG2002</td>
<td>X</td>
<td>59</td>
</tr>
</tbody>
</table>

An index stores values in ascending value order for a specific variable or variables and includes information about the location of those values within observations in the data file. That is, an index is composed of value/identifier pairs that enable you to locate an observation by value. For example, if you create an index on the sample SAS data set shown below based on the variable Name, SAS uses the index to find the observation that has a value of Smith for Name directly without having to read all the prior observations.

### SAS Data Set

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Sex</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>09JAN2000</td>
<td>X</td>
<td>34</td>
</tr>
<tr>
<td>Baker</td>
<td>14OCT2001</td>
<td>X</td>
<td>54</td>
</tr>
<tr>
<td>Davis</td>
<td>30MAR2000</td>
<td>Y</td>
<td>49</td>
</tr>
<tr>
<td>Edwards</td>
<td>28JUN2002</td>
<td>X</td>
<td>52</td>
</tr>
<tr>
<td>Smith</td>
<td>15JAN2000</td>
<td>Y</td>
<td>62</td>
</tr>
<tr>
<td>Yates</td>
<td>04AUG2002</td>
<td>X</td>
<td>59</td>
</tr>
</tbody>
</table>

### Types of Indexes

You can create two types of indexes:

- a simple index
- a composite index.

A **simple index** consists of the values of one key variable, which can be character or numeric. When you create a simple index, SAS assigns the name of the key variable as the name of the index.

A **composite index** consists of the values of multiple key variables, which can be character, numeric, or a combination. The values of these key variables are concatenated to form a single value. For example, if an index is built on the key variables Lastname and Firstname, a value for the index is composed of the value for Lastname followed by the value for Firstname. When you create a composite index, you must specify a **unique** index name that is not the name of any existing variable or index in the data set.

Often, only the first variable of a composite index is used. For example, you could use the composite index specified in the example above (Lastname plus Firstname) for
Creating Indexes in the DATA Step

To create an index at the same time that you create a data set, use the INDEX= data set option in the DATA statement.

General form, DATA statement with the INDEX= option:

```sas
DATA SAS-data-file-name (INDEX=
(index-specification-1)</UNIQUE><...index-specification-n>
</UNIQUE>);
```

where

- **SAS-data-file-name**
  - is a valid SAS data set name.
- **index-specification**
  - for a simple index is the name of the key variable.
  - for a composite index is (index-name=(variable-1...variable-n)).
- **UNIQUE**
  - option specifies that values for the key variable must be unique for each observation.

**Note:** SAS stores the name of a composite index exactly as you specify it in the INDEX= option. Therefore, if you want the name of your index to begin with a capital letter, you must specify the name with an initial capital letter in the INDEX= option.

You can create multiple indexes on a single SAS data set. However, keep in mind that creating and storing indexes does use system resources. Therefore, you should create indexes only on variables that are commonly used in a WHERE condition or on variables that are used to combine SAS data sets.

**Note:** You can create an index on a SAS data file but not on a SAS data view.

The UNIQUE option guarantees that values for the key variable or the combination of a composite group of variables remain unique for every observation in the data set. In an existing data set, if the variable(s) on which you attempt to create a unique index has duplicate values, the index is not created. Similarly, if an update tries to add a record with a duplicate value for the index variable to that data set, the update is rejected. You will see examples of updating and maintaining indexes later in this chapter.

**Examples**

The following example creates a simple index on the *Simple* data set. The index is named *Division*, and it contains values of the *Division* variable.

```sas
data simple (index=(division));
set sasuser.empdata;
```
The following example creates two simple indexes on the Simple2 data set. The first index is named Division, and it contains values of the Division variable. The second index is called EmpID, and it contains unique values of the EmpID variable.

```sas
data simple2 (index=(division empid/unique));
  set sasuser.empdata;
run;
```

The following example creates a composite index on the Composite data set. The index is named Empdiv, and it contains concatenated values of the Division variable and the EmpID variable.

```sas
data composite (index=(Empdiv=(division empid)));
  set sasuser.empdata;
run;
```

When you create or use an index, you might want to verify that it has been created or used correctly. To display information in the SAS log concerning index creation or index usage, set the value of the MSGLEVEL= system option to I.

```
General form, MSGLEVEL= system option:

    OPTIONS MSGLEVEL= N|I;
```

where

N  prints notes, warnings, and error messages only. This is the default.

I  prints additional notes or INFO messages pertaining to index usage, merge processing, and sort utilities along with standard notes, warnings, and error messages.

---

**Example**

The following code sets the MSGLEVEL= system option to I and creates the Sasuser.Sale2000 data set with two indexes:

```sas
options msglevel=i;
data sasuser.sale2000(index=(origin flightdate=(flightid date)/unique));
infile sale2000 dsd;
format date date9.;
run;
```

Here are the messages that are written to the SAS log when the program above is submitted.
### Determining Whether SAS Is Using an Index

It is not always possible or more efficient for SAS to use an existing index to access specific observations directly. An index is *not used*
- with a subsetting IF statement in a DATA step
- with particular WHERE expressions
- if SAS determines it is more efficient to read the data sequentially.

**Example**

You can use the MSGLEVEL= option to determine whether SAS is using an index. The following SAS log messages show examples of the INFO messages that indicate whether an index was used.

#### Table 13.2 SAS Log

```sas
6 options msglevel=i;
7
8 proc print data=sasuser.revenue;
9    where flightid ne 'IA11200';
INFO: Index FlightID not used. Increasing bufno to 3 may help.
```

#### Table 13.3 SAS Log

```sas
11 options msglevel=i;
12
13 data somflights;
14   set sasuser.revenue;
15   where flightid > 'IA11200';
INFO: Index FlightID selected for WHERE clause optimization.
```

### Managing Indexes with PROC DATASETS

You have seen how to create an index at the same time that you create a data set. You can also create an index on an existing data set, or delete an index from a data set.
One way to accomplish either of these tasks is to rebuild the data set. However, rebuilding the data set is not the most efficient method for managing indexes.

You can use the DATASETS procedure to manage indexes on an existing data set. This uses fewer resources than it would to rebuild the data set. You use the MODIFY statement with the INDEX CREATE statement to create indexes on a data set. You use the MODIFY statement with the INDEX DELETE statement to delete indexes from a data set. You can also use the INDEX CREATE statement and the INDEX DELETE statement in the same step.

General form, PROC DATASETS to create and delete an index:

```
PROC DATASETS LIBRARY= libref <NOLIST>;  
   MODIFY SAS-data-set-name;  
   INDEX DELETE index-name;  
   INDEX CREATE index-specification;  
QUIT;
```

where

- `libref` points to the SAS library that contains `SAS-data-set-name`.

- `NOLIST` option suppresses the printing of the directory of SAS files in the SAS log and as ODS output.

- `index-name` is the name of an existing index to be deleted.

- `index-specification` for a simple index is the name of the key variable.

- `index-specification` for a composite index is `index-name=(variable-1...variable-n)`.

The INDEX CREATE statement in PROC DATASETS cannot be used if the index to be created already exists. In this case, you must delete the existing index of the same name, then create the new index.

**Tip:** PROC DATASETS executes statements in order. Therefore, if you are deleting and creating indexes in the same step, you should delete indexes first so that the newly created indexes can reuse space of the deleted indexes.

### Example

The following example creates an index named `Origin` on the `Sasuser.Sale2000` data set. `Origin` is a simple index that is based on the key variable `Origin`.

```
proc datasets library=sasuser nolist;  
   modify sale2000;  
   index create origin;  
quit;
```

The following example first deletes the `Origin` index from the `Sasuser.Sale2000` data set, then creates two new indexes on the `Sasuser.Sale2000` data set. `FlightID` is a simple index that is based on the values of the key variable `FlightID`. `Fromto` is a composite index that is based on the concatenated values of the key variables `Origin` and `Dest`.  

---

**Creating Samples and Indexes △ Example 465**
Managing Indexes with PROC SQL

You can also create indexes on or delete indexes from an existing data set within a PROC SQL step. The CREATE INDEX statement enables you to create an index on a data set. The DROP INDEX statement enables you to delete an index from a data set.

**General form, PROC SQL to create and delete an index:**

```
PROC SQL;
   CREATE [UNIQUE] INDEX index-name
       ON table-name(column-name-1,...,column-name-n);
   DROP INDEX index-name FROM table-name;
QUIT;
```

where

- `index-name` is the same as `column-name-1` if the index is based on the values of one column only.
- `index-name` is not the same as any `column-name` if the index is based on multiple columns.
- `table-name` is the name of the data set to which `index-name` is associated.

**Example**

The following example creates a simple index named **Origin** on the Sasuser.Sale2000 data set. The index is based on the values of the **Origin** column.

```
proc sql;
   create index origin on sasuser.sale2000(origin);
quit;
```

The following example deletes the **Origin** index from the Sasuser.Sale2000 data set and creates a new index named **Tofrom** that is based on the concatenation of the values from the columns **Origin** and **Dest**:

```
proc sql;
   create index Tofrom
       on sasuser.sale2000(origin, dest);
   drop index origin from sasuser.sale2000;
quit;
```
Indexes are stored in the same SAS data library as the data set that they index, but in a separate SAS file from the data set. Index files have the same name as the associated data file, and have a member type of INDEX. There is only one index file per data set; all indexes for a data set are stored together in a single file.

The following image shows the relationship of SAS data set files and SAS index files in a Windows operating environment. Notice that although they have different file extensions, the index files have the same name as the data set with which they are associated. Also, notice that each index file can contain one or more indexes, and that different index files can contain indexes with identical names.

Note: Although index files are stored in the same location as the data sets to which they are associated
- index files do not appear in the SAS Explorer window
- index files do not appear as separate files in z/OS (OS/390) operating environment file lists.

Sometimes, you might want to view a list of the indexes that exist for a data set. You might also want to see information about the indexes such as whether they are unique, and what key variables they use. Let’s take a look at some ways to document indexes.

Information about indexes is stored in the descriptor portion of the data set. You can use either the CONTENTS procedure or the CONTENTS statement in PROC DATASETS to list information from the descriptor portion of a data set.

Output from the CONTENTS procedure or from the CONTENTS statement in PROC DATASETS contains the following information about the data set:
- general and summary information
- engine/host dependent information
- alphabetic list of variables and attributes
- alphabetic list of integrity constraints
- alphabetic list of indexes and attributes.

General form, PROC CONTENTS:

```
PROC CONTENTS DATA=<libref.>SAS-data-set-name;
RUN;
```

where
- `SAS-data-set-name` specifies the data set for which the information will be listed.
General form, PROC DATASETS with the CONTENTS statement:

```
PROC DATASETS <LIBRARY=libref> <NOLIST>;
    CONTENTS DATA=<libref.:SAS-data-set-name>;
QUIT;
```

where

- `SAS-data-set-name` specifies the data set for which the information will be listed.
- `NOLIST` option suppresses the printing of the directory of SAS files in the SAS log and as ODS output.

Note: If you use the `LIBRARY=` option, you do not need to specify a libref in the `DATA=` option. Likewise, if you specify a `libref` in the `DATA=` option, you do not need to use the `LIBRARY=` option.

Example

The following example prints information about the `Sasuser.Sale2000` data set. Notice that the library is specified in the `LIBRARY=` option of the PROC DATASETS statement.

```
proc datasets library=sasuser nolist;
    contents data=sale2000;
quit;
```

The following example also prints information about the `Sasuser.Sale2000` data set. Notice that the library is specified in the CONTENTS statement.

```
proc datasets nolist;
    contents data=sasuser.sale2000;
quit;
```

The following example also prints information about the `Sasuser.Sale2000` data set:

```
proc contents data=sasuser.sale2000;
run;
```

The PROC DATASETS and PROC CONTENTS output from these programs is identical. The last piece of information printed in each set of output is a list of the indexes that have been created for `Sasuser.Sale2000`, as shown below.

<table>
<thead>
<tr>
<th>Alphabetic List of Indexes and Attributes</th>
<th># of Unique Values</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td># Index</td>
<td>Unique Option</td>
<td></td>
</tr>
<tr>
<td>1 Origin</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>2 FlightDate</td>
<td>YES</td>
<td>26920</td>
</tr>
<tr>
<td></td>
<td>FlightID Date</td>
<td></td>
</tr>
</tbody>
</table>
You can also use either of these methods to list information about an entire SAS library rather than an individual data set. To list the contents of all files in a SAS data library with either PROC CONTENTS or with the CONTENTS statement in PROC DATASETS, you specify the keyword _ALL_ in the DATA= option.

**Example**

The following example prints information about all of the files in the Work data library:

```sas
proc contents data=work._all_;
run;
```

The following example also prints information about all of the files in the Work data library:

```sas
proc datasets library=work nolist;
  contents data=_all_;
quit;
```

Remember that indexes are stored in a separate SAS file. When you perform maintenance tasks on a data set, there might be resulting effects on the index file. If you alter the variables or values within a data set, there might be a resulting effect on the value/identifier pairs within a particular index.

The following table describes the effects on an index or an index file that result from several common maintenance tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add observation(s) to data set</td>
<td>Value/identifier pairs are added to index(es).</td>
</tr>
<tr>
<td>Delete observation(s) from data set</td>
<td>Value/identifier pairs are deleted from index(es).</td>
</tr>
<tr>
<td>Update observation(s) in data set</td>
<td>Value/identifier pairs are updated in index(es).</td>
</tr>
<tr>
<td>Delete data set</td>
<td>The index file is deleted.</td>
</tr>
<tr>
<td>Rebuild data set with DATA step</td>
<td>The index file is deleted.</td>
</tr>
<tr>
<td>Sort the data in place with the FORCE option in PROC SORT</td>
<td>The index file is deleted.</td>
</tr>
</tbody>
</table>

Let’s take a look at some of the other common tasks that you might perform on your data sets, as well as the actions that SAS performs on the index files as a result.

**Copying Data Sets**

You might want to copy an indexed data set to a new location. You can copy a data set with the COPY statement in a PROC DATASETS step. When you use the COPY statement to copy a data set that has an index associated with it, a new index file is automatically created for the new data file.
General form, PROC DATASETS with the COPY statement:

```sas
PROC DATASETS LIBRARY=old-libref <NOLIST>;
   COPY OUT=new-libref;
   SELECT SAS-data-set-name;
QUIT;
```

where

- `old-libref` names the library from which the data set will be copied.
- `new-libref` names the library to which the data set will be copied.
- `SAS-data-set-name` names the data set that will be copied.

You can also use the `COPY procedure` to copy data sets to a new location. Generally, PROC COPY functions the same as the COPY statement in the DATASETS procedure. When you use PROC COPY to copy a data set that has an index associated with it, a new index file is automatically created for the new data file. If you use the MOVE option in the COPY procedure, the index file is deleted from the original location and rebuilt in the new location.

General form, PROC COPY step:

```sas
PROC COPY OUT=new-libref IN=old-libref
   <MOVE>;
   SELECT SAS-data-set-name(s);
RUN;
QUIT;
```

where

- `old-libref` names the library from which the data set will be copied.
- `new-libref` names the library to which the data set will be copied.
- `SAS-data-set-name` names the data set or data sets that will be copied.

Examples

The following programs produce the same result. Both programs copy the `Sale2000` data set from the `Sasuser` library and place it in the `Work` library. Likewise, both of these programs cause a new index file to be created for `Work.Sale2000` that contains all indexes that exist in `Sasuser.Sale2000`.

```sas
proc datasets library=sasuser nolist;
   copy out=work;
```
Renaming Data Sets

Another common task is to rename an indexed data set. To preserve the index, you can use the `CHANGE statement` in PROC DATASETS to rename a data set. The index file will be automatically renamed as well.

General form, PROC DATASETS with the `CHANGE` statement:

```
PROC DATASETS LIBRARY=libref <NOLIST>;
   CHANGE old-data-set-name = new-data-set-name;
QUIT;
```

where

- `libref` names the SAS library where the data set is stored.
- `old-data-set-name` is the current name of the data set.
- `new-data-set-name` is the new name of the data set.

Example

The following example copies the `Revenue` data set from `Sasuser` into `Work`, then renames the `Work.Revenue` data set to `Work.Income`. The index file that is associated with `Work.Revenue` is also renamed to `Work.Income`.

```
proc copy out=work in=sasuser;
   select revenue;
run;

proc datasets library=work nolist;
   change revenue=income;
quit;
```

Renaming Variables

You have seen how to use PROC DATASETS to rename an indexed data set. Similarly, you might want to rename one or more variables within an indexed data set. In order to preserve any indexes that are associated with the data set, you can use the `RENAME statement` in the DATASETS procedure to rename variables.
General form, PROC DATASETS with the RENAME statement:

```
PROC DATASETS LIBRARY=libref <NOLIST>;
   MODIFY SAS-data-set-name;
   RENAME old-var-name-1 = new-var-name-1
      <...old-var-name-n = new-var-name-n>;
QUIT;
```

where

- `libref` names the SAS library where the data set is stored.
- `SAS-data-set-name` is the name of the data set that contains the variables to be renamed.
- `old-var-name` is the original variable name.
- `new-var-name` is the new name to be assigned to the variable.

When you use the RENAME statement to change the name of a variable for which there is a simple index, the statement also renames the index. If the variable that you are renaming is used in a composite index, the composite index automatically references the new variable name. However, if you attempt to rename a variable to a name that has already been used for a composite index, you will receive an error message.

**Example**

The following example renames the variable `FlightID` as `FlightNum` in the `Work.Income` data set. If a simple index exists that is named `FlightID`, the index will be renamed `FlightNum`.

```
proc datasets library=work nolist;
   modify income;
      rename flightid=FlightNum;
quit;
```
Summary

This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Creating a Systematic Sample from a Known Number of Observations

Sometimes you might want to create a representative sample of a large data set. One type of representative sample, called a systematic sample, contains observations that are chosen from the original data set at regular intervals. You can use the POINT= option in the SET statement to make SAS read a specific observation into the sample. Since SAS uses direct-access read mode with the POINT= option, you must use a STOP statement to prevent the DATA step from looping continuously.

Creating a Systematic Sample from an Unknown Number of Observations

You might want to create a systematic sample from a data set that contains an unknown number of observations. In order to be sure that your sample observations are chosen from regular intervals across the entire original data set, you need to know how many observations are in the data set. You can use the NOBS= option in the SET statement to determine how many observations are in the input data set. You can use the NOBS= option in conjunction with the POINT= option to direct SAS to read specific observations that will form a systematic sample.

Creating a Random Sample with Replacement

Another type of representative sample that you might want to create is a random sample, in which observations are chosen randomly from the original data set. You can use the RANUNI function in conjunction with the CEIL function to generate a random integer. With a random integer, you can direct SAS to read a specific (but random) observation into the sample. When you create a random sample with replacement, each observation in the original data set has an equal chance of being chosen for inclusion in the sample each time SAS chooses an observation. That is, in a random sample with replacement, one observation may be chosen from the original data set and included in the sample multiple times.

Creating a Random Sample without Replacement

You can also create a random sample without replacement. This means that once an observation has been included in the sample it is no longer eligible to be chosen again. You can use a DO WHILE loop to prevent replacement in your random samples.

Using Indexes

An index is a SAS file that is associated with a data set and that contains information about the location and the values of key variables in the data set. Indexes
enable SAS to directly access specific observations rather than having to read all 
observations sequentially. An index can be simple or composite.

Creating Indexes in the DATA Step

You can create an index at the same time that you create a data set by using the 
INDEX= option in the DATA statement. Both simple and composite indexes can be 
unique, if there are no duplicate values for any key variable in the data set. You can 
create multiple indexes on one data set. You can use the MSGLEVEL= system option to 
write informational messages to the SAS log that pertain to indexes. Indexes can 
 improve the efficiency of SAS, but there are certain instances where indexes will not 
 improve efficiency and therefore will not be used.

Managing Indexes with PROC DATASETS and PROC SQL

You can use the INDEX CREATE statement or the INDEX DELETE statement in 
PROC DATASETS to create an index on or delete an index from an existing data set. 
Using PROC DATASETS to manage indexes uses less system resources than it would to 
rebuild the data set and update indexes in the DATA step. If you want to delete an 
index and create an index in the same PROC DATASETS step, you should delete the 
old index before you create the new index so that SAS can reuse space from the deleted 
index. You can also use PROC SQL to create an index on or delete an index from an 
existing data set.

Documenting and Maintaining Indexes

All indexes that are created for a particular data set are stored in one file in the 
same SAS data library as the data set. You can use PROC CONTENTS to print a list of 
all indexes that exist for a data set, along with other information about the data set. 
The CONTENTS statement of the PROC DATASETS step can generate the same list of 
 indexes and other information about a data set.

Many of the maintenance tasks that you perform on your data sets will affect the 
index file that is associated with the data set. When you copy a data set with the COPY 
statement in PROC DATASETS, the index file is reconstructed for you. When you 
rename a data set or rename a variable with PROC DATASETS, the index file is 
 automatically updated to reflect this change.

Syntax

```
DATA SAS-data-set-name;
    point-variable=CEIL(RANUNI(seed) *nobs-variable);
    SET SAS-data-set-name POINT=point-variable NOBS=nobs-variable;
    STOP;
RUN;
OPTIONS MSGLEVEL= N | I;

DATA SAS-data-file-name (INDEX= 
    (index-specification-1/UNIQUE><....index-specification-n/UNIQUE>>));
    SET SAS-data-set-name;
RUN;
```
PROC DATASETS LIBRARY=libref <NOLIST>;
  MODIFY SAS-data-set-name;
  INDEX DELETE index-name;
  INDEX CREATE index-specification;
QUIT;

PROC SQL;
  CREATE <UNIQUE> INDEX index-name
  ON table-name(column-name-1,...,column-name-n);
  DROP INDEX index-name FROM table-name;
QUIT;

PROC CONTENTS DATA=<libref.>SAS-data-set-name;
RUN;

PROC DATASETS <LIBRARY=libref> <NOLIST>;
  CONTENTS DATA=<libref.>SAS-data-set-name;
QUIT;

PROC DATASETS LIBRARY=old-libref <NOLIST>;
  COPY OUT=new-libref;
  SELECT SAS-data-set-name;
QUIT;

PROC COPY OUT=new-libref IN=old-libref <MOVE>;
  SELECT SAS-data-set-name(s);
RUN;
QUIT;

PROC DATASETS LIBRARY=libref <NOLIST>;
  CHANGE old-data-set-name = new-data-set-name;
QUIT;

PROC DATASETS LIBRARY=libref <NOLIST>;
  MODIFY SAS-data-set-name;
  RENAME old-var-name-1 = new-var-name-1
  <...old-var-name-n = new-var-name-n>;
QUIT;

Sample Programs

Creating a Systematic Sample from a Known Number of Observations

data sasuser.subset;
  do pickit=1 to 142 by 15;
set sasuser.revenue point=pickit;
output;
end;
stop;
run;

Creating a Systematic Sample from an Unknown Number of Observations

data sasuser.subset;
   do pickit=1 to totobs by 10;
      set sasuser.revenue point=pickit
         nobs=totobs;
      output;
   end;
stop;
run;

Creating a Random Sample with Replacement

data work.rsubset (drop=i sampsize);
sampsize=10;
do i=1 to sampsize;
   pickit=ceil(ranuni(0)*totobs);
   set sasuser.revenue point=pickit
      nobs=totobs;
   output;
end;
stop;
run;
proc print data=work.rsubset label;
   title 'A Random Sample with Replacement';
run;

Creating a Random Sample without Replacement

data work.rsubset(drop=obsleft sampsize);
sampsize=10;
obsleft=totobs;
do while(sampsize>0);
   pickit+1;
   if ranuni(0)<sampsize/obsleft then do;
      set sasuser.revenue point=pickit
         nobs=totobs;
      output;
      sampsize=sampsize-1;
   end;
   obsleft=obsleft-1;
end;
stop;
run;
proc print data=work.rsubset heading=h label;
   title 'A Random Sample without Replacement';
run;
Creating an Index in the DATA Step

```sas
options msglevel=i;
data sasuser.sale2000(index=(origin FlightDate= (flightid date)/unique));
infile 'sale2000.dat';
RevEcon dollar15.2 SaleMon $7. CargoWgt 8./ RevCargo dollar15.2;
run;
```

Managing Indexes with PROC DATASETS

```sas
proc datasets library=sasuser nolist;
modify sale2000;
index delete origin;
index create flightid;
index create Tofrom=(origin dest);
quit;
```

Managing Indexes with PROC SQL

```sas
proc sql;
create index Tofrom on 
sasuser.sale2000(origin, dest);
drop index origin from sasuser.sale2000;
quit;
```

Points to Remember

- If you use direct-access read mode to create a representative sample of a data set, you need to use a STOP statement to prevent the DATA step from looping continuously.
- An index can enable SAS to more efficiently access specific observations of a data set, but indexes use system resources and should only be created on variables that are commonly used in a WHERE condition or on variables that are used to combine SAS data sets.
- An index is associated with a data set but is stored as a separate file. You can use PROC DATASETS, the CONTENTS statement, or the SAS Explorer to view a list of the indexes that exist for a data set. It is especially useful for you to view this information after you have performed maintenance tasks on your data set in order to be sure that the index file has been maintained.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.
1. The variable that is created by the POINT= option is assigned a value
   a. automatically during compilation of the DATA step.
   b. automatically during execution of the DATA step.
   c. during compilation of the DATA step, by program statements.
   d. during execution of the DATA step, by program statements.

2. Which of the following programs correctly creates a systematic sample from a data set with an unknown number of observations and outputs these sample observations to a data set named Sample?
   a. data sample;
      set sasuser.sale2000 point=thisone nobs=totnum;
      output;
      stop;
   run;
   b. data sample;
      do thisone=100 to totnum by 100;
      set sasuser.sale2000 point=thisone nobs=totnum;
      output;
      end;
      stop;
   run;
   c. data sample;
      do thisone=100 to 1000 by 100;
      set sasuser.sale2000 point=thisone;
      output;
      end;
      stop;
   run;
   d. data sample;
      do thisone=100 to totnum by 100;
      set sasuser.sale2000 point=thisone nobs=totnum;
      end;
   run;

3. Which of the following expressions will generate a random integer between 1 and 50?
   a. ceil(ranuni(50))
   b. ranuni(50)
   c. ceil(ranuni(0)*50)
   d. ceil(ranuni(0))*50

4. An index
   a. is an optional file that is associated with a data set.
   b. provides direct access to specific observations of a data set, based on the value of one or more key variables.
   c. can be classified as simple or composite, either of which can consist of unique values.
   d. all of the above
5 Which of the following correctly creates a data set named *Flights* from the *Sasuser:Revenue* data set, creates a composite index named *Fromto* that is based on the values of *Origin* and *Dest*, and prints informational messages about the index to the SAS log?

a. options msglevel=i;
   data flights index=(Fromto=origin dest);
   set sasuser.revenue;
   run;

b. options msglevel=n;
   data flights (index=(Fromto=origin dest));
   set sasuser.revenue;
   run;

c. options msglevel=i;
   data flights (index=(Fromto=(origin dest)));
   set sasuser.revenue;
   run;

d. options msglevel=n;
   data flights (index=Fromto);
   set sasuser.revenue;
   run;

6 Which of the following is true?

a. When you add observations to a data set, the index(es) are automatically updated with additional value/identifier pairs.

b. When you rename a variable that is used as the key variable in a simple index, you must re-create the index.

c. When you delete a data set, the index file remains until you delete it as well.

d. When you copy a data set with the COPY statement, you must also copy the index file in another step.

7 To create an index on an existing data set, you use

a. PROC DATASETS.

b. PROC SQL.

c. the DATA step with the INDEX= option, to rebuild the data set.

d. any of the above

8 Which of the following correctly creates a simple index named *Origin* on the *Revenue* data set?

a. proc sql;
   create index origin on revenue(origin);
   quit;

b. proc sql;
   modify revenue;
   index=origin;
   quit;

c. proc sql data=revenue;
   create index origin;
   quit;

d. proc sql;
   index=origin on revenue;
   quit;
9 To view a list of the indexes that are associated with a data set, you use
   a PROC COPY or the COPY statement in PROC DATASETS.
   b PROC CONTENTS or the CONTENTS statement in PROC DATASETS.
   c the MSGLEVEL= system option and a PROC PRINT step.
   d any of the above

10 Suppose that the `Sasuser.Revenue` data set has a simple index named `FlightID`.
    For which of the following programs will the index be used?
    a proc print data=sasuser.revenue;
       where flightid ne ‘IA11200’;
       run;
    b data someflights;
       set sasuser.revenue;
       where flightid > ‘IA11200’;
       run;
    c data someflights;
       set sasuser.revenue;
       if flightid > ‘IA11200’;
       run;
    d proc print data=sasuser.revenue;
       where origin='RDU' or flightid='IA03400';
       run;
Overview 482
  Introduction 482
  Objectives 482
Using a FILENAME Statement 483
  Example 483
Using an INFILE Statement 486
  Example 486
  Assigning the Names of the Files to Be Read 487
  Example 488
  Using the COMPRESS Function 488
  Example 489
  Using the END= Option 490
  Example 491
  Using Date Functions 491
  Example 491
  Using the INTNX Function 493
  Example 494
Appending SAS Data Sets 494
  Example 495
  Using the FORCE Option 496
  Example 497
  Appending Variables with Different Lengths 498
  Example 498
  Appending Variables with Different Types 500
  Example 500
Additional Features 502
  Storing Raw Data Filenames in a SAS Data Set 502
  Storing Raw Data Filenames in an External File 502
Summary 504
  Text Summary 504
  Using a FILENAME Statement 504
  Using an INFILE Statement 504
  Appending SAS Data Sets 504
  Additional Features 505
Syntax 505
  Combining Raw Data Files Using a FILENAME Statement 505
  Combining Raw Data Files Using an INFILE Statement 505
  Combining SAS Data Sets Using PROC APPEND 505
Sample Programs 506
  Combining Raw Data Files Using a FILENAME Statement 506
  Combining Raw Data Files Using an INFILE Statement 506
Overview

Introduction

Combining data vertically refers to the process of concatenating or interleaving data. In some cases the data may be in SAS data sets. In other cases the data may be stored in raw data files.

In this chapter you learn how to create a SAS data set by concatenating multiple raw data files using the FILENAME and INFILE statements. You also learn how to concatenate SAS data sets using PROC APPEND.

Objectives

In this chapter you learn to

- create a SAS data set from multiple raw data files using a FILENAME statement
- create a SAS data set from multiple raw data files using an INFILE statement with the FILEVAR= option
- append SAS data sets using the APPEND procedure.
Using a FILENAME Statement

You already know that you can use a FILENAME statement to associate a fileref with a single raw data file. You can also use a FILENAME statement to concatenate raw data files by assigning a single fileref to the raw data files that you want to combine.

General form, FILENAME statement:

```
FILENAME fileref ('external-file1' 'external-file2' ...'external-filen');
```

where

- `fileref` is any SAS name that is eight characters or fewer.
- `external-file` is the physical name of an external file. The physical name is the name that is recognized by the operating environment.

**CAUTION:**

All of the file specifications must be enclosed in one set of parentheses.

When the fileref is specified in an INFILE statement, each raw data file that has been referenced can be sequentially read into a data set using an INPUT statement.

*Tip:* If you are not familiar with the content and structure of your raw data files, you can use PROC FSLIST to view them.

**Example**

In the following program, the FILENAME statement creates the fileref `Qtr1`, which references the raw data files `Month1.dat`, `Month2.dat`, and `Month3.dat`. The files are stored in the `C:\Sasuser` directory in the Windows operating environment. In the DATA step, the INFILE statement identifies the fileref, and the INPUT statement describes the data, just as if `Qtr1` referenced a single raw data file.

```
filename qtr1 ('c:\sasuser\month1.dat''c:\sasuser\month2.dat' 'c:\sasuser\month3.dat');
data work.firstqtr;
infile qtr1;
input Flight $ Origin $ Dest $
    Date : date9. RevCargo : comma15.2;
run;
```
Table 14.1  Raw Data File Month1.dat (first five records)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IA10200</td>
<td>SYD</td>
<td>HKG</td>
<td>01JAN2000</td>
</tr>
<tr>
<td>IA10201</td>
<td>SYD</td>
<td>HKG</td>
<td>01JAN2000</td>
</tr>
<tr>
<td>IA10300</td>
<td>SYD</td>
<td>CBR</td>
<td>01JAN2000</td>
</tr>
<tr>
<td>IA10301</td>
<td>SYD</td>
<td>CBR</td>
<td>01JAN2000</td>
</tr>
<tr>
<td>IA10302</td>
<td>SYD</td>
<td>CBR</td>
<td>01JAN2000</td>
</tr>
</tbody>
</table>

Table 14.2  Raw Data File Month2.dat (first five records)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IA10200</td>
<td>SYD</td>
<td>HKG</td>
<td>01FEB2000</td>
</tr>
<tr>
<td>IA10201</td>
<td>SYD</td>
<td>HKG</td>
<td>01FEB2000</td>
</tr>
<tr>
<td>IA10300</td>
<td>SYD</td>
<td>CBR</td>
<td>01FEB2000</td>
</tr>
<tr>
<td>IA10301</td>
<td>SYD</td>
<td>CBR</td>
<td>01FEB2000</td>
</tr>
<tr>
<td>IA10302</td>
<td>SYD</td>
<td>CBR</td>
<td>01FEB2000</td>
</tr>
</tbody>
</table>

Table 14.3  Raw Data File Month3.dat (first five records)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IA10200</td>
<td>SYD</td>
<td>HKG</td>
<td>01MAR2000</td>
</tr>
<tr>
<td>IA10201</td>
<td>SYD</td>
<td>HKG</td>
<td>01MAR2000</td>
</tr>
<tr>
<td>IA10300</td>
<td>SYD</td>
<td>CBR</td>
<td>01MAR2000</td>
</tr>
<tr>
<td>IA10301</td>
<td>SYD</td>
<td>CBR</td>
<td>01MAR2000</td>
</tr>
<tr>
<td>IA10302</td>
<td>SYD</td>
<td>CBR</td>
<td>01MAR2000</td>
</tr>
</tbody>
</table>

The SAS log indicates that the raw data files referenced by Qtr1 are sequentially read into the SAS data set Work.FirstQtr.
Table 14.4  SAS Log

9  filename qtr1 ('c:\sasuser\month1.dat' 'c:\sasuser\month2.dat'
10     'c:\sasuser\month3.dat');

11  data work.firstqtr;
12    infile qtr1;
13    input Flight $ Origin $ Dest $
14      Date : date9. RevCargo : comma15.2;
15    run;

NOTE: The infile QTR1 is:
   File Name=c:\sasuser\month1.dat,
   File List=('c:\sasuser\month1.dat' 'c:\sasuser\month2.dat'
   'c:\sasuser\month3.dat'),
   RECFM=V,LRECL=256

NOTE: The infile QTR1 is:
   File Name=c:\sasuser\month2.dat,
   File List=('c:\sasuser\month1.dat' 'c:\sasuser\month2.dat'
   'c:\sasuser\month3.dat'),
   RECFM=V,LRECL=256

NOTE: The infile QTR1 is:
   File Name=c:\sasuser\month3.dat,
   File List=('c:\sasuser\month1.dat' 'c:\sasuser\month2.dat'
   'c:\sasuser\month3.dat'),
   RECFM=V,LRECL=256

NOTE: 50 records were read from the infile QTR1.
The minimum record length was 33.
The maximum record length was 37.

NOTE: 50 records were read from the infile QTR1.
The minimum record length was 33.
The maximum record length was 37.

NOTE: 50 records were read from the infile QTR1.
The minimum record length was 33.
The maximum record length was 37.

NOTE: 50 records were read from the infile QTR1.
The minimum record length was 33.
The maximum record length was 37.

NOTE: The data set WORK.FIRSTQTR has 150 observations
and 5 variables.

NOTE: DATA statement used (Total process time):
real time          4.02 seconds
                  cpu time         0.93 seconds

The following PROC PRINT output shows a portion of the observations in the
Work.FirstQtr data set.
Using an INFILE Statement

You can make the process of concatenating raw data files more flexible by using an INFILE statement with the FILEVAR= option. The FILEVAR= option enables you to dynamically change the currently opened input file to a new input file.

General form, INFILE statement with the FILEVAR= option:

```
INFILE file-specification FILEVAR= variable;
```

where

- **FILEVAR= variable**
  - names a variable whose change in value causes the INFILE statement to close the current input file and open a new input file.

- **variable**
  - contains a character string that is a physical filename.

When you use an INFILE statement with the FILEVAR= option, the file specification is a placeholder, not an actual filename or a fileref that had been assigned previously to a file. SAS uses this placeholder for reporting processing information to the SAS log. The file specification must conform to the same rules as a fileref.

When the INFILE statement executes, it reads from the file that the FILEVAR= variable specifies. Like automatic variables, this variable is not written to the data set.

Example

Suppose you want to create a SAS data set that contains three months of data stored in three raw data files. The three months are the current month and the previous two months.
In the following INFILE statement, temp is an arbitrarily named placeholder, not an actual filename or a fileref that had been assigned to a file previously. The FILEVAR= variable nextfile contains the name of the raw data file to be read, for example, Month9.dat, Month10.dat, or Month11.dat. A RUN statement is not included because the program is not complete.

```
data work.quarter
    infile temp filevar=nextfile;
    input Flight $ Origin $ Dest $ Date : date9. RevCargo : comma15.2;
```

Table 14.5 Raw Data File Month9.dat (first five records)

```
---------10--------20--------30--------40
IA10200 SYD HKG 01SEP2000 $189,441.00
IA10201 SYD HKG 01SEP2000 $175,473.00
IA10300 SYD CBR 01SEP2000 $1,370.00
IA10301 SYD CBR 01SEP2000 $710.00
IA10302 SYD CBR 01SEP2000 $1,210.00
```

Table 14.6 Raw Data File Month10.dat (first five records)

```
---------10--------20--------30--------40
IA10200 SYD HKG 01OCT2000 $182,457.00
IA10201 SYD HKG 01OCT2000 $160,923.00
IA10300 SYD CBR 01OCT2000 $1,030.00
IA10301 SYD CBR 01OCT2000 $870.00
IA10302 SYD CBR 01OCT2000 $770.00
```

Table 14.7 Raw Data File Month11.dat (first five records)

```
---------10--------20--------30--------40
IA10200 SYD HKG 01NOV2000 $176,637.00
IA10201 SYD HKG 01NOV2000 $164,997.00
IA10300 SYD CBR 01OCT2000 $1,230.00
IA10301 SYD CBR 01OCT2000 $1,230.00
IA10302 SYD CBR 01OCT2000 $790.00
```

Note: You can also use multiple INFILE statements or operating system techniques to combine raw data files. However, this chapter discusses only the FILENAME statement and the INFILE statement with the FILEVAR= option.

---

Assigning the Names of the Files to Be Read

The next step is to assign the names of the three files to be read to the variable nextfile:
data work.quarter;
    infile temp filevar=nextfile;
    input Flight $ Origin $ Dest $ Date : date9. RevCargo : comma15.2;

In this case, let's use the raw data files Month9.dat, Month10.dat, and Month11.dat. Notice that the titles of the raw data files are very similar. They each start with "Month" and are followed by numeric characters and the file extension .dat:

Month9.dat
Month10.dat
Month11.dat

You can use an iterative DO loop and the PUT function to automatically change the values assigned to nextfile.

Example

In the following code, the DO statement creates the index variable i and assigns it the values of 9, 10, and 11. The assignment statement then assigns the name of the raw data file to nextfile using the current value of i and the PUT function. Month9.dat, Month10.dat, and Month11.dat are stored in the C:\Sasuser directory in the Windows operating environment. On the right side of the assignment statement, the text string c:sasuser\month is concatenated with the current value of i using the double exclamation point (!!) concatenation operator. c:sasuser\month is then concatenated with the text string .dat.

data work.quarter;
    do i = 9, 10, 11;
        nextfile="c:sasuser\month"
                   !!put(i,2.)!!".dat";
        infile temp filevar=nextfile;
        input Flight $ Origin $ Dest $ Date : date9. RevCargo : comma15.2;
    end;

The following table shows the value of nextfile as the value of i changes.

<table>
<thead>
<tr>
<th>When i=</th>
<th>nextfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>c:sasuser\month9.dat</td>
</tr>
<tr>
<td>10</td>
<td>c:sasuser\month10.dat</td>
</tr>
<tr>
<td>11</td>
<td>c:sasuser\month11.dat</td>
</tr>
</tbody>
</table>

Tip: Depending on the characters available on your keyboard, the symbol you use as the concatenation operator can be a double vertical bar (||), a double broken vertical bar (¦¦), or a double exclamation point (!!).

Using the COMPRESS Function

Note the space between month and 9 in c:sasuser\month 9.dat. You can eliminate the space by using the COMPRESS function.
Combining Data Vertically

Example

In the following code, the COMPRESS function removes blank spaces from the value of `nextfile`:

```sas
data work.quarter;
  do i = 9, 10, 11;
    nextfile="c:\sasuser\month"!!put(i,2.)!!".dat";
    nextfile=compress(nextfile,' ');  
    infile temp filevar=nextfile;
    input Flight $ Origin $ Dest $ Date : date9. RevCargo : comma15.2;
  end;
```

The COMPRESS function can be combined with the assignment statement for greater efficiency:

```sas
data work.quarter;
  do i = 9, 10, 11;
    nextfile="c:\sasuser\month"!!compress(put(i,2.)!!".dat"," ");
    infile temp filevar=nextfile;
    input Flight $ Origin $ Dest $ Date : date9. RevCargo : comma15.2;
  end;
```

With the addition of the COMPRESS function, when the value of `i` equals 9, `nextfile` is assigned the correct value, `c:\sasuser\month9.dat`. 

---

<table>
<thead>
<tr>
<th>When i=</th>
<th>nextfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>c:sasuser:month9.dat</td>
</tr>
<tr>
<td>10</td>
<td>c:sasuser:month10.dat</td>
</tr>
<tr>
<td>11</td>
<td>c:sasuser:month11.dat</td>
</tr>
</tbody>
</table>

General form, COMPRESS function:

```sas
COMPRESS(source, <characters-to-remove>);
```

where

- `source` specifies a source string that contains the characters to remove.
- `characters-to-remove` specifies the character or characters that SAS removes from the source string.

**Note:** If the `characters-to-remove` is omitted, the COMPRESS function removes blank spaces from the source.
Let’s add a few more statements to the program. An OUTPUT statement within the DO loop outputs each record to the SAS data set Work.Quarter and a STOP statement prevents an infinite loop of the DATA step.

```
data work.quarter;
do i = 9, 10, 11;
   nextfile="c:sasuser\month\"
       !compress(put(i,2.)!!".dat"," ' ');
   infile temp filevar=nextfile;
   input Flight $ Origin $ Dest $ Date : date9.
      RevCargo : comma15.2;
   output;
end;
stop;
```

The program is almost complete. However, there are several other statements that need to be added in order to read all of the input data.

### Using the END= Option

When you read past the last record in a raw data file, the DATA step normally stops processing. In this case, you need to read the last record in the first two raw data files. However, you do not want to read past the last record in either of those files because the DATA step will stop processing. You can use the END= option with the INFILE statement to determine when you are reading the last record in the last raw data file.

**General form, INFILE statement with the END= option:**

```
INFILE file-specification END=variable;
```

where

- **variable**
  - names a variable that SAS sets to
    - 0 when the current input data record is not the last record in the input file
    - 1 when the current input record is the last record in the input file.

**Note:** Like automatic variables, the END= variable is not written to the SAS data set.

The END= option enables you to name a variable whose value is controlled by SAS. The value of the variable is 0 when you are not reading the last record in an input file and 1 when you are reading the last record in an input file. You can test the value of the END= variable to determine if the DATA step should continue processing.
Combining Data Vertically

**Example**

The END= variable `lastobs` is created in the INFILE statement. The DO UNTIL statement conditionally executes until the value of `lastobs` equals 1. A RUN statement completes the program.

```sas
data work.quarter;
  do i = 9, 10, 11;
    nextfile="c:sasuser\month"
      !compress(put(i,2.)!".dat",');
    do until (lastobs);
      infile temp filevar=nextfile end=lastobs;
      input Flight $ Origin $ Dest $ Date : date9.
      RevCargo : comma15.2;
      output;
    end;
  end;
stop;
run;
```

PROC PRINT output shows a portion of the observations in the SAS data set `Work.Quarter`. A LABEL statement is used to assign the descriptive label `Month` to the variable `i`. Notice that the variables `nextfile` and `lastobs` are not written to the data set.

```sas
proc print data=work.quarter (firstobs=45 obs=55) label;
  label i='Month';
  format date date9. revcargo dollar11.2;
run;
```

**Using Date Functions**

You can make your program more flexible by eliminating the need to include explicit month numbers in your SAS statements. To create a program that will always read the current month and the previous two months, you can use date functions to obtain the month number of today's date to begin the quarter.

**Example**

In the following program, the MONTH and TODAY functions are used to obtain the value of the variable `monthnum`. The TODAY function returns the current date from the
system clock as a SAS date value. The month number is then extracted from the current date using the MONTH function.

The value of `midmon` is calculated by subtracting 1 from the value of `monthnum`. The value of `lastmon` is then calculated by subtracting 1 from the values of `midmon`. The following table shows the values `monthnum`, `midmon`, and `lastmon` if the current date is October 22, 2002.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>monthnum</td>
<td>10</td>
</tr>
<tr>
<td>midmon</td>
<td>9</td>
</tr>
<tr>
<td>lastmon</td>
<td>8</td>
</tr>
</tbody>
</table>

In the previous example, the DO statement created the index variable `i` and assigned it the values of 9, 10, and 11. Here, the DO statement assigns `i` the values of `monthnum`, `midmon`, and `lastmon`:

```sas
data work.quarter (drop=monthnum midmon lastmon);
   monthnum=month(today());
   midmon=monthnum-1;
   lastmon=midmon-1;
   do i = monthnum, midmon, lastmon;
      nextfile="c:sasuser\month"
         !!compress(put(i,2.)!!".dat",’ ’);
      do until (lastobs);
         infile temp filevar=nextfile end=lastobs;
         input Flight $ Origin $ Dest $ Date : date9.
            RevCargo : comma15.2;
         output;
      end;
   end;
   stop;
run;
```

The following PROC PRINT output shows a portion of the observations in `Work.Quarter`.
Combining Data Vertically

Using the INTNX Function

In the previous example the current month was October. What happens if the current month is January or February?

Suppose the current date is February 16, 2003. Using the following program, the values for `midmon` (January) and `lastmon` (December) would be 1 and 0 respectively. Since there is no “0” month, the program would fail to read the third raw data file.

You can use the INTNX function with the TODAY and MONTH functions to correctly determine the values of `midmon` and `lastmon` for any given date. Remember that the INTNX function increments a date, time, or datetime value by a given interval or intervals, and returns a date, time, or datetime value.
Example

Suppose the current date is January 30, 2003. In the following program `monthnum` is assigned a value of 1 using the `TODAY` and `MONTH` functions. The `INTNX` function is used with the `TODAY` and `MONTH` functions to assign a value of 12 to `midmon` and a value of 11 to `lastmon`.

```sas
data work.quarter (drop=monthnum midmon lastmon);
monthnum=month(today());
midmon=month(intnx('month',today(),-1));
lastmon=month(intnx('month',today(),-2));
do i = monthnum, midmon, lastmon;
   nextfile="c:sasuser\month"!!compress(put(i,2.)!!".dat"," ");
   do until (lastobs);
      infile temp filevar=nextfile end=lastobs;
      input Flight $ Origin $ Dest $ Date : date9.
      RevCargo : comma15.2;
      output;
   end;
end;
stop;
run;
```

The following PROC PRINT output shows a portion of the observations in `Work.Quarter`.

```sas
proc print data=work.quarter (firstobs=45 obs=55) label;
   label i='Month';
   format date date9. revcargo dollar11.2;
run;
```

### Appending SAS Data Sets

Now that you have seen several methods for concatenating raw data files, let's take a look at how you can use the APPEND procedure to concatenate two SAS data sets.
**Example**

The SAS data sets `Work.Cap2001` and `Work.Capacity` both contain the following variables: `Cap1st, CapBusiness, CapEcon, Dest, FlightID, Origin`, and `RouteID`. However, the BASE= data set (`Work.Cap2001`) contains an additional variable, `Date`, that is not included in the DATA= data set (`Work.Capacity`).

When the following program is submitted, the SAS log indicates that the variable `Date` was not found in the DATA= file.

```sas
proc append base=work.cap2001
data=work.capacity;
run;
```

**Table 14.8  SAS Log**

```
WARNING: Variable Date was not found on DATA file.
NOTE: There were 50 observations read from the data set WORK.CAPACITY.
NOTE: 50 observations added.
NOTE: The data set WORK.CAP2001 has 100 observations and 8 variables.
NOTE: PROCEDURE APPEND used (Total process time):
```

PROC PRINT output of the appended version of `Work.Cap2001` shows that missing values have been assigned to `Date` in the observations that were read in from the DATA= data set.
Using the FORCE Option

In the previous example, the DATA= data set (Work.Capacity) contained fewer variables than the BASE= data set (Work.Cap2001). However, you may need to append data sets when the DATA= data set contains more variables than the BASE= data set.

You must use the FORCE option with the APPEND procedure to concatenate data sets when the DATA= data set contains variables that are not in the BASE= data set.

**General form, PROC APPEND with the FORCE option:**

```sql
PROC APPEND BASE=SAS-data-set DATA=SAS-data-set <FORCE>;
```

**CAUTION:**

The FORCE option can cause loss of data due to truncation or dropping of variables.

When you use the FORCE option, the structure of the BASE= data set is used for the appended data set.
Example

Remember that the SAS data sets `Work.Cap2001` and `Work.Capacity` both contain the following variables: `Cap1st`, `CapBusiness`, `CapEcon`, `Dest`, `FlightID`, `Origin`, and `RouteID`. In this case, the `DATA=` data set (`Work.Cap2001`) contains an additional variable, `Date`, that is not included in the `BASE=` data set (`Work.Capacity`).

When the following program is submitted, the SAS log indicates that the data sets were not appended because the variable `Date` was not found in the `BASE=` file.

```sas
proc append base=work.capacity
data=work.cap2001;
run;
```

Table 14.9 SAS Log

| NOTE: Appending WORK.CAP2001 to WORK.CAPACITY. |
| WARNING: Variable Date was not found on BASE file. |
| ERROR: No appending done because of anomalies listed above. |
| Use FORCE option to append these files. |
| NOTE: 0 observations added. |
| NOTE: The data set WORK.CAPACITY has 50 observations and 7 variables. |
| NOTE: Statements not processed because of errors noted above. |
| NOTE: PROCEDURE APPEND used (Total process time): |
| real time 0.02 seconds |
| cpu time 0.03 seconds |
| NOTE: The SAS System stopped processing this step because of errors. |

When the FORCE option is used with PROC APPEND, the SAS log indicates that observations have been read from the `DATA=` data set, but that dropping or truncating will occur.

```sas
proc append base=work.capacity
data=work.cap2001 force;
run;
```

Table 14.10 SAS Log

| NOTE: Appending WORK.CAP2001 to WORK.CAPACITY. |
| WARNING: Variable Date was not found on BASE file. |
| NOTE: FORCE is specified, so dropping/truncating will occur. |
| NOTE: There were 50 observations read from the data set WORK.CAP2001. |
| NOTE: 50 observations added. |
| NOTE: The data set WORK.CAPACITY has 100 observations and 7 variables. |
| NOTE: PROCEDURE APPEND used (Total process time): |
| real time 0.03 seconds |
| cpu time 0.03 seconds |

PROC PRINT output shows that the variable `Date` has been dropped from the appended version of `Work.Capacity`. 
proc print data=work.capacity (firstobs=45 obs=55);
run;

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>RouteID</th>
<th>Origin</th>
<th>Dest1st</th>
<th>CapBusiness</th>
<th>CapEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>W404600</td>
<td>0000046</td>
<td>LHR</td>
<td>DLA</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>46</td>
<td>W404600</td>
<td>0000046</td>
<td>GLA</td>
<td>LHR</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>47</td>
<td>W404700</td>
<td>0000047</td>
<td>LHR</td>
<td>AYA</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>48</td>
<td>W404600</td>
<td>0000048</td>
<td>FRA</td>
<td>LHR</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>49</td>
<td>W404600</td>
<td>0000049</td>
<td>LHR</td>
<td>BRU</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>50</td>
<td>W405000</td>
<td>0000050</td>
<td>BRU</td>
<td>LHR</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>51</td>
<td>W405000</td>
<td>0000001</td>
<td>RTD</td>
<td>LHR</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>52</td>
<td>W405001</td>
<td>0000001</td>
<td>RTD</td>
<td>LHR</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>53</td>
<td>W405001</td>
<td>0000002</td>
<td>LHR</td>
<td>RTD</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>54</td>
<td>W405001</td>
<td>0000003</td>
<td>RTD</td>
<td>FRA</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>55</td>
<td>W405003</td>
<td>0000008</td>
<td>JFK</td>
<td>RTD</td>
<td>16</td>
<td>34</td>
</tr>
</tbody>
</table>

Appending Variables with Different Lengths

If the DATA= data set contains variables that are longer than the variables in the BASE= data set, the FORCE option must be used with PROC APPEND. Using the FORCE option enables you to append the data sets. However, the DATA= variable values will be truncated.

Example

In the SAS data set Work.Acities, the variable City has a length of 22. In the SAS data set Work.WestAust, City has a length of 50. You can use the CONTENTS procedure to view the attributes of the variables in each data set.

proc contents data=work.acities;
run;

proc contents data=work.westaust;
run;

When the following program is submitted, the SAS log indicates that the data sets were not appended due to different lengths for City in the BASE= and DATA= data sets.

proc append base=work.acities
    data=work.westaust;
run;
Table 14.11  SAS Log

NOTE: Appending WORK.WESTAUST to WORK.ACITIES.
WARNING: Variable City has different lengths on BASE and DATA files (BASE 22 DATA 50).
ERROR: No appending done because of anomalies listed above.
Use FORCE option to append these files.
NOTE: 0 observations added.
NOTE: The data set WORK.ACITIES has 50 observations and 4 variables.
NOTE: Statements not processed because of errors noted above.
NOTE: PROCEDURE APPEND used (Total process time):
real time  1.44 seconds
cpu time 0.06 seconds
NOTE: The SAS System stopped processing this step because of errors.

When the FORCE option is used, the SAS log indicates that the data sets are appended, but that dropping or truncating will occur.

```sas
proc append base=work.acities
data=work.airports force;
run;
```

Table 14.12  SAS Log

NOTE: Appending WORK.WESTAUST to WORK.ACITIES.
WARNING: Variable City has different lengths on BASE and DATA files (BASE 22 DATA 50).
NOTE: FORCE is specified, so dropping/truncating will occur.
NOTE: There were 50 observations read from the data set WORK.WESTAUST.
NOTE: 50 observations added.
NOTE: The data set WORK.ACITIES has 100 observations and 4 variables.
NOTE: PROCEDURE APPEND used (Total process time):
real time  1.44 seconds
cpu time 0.06 seconds

PROC CONTENTS output for the appended version of Work.Acities shows that the variable City has retained a length of 22 from the BASE= data set. Also notice that the variable Code has retained the label Start Point from the BASE= data set.

```sas
proc contents data=work.acities;
run;
```

<table>
<thead>
<tr>
<th>Alphabetical List of Variables and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

PROC PRINT output shows that some of the values of City are truncated in the appended version of Work.Acities.
Appending Variables with Different Types

If the DATA= data set contains a variable that does not have the same type as the corresponding variable in the BASE= data set, the FORCE option must be used with PROC APPEND. Using the FORCE option enables you to append the data sets. However, missing values are assigned to the DATA= variable values for the variable whose type did not match.

Example

In the SAS data set Work.Allemps, the variable Phone is a character variable. In the SAS data set Work.Newemps, Phone is a numeric variable. You can use PROC CONTENTS to view the attributes of the variables in each data set.

```
proc contents data=work.allemps;
run;
```

```
proc contents data=work.newemps;
run;
```

When the following program is submitted, the SAS log indicates that there is a type mismatch for the variable Phone and that data sets were not appended.

```
proc append base=work.allemps
   data=work.newemps;
```
run;

Table 14.13 SAS Log

NOTE: Appending WORK.NEWEMPS to WORK.ALLEMPS.
WARNING: Variable Phone not appended because of type mismatch.
ERROR: No appending done because of anomalies listed above.
   Use FORCE option to append these files.
NOTE: 0 observations added.
NOTE: The data set WORK.ALLEMPS has 550 observations and 5 variables.
NOTE: Statements not processed because of errors noted above.
NOTE: PROCEDURE APPEND used (Total process time):
   real time 0.08 seconds
   cpu time 0.01 seconds
NOTE: The SAS System stopped processing this step because of errors.

When the FORCE option is used, the SAS log indicates that the data sets are appended, but that the variable Phone is not appended due to the type mismatch.

proc append base=work.allemps
   data=work.newemps force;
run;

Table 14.14 SAS Log

NOTE: Appending WORK.NEWEMPS to WORK.ALLEMPS.
WARNING: Variable Phone not appended because of type mismatch.
NOTE: FORCE is specified, so dropping/truncating will occur.
NOTE: There were 19 observations read from the data set WORK.NEWEMPS.
NOTE: 19 observations added.
NOTE: The data set WORK.ALLEMPS has 569 observations and 5 variables.
NOTE: PROCEDURE APPEND used (Total process time):
   real time 0.05 seconds
   cpu time 0.02 seconds

PROC CONTENTS output for the appended version of Work.Allemps shows that the variable Phone has retained the type of character from the BASE= data set.

proc contents data=work.allemps;
run;

PROC PRINT output of the appended version of Work.Allemps shows that the the values for Phone are missing in the records that were read in from the DATA= data set.
Additional Features

In addition to the methods for appending raw data files that were discussed earlier in this chapter, you can also append raw data files using a SAS data set or an external file that contains the names of the raw data files to be appended.

Storing Raw Data Filenames in a SAS Data Set

In the following program, five raw data files, `Route1.dat`, `Route2.dat`, `Route3.dat`, `Route4.dat`, and `Route5.dat`, are concatenated to create the SAS data set `Work.NewRoute`. The names of the raw data files are stored in the SAS data set `Sasuser.Rawdata`, which is referenced using a SET statement. The name of the FILEVAR= variable, `readit`, is the name of the variable in `Sasuser.Rawdata` whose value is the name of the file to be read.

SAS Data Set

```
Sasuser.Rawdata
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>readit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>route1.dat</td>
</tr>
<tr>
<td>2</td>
<td>route2.dat</td>
</tr>
<tr>
<td>3</td>
<td>route3.dat</td>
</tr>
<tr>
<td>4</td>
<td>route4.dat</td>
</tr>
<tr>
<td>5</td>
<td>route5.dat</td>
</tr>
</tbody>
</table>

```
data work.newroute;
  set sasuser.rawdata;
  infile in filevar = readit end = lastfile;
  do while(lastfile = 0);
    input @1 RouteID $7. @8 Origin $3. @11 Dest $3.
    @14 Distance 5. @19 Fare1st 4.
    @23 FareBusiness 4. @27 FareEcon 4.
    @31 FareCargo 5.;
    output;
  end;
run;
```

Storing Raw Data Filenames in an External File

In the following program, `Route1.dat`, `Route2.dat`, `Route3.dat`, `Route4.dat`, and `Route5.dat` are also concatenated to create the SAS data set `Work.NewRoute`. In this example, the names of the raw data files are stored in the external file `Rawdatafiles.dat`, which is referenced in the first INFILE statement. The name of the
FILEVAR= variable, `readit`, is the name of the variable read from `Rawdatafiles.dat`. The value of `readit` is the name of the raw data file to be read.

**Table 14.15** Raw Data File Rawdatafiles.dat

<table>
<thead>
<tr>
<th>route1.dat</th>
<th>route2.dat</th>
<th>route3.dat</th>
<th>route4.dat</th>
<th>route5.dat</th>
</tr>
</thead>
</table>

```plaintext
data work.newroute;
  infile 'rawdatafiles.dat';
  input readit $10.;
  infile in filevar=readit end=lastfile;
  do while(lastfile = 0);
    input @1 RouteID $7. @8 Origin $3. @11 Dest $3. @14 Distance 5. @19 Fare1st 4. @23 FareBusiness 4. @27 FareEcon 4. @31 FareCargo 5.;
    output;
  end;
run;
```
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Using a FILENAME Statement

You can use a FILENAME statement to concatenate raw data files by assigning a single fileref to the raw data files that you want to combine. When the fileref is specified in an INFILE statement, each raw data file that has been referenced can be sequentially read into a data set using an INPUT statement.

Using an INFILE Statement

You can make the process of concatenating raw data files more flexible by using an INFILE statement with the FILEVAR= option. The FILEVAR= option enables you to dynamically change the currently opened input file to a new input file. When the INFILE statement executes, it reads from the file that the FILEVAR= variable specifies.

In some cases, you may need to use the COMPRESS function to eliminate spaces in the filenames you generate.

When you read the last record in a raw data file, the DATA step normally stops processing. When you are concatenating raw data files, you do not want to read past the last record until you reach the end of the last input file. You can determine if you are reading the last record in the last raw data file by using the END= option with the INFILE statement. You can then test the value of the END= variable to determine if the DATA step should continue processing.

If you are working with date-related data, you may be able to make your program more flexible by eliminating the need to include explicit month numbers in your SAS statements. To create a program that will always read the current month and the previous two months, you can use the MONTH and TODAY functions to obtain the month number of today’s date to begin the quarter. In some cases, you may need to use the INTNX function with the TODAY and MONTH functions to correctly determine the month numbers.

Appending SAS Data Sets

You can use PROC APPEND to concatenate two SAS data sets. PROC APPEND reads only the data in the DATA= SAS data set, not in the BASE= SAS data set. PROC APPEND concatenates data sets even though there may be variables in the BASE= data set that do not exist in the DATA= data set.

The FORCE option must be used if the DATA= data set contains variables that
- are not in the BASE= data set
- are longer than the variables in the BASE= data set
- do not have the same type as the variables in the BASE= data set.

The FORCE option can cause loss of data due to truncation or dropping of variables. The following table summarizes the consequences of using the FORCE option.
Combining Data Vertically

<table>
<thead>
<tr>
<th>DATA= data set contains variables that...</th>
<th>FORCE required?</th>
<th>Consequences of using the FORCE option</th>
</tr>
</thead>
<tbody>
<tr>
<td>are in the BASE= data set, but the BASE= data set has more variables</td>
<td>no</td>
<td>Missing values are assigned to the extra BASE= data set variables.</td>
</tr>
<tr>
<td>are not in the BASE= data set</td>
<td>yes</td>
<td>Extra DATA= data set variables are dropped.</td>
</tr>
<tr>
<td>are longer than the variables in the BASE= data set</td>
<td>yes</td>
<td>DATA= data set variable values are truncated.</td>
</tr>
<tr>
<td>do not have the same type as the variables in the BASE= data set</td>
<td>yes</td>
<td>Missing values are assigned to the DATA= data set variables with the data type mismatch.</td>
</tr>
</tbody>
</table>

Additional Features

You can also append raw data files using a SAS data set or an external file that contains the names of the raw data files to be appended.

Syntax

Combining Raw Data Files Using a FILENAME Statement

FILENAME fileref ('external-file1' 'external-file2' 'external-filen');
DATA SAS-data-set;
  INFILE file-specification;
  INPUT variable <$> <&|:|> <informat>;
RUN;

Combining Raw Data Files Using an INFILE Statement

DATA SAS-data=set;
  DO index-variable=variable1, variable2, variablen;
    variable = “text-string” !!PUT(index-variable,format)!”text-string”;
    variable = COMPRESS (variable,’ ’);
    DO UNTIL( variable);
      INFILE file-specification FILEVAR=variable END=variable;
      INPUT variable <$> <&|:|> <informat>;
      OUTPUT;
    END;
  END;
  STOP;
RUN;

Combining SAS Data Sets Using PROC APPEND

PROC APPEND BASE=SAS-data-set DATA=SAS-data-set <FORCE>;
RUN;
Sample Programs

Combining Raw Data Files Using a FILENAME Statement

```sas
filename qtr1 ('c:\data\month1.dat' 'c:\data\month2.dat' 'c:\data\month3.dat');
data work.firstqtr;
  infile qtr1;
  input Flight $ Origin $ Dest $ Date : date9. RevCargo : comma15.2;
run;
```

Combining Raw Data Files Using an INFILE Statement

```sas
data quarter (drop=monthnum midmon lastmon);
  monthnum=month(today());
  midmon=month(intnx('month',today(),-1));
  lastmon=month(intnx('month',today(),-2));
  do i = monthnum, midmon, lastmon;
    nextfile="c:sasuser\month" !!compress(put(i,2.)!!".dat"," ");
    do until (lastobs);
      infile temp filevar=nextfile end=lastobs;
      input Flight $ Origin $ Dest $ Date : date9.
      RevCargo : comma15.2;
      output;
    end;
  end;
  stop;
run;
```

Combining SAS Data Sets Using PROC APPEND

```sas
proc append base=work.acities data=work.airports force;
run;
```

Points to Remember

- When you use an INFILE statement with the FILEVAR= option, the file specification is just a placeholder, not an actual filename or a fileref that has been previously assigned to a file.
- Like automatic variables, the FILEVAR= variable and the END= variable are not written to the data set.
- Using the FORCE option with PROC APPEND can cause loss of data due to truncation or dropping of variables.
- When you use the FORCE option, the structure of the BASE= data set is used for the appended data set.
Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Which of the following statements associates the fileref OnSale with the raw data files London.dat, Paris.dat, and Zurich.dat? The files are stored in the C:\Routes\New directory in the Windows operating environment.
   a filename onsale (c:\routes\new\london.dat,
      c:\routes\new\paris.dat,
      c:\routes\new\zurich.dat);
   b filename onsale ‘c:\routes\new\london.dat’
      ‘c:\routes\new\paris.dat’
      ‘c:\routes\new\zurich.dat’;
   c filename onsale (‘c:\routes\new\london.dat’
      ‘c:\routes\new\paris.dat’
      ‘c:\routes\new\zurich.dat’);
   d filename onsale ‘c:\routes\new\london.dat
      c:\routes\new\paris.dat
      c:\routes\new\zurich.dat’;

2 Which of the following statements is true?
   a The FILEVAR= option can be used to dynamically change the currently
      opened input file to a new physical file.
   b The FILEVAR= variable is not written to the data set.
   c The FILEVAR= variable must contain a character string that is a physical
      filename.
   d all of the above

3 Given the following program, which table correctly shows the corresponding values
   of the variables x and readfile?

   data work.revenue;
      do x = 8, 9, 10;
         readfile=compress("c:\data\month"
            !!put(x,2.)!!".dat",’ ‘);
      do until (lastobs);
         infile temp filevar=nextfile
            end=lastobs;
         input Date : date7. Location $
            Sales : dollar10.2;
      output;
      end;
   end;
   stop;
run;

   a

<table>
<thead>
<tr>
<th>When x=</th>
<th>readfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>month8.dat</td>
</tr>
<tr>
<td>9</td>
<td>month9.dat</td>
</tr>
<tr>
<td>10</td>
<td>month10.dat</td>
</tr>
</tbody>
</table>
If the current date is March 30, 2003, which table correctly shows the corresponding values of the variables $y_1, y_2, y_3,$ and $nextfile$?

```
data work.quarter (drop=monthnum midmon lastmon);
  y3=year(today());
  y2=y3-1;
  y1=y3-2;
  do i = y3, y2, y1;
    nextfile="c:\data\Y"!!put(i,4.)!!".dat";
    do until (lastobs);
      infile temp filevar=nextfile
      end=lastobs;
      input Flight $ Origin $ Dest $ Date : date9.;
      output;
    end;
  end;
run;
```

4. If the current date is March 30, 2003, which table correctly shows the corresponding values of the variables $y_1, y_2, y_3,$ and $nextfile$?

<table>
<thead>
<tr>
<th>When $x$</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$nextfile$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>$c:\data\month8.dat$</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>$c:\data\month9.dat$</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>$c:\data\month10.dat$</td>
</tr>
</tbody>
</table>

a

<table>
<thead>
<tr>
<th>When $i$</th>
<th>$nextfile$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1$</td>
<td>$c:\data\Y2001.dat$</td>
</tr>
<tr>
<td>$y_2$</td>
<td>$c:\data\Y2002.dat$</td>
</tr>
<tr>
<td>$y_3$</td>
<td>$c:\data\Y2003.dat$</td>
</tr>
</tbody>
</table>
5 Which of the following statements is false?
   a The END= variable is set to 0 when SAS processes the last data record in the input file.
   b The END= variable is set to 1 when SAS processes the last data record in the input file.
   c The END= variable is not written to the data set.
   d a and c

6 Which program appends Work.London to Work.Flights?

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Destination</td>
<td>char</td>
<td>3</td>
</tr>
<tr>
<td>Departure</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Gate</td>
<td>char</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Destination</td>
<td>char</td>
<td>3</td>
</tr>
<tr>
<td>Departure</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Gate</td>
<td>char</td>
<td>2</td>
</tr>
</tbody>
</table>

   a proc append base=work.london data=work.flights;
   run;
b proc append data=work.london
   base=work.flights;
   run;

c proc append data=work.london work.flights;
   run;

d proc append data=work.flights work.london;
   run;

7 What happens when the following program is submitted?

   proc append base=staff.marketing
       data=staff.sales force;
   run;

Data Set Description for Staff.Marketing

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>char</td>
<td>12</td>
</tr>
<tr>
<td>FirstName</td>
<td>char</td>
<td>10</td>
</tr>
<tr>
<td>EmpID</td>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>Office</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Phone</td>
<td>char</td>
<td>12</td>
</tr>
</tbody>
</table>

Data Set Description for Staff.Sales

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>char</td>
<td>20</td>
</tr>
<tr>
<td>FirstName</td>
<td>char</td>
<td>10</td>
</tr>
<tr>
<td>EmpID</td>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>Office</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Phone</td>
<td>char</td>
<td>12</td>
</tr>
</tbody>
</table>

a The length of **LastName** is converted to 20 in **Staff.Marketing**.
b **LastName** is dropped from **Staff.Marketing**.
c Missing values are assigned to **LastName** observations that are read in from **Staff.Sales**.
d Some of the values of **LastName** may be truncated in the observations that are read in from **Staff.Sales**.

8 Which program appends **Work.April** to **Work.Y2003**?

Data Set Description for Work.Y2003

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>FirstClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>BusinessClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Coach</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

Data Set Description for Work.April

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>FirstClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>BusinessClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Coach</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

a proc append base=work.y2003
       data=work.april;
   run;

b proc append base=work.april
       data=work.y2003 force;
Combining Data Vertically

Quiz

9 What happens when the SAS data set Work.NewHires is appended to the SAS data set Work.Employees using PROC APPEND?

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>EmpID</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Name</td>
<td>char</td>
<td>20</td>
</tr>
<tr>
<td>Room</td>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>Extension</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

Data Set Description for Work.Employees

Data Set Description for Work.NewHires

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>EmpID</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Name</td>
<td>char</td>
<td>20</td>
</tr>
<tr>
<td>Extension</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

a Missing values are assigned to Room for the observations that are read in from Work.NewHires.
b Missing values are assigned to Room for all of the observations in Work.Employees.
c Room is dropped from Work.Employees.
d The values of Name are truncated in the observations that are read in from Work.NewHires.

10 You do not need to use the FORCE option with PROC APPEND when

a the DATA= data set contains variables that are not in the BASE= data set.
b the BASE= data set contains variables that are not in the DATA= data set.
c the variables in the DATA= data set are longer than the corresponding variables in the BASE= data set.
d the variables in the DATA= data set have a different type than the corresponding variables in the BASE= data set.
CHAPTER 15
Combining Data Horizontally

Overview

Introduction 514
Objectives 515
Prerequisites 515
Reviewing Terminology 515

Relationships between Input Data Sources 517

Working with Lookup Values Outside of SAS Data Sets 519

The IF-THEN/ELSE Statement 519
Example: Using the IF-THEN/ELSE Statement to Combine Data 519

SAS Arrays 519
Example: Using the ARRAY Statement to Combine Data 520

User-Defined SAS Formats 520
Example: Using the FORMAT Procedure to Combine Data 521

Combining Data with the DATA Step Match-Merge 521

The DATA Step Match-Merge 521
Working with Multiple Lookup Tables 522
Example 523

Using PROC SQL to Join Data 525

The SQL Procedure 525
Example: Working with Multiple Lookup Tables 525

Comparing DATA Step Match-Merges and PROC SQL Joins 526
Examples 528

DATA Step Match-Merge 531
Execution of a DATA Step Match-Merge 531

PROC SQL Join 532
Execution of a PROC SQL Join 532
Example: Combining Data from a Many-to-Many Match 533
Using Multiple SET Statements 534
Example: Using Multiple SET Statements with a Many-to-Many Match 535

Combining Summary Data and Detail Data 535

The MEANS Procedure 537
Example 537
Example 538
The Sum Statement 539
Example 539

Using an Index to Combine Data 541

The KEY= Option 541
Example 542
Example 543
Example 544
The _IORC_ Variable 545
Overview

Introduction

Combining data horizontally refers to the process of merging or joining multiple data sets into one data set. This process is referred to as a horizontal combination because in the final data set, each observation (or horizontal row) will have variables from more than one input data set.
It is useful to combine data horizontally if you have several data sets that contain different but related information. For example, suppose you have one data set that contains employee data with the variables IDNumber, Name, and Address, and another data set that contains employee data with the variables IDNumber and Salary. You can combine the data from these two input data sets horizontally to create an output data set that contains IDNumber, Name, Address, and Salary.

There are several methods for combining data horizontally. This chapter focuses on several methods of combining data horizontally in the DATA step, and compares a DATA step match-merge with a PROC SQL join. This chapter also covers several techniques for horizontally combining data from an input data set with values that are not stored in a SAS data set.

Objectives

In this chapter, you learn to

- identify factors that affect which technique is most appropriate for combining data horizontally
- use the IF-THEN/ELSE statement, SAS arrays, or user-defined SAS formats to combine data horizontally
- use the DATA step with the MERGE statement to combine data sets that don’t have a common variable
- use the SQL procedure to combine data sets that don’t have a common variable
- identify the differences between the DATA step match-merge and the PROC SQL join
- create an output data set that contains summary statistics from PROC MEANS
- combine summary statistics in a data set with a detail data set
- calculate summary data and combine it with detail data within one DATA step
- use the SET statement with the KEY= option to combine two SAS data sets
- use an index to combine two data sets
- use _IORC_ to determine whether an index search was successful
- use the UPDATE statement to update a master data set with a transactional data set.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 13, “Creating Samples and Indexes,” on page 451.

Reviewing Terminology

Before examining the various techniques for combining data horizontally, let’s review some of the terminology that this chapter uses. The table below lists important terms that you will need to know, along with their definitions.
Reviewing Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>combining data horizontally</td>
<td>A technique in which information is retrieved from an auxiliary source or sources, based on the values of variables in the primary source.</td>
</tr>
<tr>
<td>performing a table lookup</td>
<td>A technique in which information is retrieved from an auxiliary source or sources, based on the values of variables in the primary source.</td>
</tr>
<tr>
<td>base table</td>
<td>The primary source in a horizontal combination. In this chapter, the base table is always a SAS data set.</td>
</tr>
<tr>
<td>lookup table(s)</td>
<td>All input data sources, except the base table, that are used in a horizontal combination.</td>
</tr>
<tr>
<td>lookup values</td>
<td>The data values that are retrieved from the lookup table(s) during a horizontal combination.</td>
</tr>
<tr>
<td>key variable(s)</td>
<td>One or more variables that reside in both the primary file and the lookup file. The values of the key variable(s) are the common elements between the files. Often, key values are unique in the base file but are not necessarily unique in the lookup file(s).</td>
</tr>
<tr>
<td>key value(s)</td>
<td>For each observation, the value(s) for the key variable(s).</td>
</tr>
</tbody>
</table>

Note: The terms combining data horizontally and performing a table lookup are synonymous and are used interchangeably throughout this chapter.

Note: This chapter compares PROC SQL techniques with DATA step techniques. In PROC SQL terms, a SAS data set is usually referred to as a table, a variable is usually referred to as a column, and an observation is usually referred to as a row.

The following figure illustrates a base table and a lookup table that are used in a horizontal combination. The key variable is Num. The key values are listed vertically below Num.
Relationships between Input Data Sources

One important factor to consider when you perform a table lookup is the relationship between the input data sources. In order to combine data horizontally, you must be able to match observations from each input data source. For example, there might be one or more variables that are common to each input data source. The relationship between input data sources describes how the observations in one source relate to the observations in the other source according to these key values.

The following terms describe the possible relationships between base tables and lookup tables:

- one-to-one match
- one-to-many match
- many-to-many match
- nonmatching data.

Let’s look at each of these relationships in more detail.

In a one-to-one match, key values in both the base table and the lookup table are unique. Therefore, for each observation in the base table, no more than one observation in the lookup table has a matching key value.

In a one-to-many match, key values in the base table are unique, but key values in the lookup table are not unique. That is, for each observation in the base table there can be one observation or possibly multiple observations in the lookup table that have a matching key value.

In a many-to-many match, key values are not unique in the base table or in the lookup table. That is, at least one observation in the base table matches multiple observations in the lookup table, and at least one observation in the lookup table matches multiple observations in the base table.
Sometimes you will have a one-to-one, a one-to-many, or a many-to-many match that also includes nonmatching data. That is, there are observations in the base table that do not match any observations in the lookup table, or there are observations in the lookup table that do not have matching observations in the base table. If your base table or lookup table(s) include nonmatching data, you will have one of the following:

- **a dense match**, in which nearly every observation has a matching observation in the corresponding table. In the following figure, the first observation in the base table is unmatched.

\[
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarA} \\
\hline
1 & A1 \\
2 & A2 \\
3 & A3 \\
4 & A4 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarB} \\
\hline
1 & B1 \\
1 & B2 \\
1 & B3 \\
3 & B4 \\
4 & B1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarB} \\
\hline
2 & B1 \\
3 & B2 \\
4 & B3 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarA} \\
\hline
1 & A1 \\
2 & A2 \\
3 & A3 \\
\hline
\end{array}
\]

*Note:* A dense match can also refer to a relationship in which every observation has a matching observation in the corresponding table and there is no nonmatching data.

- **a sparse match**, in which there are more unmatched observations than matched observations in either the base table or the lookup table. In the following figure, the first, third, and fourth observations in the base table, and the second and third observations in the lookup table are unmatched.

\[
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarA} \\
\hline
1 & A1 \\
2 & A2 \\
3 & A3 \\
5 & A5 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarB} \\
\hline
2 & B1 \\
4 & B2 \\
5 & B3 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarB} \\
\hline
2 & B1 \\
3 & B2 \\
4 & B3 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Num} & \text{VarA} \\
\hline
1 & A1 \\
2 & A2 \\
3 & A3 \\
\hline
\end{array}
\]
Working with Lookup Values Outside of SAS Data Sets

Remember that it is not necessary for your lookup table to be a SAS data set. Suppose you want to combine the data from your base table with lookup values that are not stored in a SAS data set. You can use the following techniques to hard-code lookup values into your program:

- the IF-THEN/ELSE statement
- SAS arrays
- user-defined SAS formats.

The IF-THEN/ELSE Statement

You should be familiar with the syntax and use of the IF-THEN/ELSE statement. Overall, this technique is easy to use and easy to understand. Because of its simplicity and because you can use other DATA step syntax with it, the IF-THEN/ELSE statement can be quite versatile as a technique for performing lookup operations. You can use this technique if your lookup values are not stored in a data set, and you can use it to handle any of the possible relationships between your base table and your lookup table if your lookup values are stored in a data set. You can use it to retrieve single or multiple values. For example, you can use DO groups to provide multiple values based on a condition.

Keep in mind that this technique will require maintenance. If you expect your lookup values to change and you have a large number of lookup values, or if you use the lookup values in multiple programs, the resources required for maintaining the IF-THEN/ELSE statements in your programs might make this technique inappropriate. Also, this technique might result in a prohibitively long program or even in a program that will not execute because it times out.

Example: Using the IF-THEN/ELSE Statement to Combine Data

Suppose you have a data set, Mylib.Employees, that contains information about employees. Mylib.Employees contains a variable named IDnum that records each employee’s unique identification number. If you want to combine the data from Mylib.Employees with a list of employees’ birthdates that is not stored in a data set, you can use the IF-THEN/ELSE statement to do so.

```sas
data mylib.employees_new;
    set mylib.employees;
    if IDnum=1001 then Birthdate='01JAN1963'd;
    else if IDnum=1002 then Birthdate='08AUG1946'd;
    else if IDnum=1003 then Birthdate='23MAR1950'd;
    else if IDnum=1004 then Birthdate='17JUN1973'd;
    run;
```

Note: Mylib.Employees is a fictitious data set that is used for this example and for the examples on the next two pages.

SAS Arrays

You should be familiar with the syntax and use of the ARRAY statement. With the ARRAY statement, you can either hard-code your lookup values into the program, or you can read them into the array from a data set. Elements of a SAS array are
referenced positionally. That is, you use a numeric value as a pointer to the array element, so you must be able to identify elements of the array either by position or according to another numeric value. You can use multiple values or numeric mathematical expressions to determine the array element to be returned. Exact matches are not required with this technique.

The memory requirements for loading the entire array can be a drawback to using the ARRAY statement to perform a table lookup. Also, this technique is capable of returning only a single value from the lookup operation. Finally, the dimensions of the array must be supplied at compile time either by hard-coding or through the use of macro variables.

---

**Example: Using the ARRAY Statement to Combine Data**

Let’s take another look at our example of combining the data from `Mylib.Employees` with a list of lookup values. Remember that `Mylib.Employees` contains data about employees, which includes their identification numbers (`IDnum`) but does not include their birthdates. You can use the ARRAY statement to hard-code the birthdates into a temporary array named `birthdates`, and then use the array to combine the birthdates with the data in `Mylib.Employees`.

In the following DATA step, the values that are specified as subscripts for the array correspond to values of the variable `IDnum` in the base table, `Mylib.Employees`. The assignment statement for the new variable `Birthdate` retrieves a value from the `birthdates` array according to the current value of `IDnum`.

```sas
data mylib.employees_new;
  array birthdates{1001:1004} _temporary_ ('01JAN1963'd '08AUG1946'd '23MAR1950'd '17JUN1973'd);
  set mylib.employees;
  Birthdate=birthdates(IDnum);
run;
```

---

**User-Defined SAS Formats**

You should be familiar with the syntax and use of the FORMAT procedure with the VALUE statement. You can associate a format with a variable permanently by using a FORMAT or ATTRIB statement in a DATA step or PROC step that creates a SAS data set. In a DATA step, you can use a PUT statement in an assignment statement in order to use the format only while the PUT function executes. In a DATA step or PROC step, you can use the PUT function in a WHERE statement in order to use the format only during execution of the PUT function.

One advantage of using formats to combine data is that you do not have to create a new SAS data set in order to perform the lookup. Formats can be used to collapse data into categories as well as to expand data, and they can change the appearance of a report without the creation of a new variable. You can create multiple formats and use all of them in the same DATA or PROC step.

The FORMAT procedure uses a binary search (a rapid search technique) through the lookup table. Another benefit of using this technique is that maintenance is centralized; if a lookup value changes, you only have to change it in one place (in the format), and every program that uses the format will use the new value.

On the other hand, the FORMAT procedure requires the entire format to be loaded into memory for the binary search, so this technique might use more memory than others if there are a large number of lookup values.
Example: Using the FORMAT Procedure to Combine Data

Once again, suppose the data set *Mylib.Employees* contains information about employees according to their employee identification numbers (**IDnum**), but does not contain employees’ birthdates. You can use a format to combine employees’ birthdates with the data that is stored in *Mylib.Employees*.

The following PROC FORMAT step uses a VALUE statement to hard-code the lookup values in the BIRTHDATE format. Then the DATA step uses the PUT function to associate the lookup values from the format with the values of **IDnum**, uses the INPUT function to associate the lookup value with the values of **DATE9.** informat, and assigns the formatted values to a new variable named **Birthdate**.

```
proc format;
  value birthdate 1001 = ‘01JAN1963’
     1002 = ‘08AUG1946’
     1003 = ‘23MAR1950’
     1004 = ‘17JUN1973’;
run;

data mylib.employees_new;
  set mylib.employees;
  Birthdate=input(put(IDnum,birthdate.),date9.);
run;
```

Combining Data with the DATA Step Match-Merge

The DATA Step Match-Merge

You should already know how to merge multiple data sets in the DATA step when there is a **BY variable** that is common to each of the input data sets. When you use the MERGE statement to perform a table lookup operation, your lookup values must be stored in one or more SAS data sets. Also, this technique requires that both the base table and the lookup table(s) be either sorted by or indexed on the **BY variable(s)**.

You can specify any number of input data sets in the MERGE statement as long as all input data sets have a common **BY variable**. Also, the MERGE statement can combine data sets of any size. The MERGE statement is capable of returning multiple values, and you can use multiple **BY variables** to perform lookups that are dependent on more than one variable. The MERGE statement returns both matches and non-matches by default, but you can use DATA step syntax to return only exact matches or to include only specific values from the lookup table.

**CAUTION:**

Although you can use the MERGE statement to combine data from sources that have any type of relationship, this technique might not produce the desired results when you are working with a many-to-many match. When the data sets are merged in a DATA step, the observations are matched and combined sequentially. Once an observation is read, it is never re-read. That is, the DATA step MERGE statement does not create a Cartesian product. Therefore, the DATA step MERGE statement is probably not an appropriate technique to use for performing lookup operations when you are working with a many-to-many match. △
Working with Multiple Lookup Tables

Sometimes you might need to combine data from three or more related SAS data sets in order to create one new data set. For example, the three data sets listed below all contain different information that relates to a fictional airline's flights and airports. *Sasuser.Acities* contains data about various airports, *Sasuser.Revenue* contains data about the revenue generated by various flights, and *Sasuser.Expenses* contains data about the expenses incurred by various flights. The variables in each of these data sets are listed here.

<table>
<thead>
<tr>
<th><em>Sasuser.Acities</em> Variables</th>
<th><em>Sasuser.Revenue</em> Variables</th>
<th><em>Sasuser.Expenses</em> Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td>Date</td>
<td>Date</td>
</tr>
<tr>
<td>Code</td>
<td>Dest</td>
<td>FlightID</td>
</tr>
<tr>
<td>Country</td>
<td>FlightID</td>
<td>Origin</td>
</tr>
<tr>
<td>Name</td>
<td>Origin</td>
<td>RevBusiness</td>
</tr>
<tr>
<td></td>
<td>RevEcon</td>
<td>Rev1st</td>
</tr>
</tbody>
</table>

Suppose you want to create a new data set, named *Sasuser.Alldata*, that contains data from each of these three input data sets. As shown below, the *Sasuser.Alldata* data set contains the variable *Profit*, which is calculated from the revenue values that are stored in *Sasuser.Revenue* and the expense values that are stored in *Sasuser.Expenses*.

<table>
<thead>
<tr>
<th><em>Sasuser.Alldata</em> Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Dest</td>
</tr>
<tr>
<td>FlightID</td>
</tr>
<tr>
<td>Origin</td>
</tr>
<tr>
<td>Profit</td>
</tr>
<tr>
<td>DestAirport</td>
</tr>
<tr>
<td>DestCity</td>
</tr>
</tbody>
</table>

You know that you can specify any number of input data sets in the MERGE statement as long as all input data sets have a common BY variable. However, you can see from the data set variable lists above that these three data sets do not have one common variable. Let's look at a method for performing a match-merge on these three data sets.

In order to use the MERGE statement in the DATA step to combine data sets, all input data sets must have a common variable. Although the three data sets *Sasuser.Acities*, *Sasuser.Revenue*, and *Sasuser.Expenses* do not have a common BY variable, there are several variables that are common to two of the three data sets. As shown below, *Date* and *FlightID* are both common to *Revenue* and *Expenses*. The variable *Code* in the *Acities* data set and the variable *Dest* in the *Revenue* data set are also common variables.
Combining Data Horizontally

Notice that **Code** in `Acities` and **Dest** in `Revenue` are listed as corresponding to one another even though they have different names. When you are looking for common variables between data sets, the variable names are not important since they can be changed with the RENAME= option in the MERGE statement. Instead, you should look for variables that record the same information and that have the same type in each input data set. Common variables do not need to have the same length, although you should remember that the length of the variable in the first-listed data set will determine the length of the variable in the output data set.

**Note:** Any variables that have the same name in multiple data sets in the MERGE statement must also have the same type. If any variables in different input data sets have identical names but do not have identical types, ERROR and WARNING messages are written to the SAS log, and the match-merge fails.

In this case, both **Code** in `Acities` and **Dest** in `Revenue` record the three-letter abbreviation of an airport.

**Tip:** You can use PROC CONTENTS to view information about variables such as type, length, and description.

Since there are variables that are common to two different pairs of the three data sets shown above, you can combine these data sets into one data set by using the MERGE statement in two subsequent DATA steps. That is, you can perform one match-merge on two of the data sets to create one new data set that combines information from the two. Then you can perform another match-merge on the new data set and the remaining original data set. Let’s take a closer look.

### Example

In the following program, both `Sasuser.Expenses` and `Sasuser.Revenue` are sorted by **FlightID** and **Date** and are placed into temporary data sets in order to prepare them for being merged. Then these two sorted data sets are merged in a DATA step that creates a temporary output data set named `Revexpns`. In order to reduce the total number of variables in the output data set, a new variable named **Profit** is created, and the variables that are used to create **Profit** are dropped from `Revexpns`.

```sas
proc sort data=sasuser.expenses out=expenses;
    by flightid date;
run;

proc sort data=sasuser.revenue out=revenue;
    by flightid date;
run;

data revexpns (drop=rev1st revbusiness revecon expenses);
    merge expenses(in=e) revenue(in=r);
```
by flightid date;
if e and r;
   Profit=sum(rev1st, revbusiness, revecon, -expenses);
run;

Note: The use of the temporary IN= variables E and R in the IF statement above ensures that only observations that contain data from each of the two input data sets are included in the output data set.

In the following program, the output data set named Revexpns is sorted by Dest. Sasuser.Actities is sorted by Code and is placed in a temporary data set. Remember that Dest and Code are corresponding variables even though they have different names.

The sorted data sets are then merged in a DATA step. Since two data sets must have at least one variable that matches exactly in order to be merged, the RENAME= option renames Code to Dest in the output data set. The DATA step merges Revexpns and Actities into a new output data set named Alldata.

```
proc sort data=revexpns;
   by dest;
run;

proc sort data=sasuser.actities out=acities;
   by code;
run;

data sasuser.alldata;
   merge revexpns(in=r) acities
     (in=a rename=(code=dest) keep=city name code);
   by dest;
   if r and a;
run;

proc print data=sasuser.alldata(obs=5) noobs;
   title 'Result of Merging Three Data Sets';
   format Date date9.;
run;
```

The PROC PRINT step prints the first five observations in the Sasuser.Alldata data set that is created in this example, as shown here.

### Result of Merging Three Data Sets

<table>
<thead>
<tr>
<th>FlightID</th>
<th>Date</th>
<th>Origin</th>
<th>Dest</th>
<th>Profit</th>
<th>City</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA03300</td>
<td>06DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>34010</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
</tr>
<tr>
<td>IA03300</td>
<td>18DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>73471</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
</tr>
<tr>
<td>IA03300</td>
<td>30DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>77755</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
</tr>
<tr>
<td>IA03301</td>
<td>13DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>110402</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
</tr>
<tr>
<td>IA03301</td>
<td>25DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>111151</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
</tr>
</tbody>
</table>
Using PROC SQL to Join Data

The SQL Procedure

Another method that you can use to join data sets that do not have a common variable is the SQL procedure. You should already be familiar with using PROC SQL to create a table from the results of an inner join.

In a PROC SQL step, you can choose from each input data set only the specific variables that you want to include in the new data set, and you can return multiple values. The input data sets do not need to contain a common BY variable, nor do they need to be sorted or indexed. However, if the lookup table does have an index, the SQL procedure can take advantage of the index to provide faster retrieval of lookup values.

You can join up to 32 tables with the SQL procedure, and you can use this technique to combine data horizontally from sources that have any type of relationship (one-to-one, one-to-many, many-to-many, or non-matching). Exact matches are returned by default from an inner join.

Note: Although numerous kinds of joins are possible with PROC SQL, only inner joins are discussed in this chapter. Therefore, in the remainder of this chapter, “a PROC SQL join” refers to an inner join on multiple tables, with the results stored in a new table. You can learn more about PROC SQL joins in Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79.

One drawback to using the SQL procedure to perform table lookups is that you cannot use DATA step syntax with PROC SQL, so complex business logic is difficult to incorporate into the join. However, by using PROC SQL you can often do in one step what it takes multiple PROC SORT and DATA steps to accomplish.

Example: Working with Multiple Lookup Tables

The following example joins Sasuser.Revenue, Sasuser.Expenses, and Sasuser.Acities into a new data set named Work.Sqljoin:

```sql
proc sql;
    create table sqljoin as
    select revenue.flightid, revenue.date format=date9.,
        revenue.origin, revenue.dest,
        sum(revenue.rev1st,
            revenue.revbusiness,
            revenue.revecon) -expenses.expenses as Profit,
        acities.city,
        acities.name
    from sasuser.expenses, sasuser.revenue,
        sasuser.acities
    where expenses.flightid=revenue.flightid
        and expenses.date=revenue.date
        and acities.code=revenue.dest
    order by revenue.dest,
        revenue.flightid,
        revenue.date;
quit;
```
proc print data=work.sqljoin(obs=5);
   title 'Result of Joining Three Data Sets';
run;

The PROC PRINT step produces the first five observations of the Work.Sqljoin data set that is created in the PROC SQL step above, as shown here:

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>Date</th>
<th>Origin</th>
<th>Dest</th>
<th>Profit</th>
<th>City</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA03300</td>
<td>06DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>34010</td>
<td>Anchorage, AK Anchorage International Airport</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IA03300</td>
<td>18DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>73471</td>
<td>Anchorage, AK Anchorage International Airport</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IA03300</td>
<td>30DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>77756</td>
<td>Anchorage, AK Anchorage International Airport</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IA03301</td>
<td>13DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>110402</td>
<td>Anchorage, AK Anchorage International Airport</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>IA03301</td>
<td>25DEC1999</td>
<td>RDU</td>
<td>ANC</td>
<td>111151</td>
<td>Anchorage, AK Anchorage International Airport</td>
<td></td>
</tr>
</tbody>
</table>

Note: Notice that the Work.Sqljoin data set is identical to the Sasuser.Alldata data set that was previously created in the DATA step merge.

Comparing DATA Step Match-Merges and PROC SQL Joins

You have seen that it is possible to create identical results with a DATA step match-merge and a PROC SQL inner join. Although the results might be identical, these two processes are very different, and trade-offs are associated with choosing one method over the other. The following tables summarize some of the advantages and disadvantages of each of these two methods.
Combining Data Horizontally △ Comparing DATA Step Match-Merges and PROC SQL Joins 527

<table>
<thead>
<tr>
<th>Table 15.1</th>
<th>DATA Step Match-Merge</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
<td>Disadvantages</td>
<td></td>
</tr>
<tr>
<td>□ There is no limit to the number of input data sets, other than memory.</td>
<td>□ Data sets must be sorted by or indexed on the BY variable(s) prior to merging.</td>
<td></td>
</tr>
<tr>
<td>□ Allows for complex business logic to be incorporated into the new data set by using DATA step processing, such as arrays and DO loops, in addition to MERGE features.</td>
<td>□ The BY variable(s) must be present in all data sets, and the names of the key variable(s) must match exactly.</td>
<td></td>
</tr>
<tr>
<td>□ Multiple BY variables enable lookups that depend on more than one variable.</td>
<td>□ An exact match on the key value(s) must be found.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 15.2</th>
<th>PROC SQL Join</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
<td>Disadvantages</td>
<td></td>
</tr>
<tr>
<td>□ Data sets do not have to be sorted or indexed, but an index can be used to improve performance.</td>
<td>□ The maximum number of tables that can be joined at one time is 32.</td>
<td></td>
</tr>
<tr>
<td>□ Multiple data sets can be joined in one step without having common variables in all data sets.</td>
<td>□ Complex business logic is difficult to incorporate into the join.</td>
<td></td>
</tr>
<tr>
<td>□ You can create data sets (tables), views, or query reports with the combined data.</td>
<td>□ PROC SQL might require more resources than the DATA step with the MERGE statement for simple joins.</td>
<td></td>
</tr>
</tbody>
</table>

Although it is possible to produce identical results with a DATA step match-merge and a PROC SQL join, these two processes will not always produce results that are identical by default. These two techniques work differently. In order to decide which technique you should use in a particular situation, you should carefully consider both the data that you want to combine and the results that you want to produce.

Let’s take a look at some simplified examples to see how each method works in various circumstances.

The following two steps show two different ways to produce the same combination of two data sets, Data1 and Data2, that have a common variable, X. If Data1 contains two variables, X and Y, and Data2 contains two variables, X and Z, then both of the following steps produce an output data set named Data3 that contains three variables, X, Y, and Z.

**Note:** The code shown in the following two steps illustrates a simple comparison of a DATA step match-merge and a PROC SQL join. This comparison will be explored in the next several sections. △

data data3;
   merge data1 data2;
   by x;
run;

proc sql;
   create table data3 as
      select data1.x, data1.y, data2.z
      from data1, data2
where data1.x=data2.x;
quit;

The contents of Data3 will vary depending on the values that are in each input data set and on the method used for merging. Let's take a closer look at some examples.

**Examples**

*One-to-one matches* produce identical results whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose that Data1 and Data2 contain the same number of observations. Also, suppose that in each data set, the values of x are unique, and that each value appears in both data sets.

When these data sets are either merged in a DATA step or joined in a PROC SQL step, Data3 will contain one observation for each unique value of x, and it will have the same number of observations as Data1 and Data2.

**One-to-many matches** produce identical results whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose that Data1 contains unique values for x, but that Data2 does not contain unique values for x. That is, Data2 contains multiple observations that have the same value for x and therefore contains more observations than Data1.

When these two data sets are either merged in a DATA step or joined in a PROC SQL step, Data3 will contain the same number of observations as Data2. In Data3, one observation from Data1 that has a particular value for x might be matched with multiple observations from Data2 that have the same value for x.
Many-to-many matches produce different results depending on whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose the values of \( x \) are not unique in either Data1 or in Data2.

When the data sets are merged in a DATA step, the observations are matched and combined sequentially. That is, an observation from the first input data set will be combined with the first observation from the second input data set that has a matching value for the BY variable. Although there might be additional observations in the second input data set that match the observation from the first input data set, these will not be included in the output data set.

In the example below, Data3 will contain the same number of observations as the larger of the two input sets. In cases where there is a many-to-many match on the values of the BY variable, a DATA step match-merge probably does not produce the desired output because the output data set will not contain all of the possible combinations of matching observations.

When the data sets are joined in a PROC SQL step, each match appears as a separate observation in the output data set. In the example below, the first observation that has a value of 1 for \( x \) in Data1 is matched and combined with each observation from Data2 that has a value of 1 for \( x \). Then, the second observation that has a value of 1 for \( x \) in Data1 is matched and combined with each observation from Data2 that has a value of 1 for \( x \), and so on.
Nonmatching data between the data sets produces different results depending on whether the data sets are merged in a DATA step or combined by using a PROC SQL inner join.

When data sets that contain nonmatching values for the BY variable are merged in a DATA step, the observations in each are processed sequentially. Data3 will contain one observation for each unique value of X that appears in either Data1 or Data2. For those values that have a nonmatching value for X, the observation in Data3 will have a missing value for the variable that is taken from the other input data set.

When data sets that contain nonmatching values for the BY variable are joined in a PROC SQL step, the output data set will contain only those observations that have matching values for the BY variable. In the example below, Data3 does not have any observations with missing values, because any observation from Data1 or from Data2 that contains a nonmatching value for X is not included in Data3.
You have seen the results of DATA step match-merges and PROC SQL joins in several simple scenarios. To help you understand the differences more fully, let’s take a closer look at how the DATA step processes a match-merge and at how PROC SQL processes a join.

**DATA Step Match-Merge**

When you merge data sets in a DATA step, the observations in each input data set are read sequentially and are matched and combined in the output data set. The example below depicts a DATA step match-merge of two simple input data sets.

**Execution of a DATA Step Match-Merge**

1. This example shows the execution of the DATA step below. This DATA step creates a new data set by performing a basic match-merge on two input data sets.

   ```sas
   data work.data3;
   merge data1 data2;
   by x;
   run;
   ```

2. During the compilation phase, SAS reads the descriptor portions of the input data sets and creates the PDV. Also, SAS determines the BY groups in each input data set for the variables listed in the BY statement.

3. Execution begins. SAS looks at the first BY group in each input data set to determine if the BY values match. If so, SAS reads the first observation of that BY group from the first input data set and records the values in the PDV.

4. Since the BY values match, SAS reads the first observation of the same BY group from the second input data set and records the remaining values in the PDV.

5. SAS writes the merged data to the output data set.

6. SAS continues to merge observations in the same manner until it has written all observations from the first BY group to the new data set. In this example, there are two observations in the new data set that result from the first BY group (x = 11).
7 If the BY values do not match, SAS reads the input data set with the lowest BY value. The PDV and the output data set will contain missing values for variables that are unique to the other data set.

8 If an input data set does not contain any observations in a particular BY group, the PDV and the output data set will contain missing values for the variables that are unique to that input data set.

9 SAS continues to match-merge observations until all observations from both input data sets have been read and written to the new data set. In this example, `Work.Data3` contains three variables and four observations.

---

**PROC SQL Join**

A PROC SQL join uses a different process than a DATA step merge to combine data sets.

Conceptually, PROC SQL first creates a Cartesian product of all input sets. That is, PROC SQL first matches each observation with every other observation in the other input data sets. Then, PROC SQL eliminates any observations from the result set that do not satisfy the WHERE clause of the join statement. The PROC SQL query optimizer contains methods to minimize the Cartesian product that must be built.

**Execution of a PROC SQL Join**

1 This example shows the execution of the PROC SQL step below. This PROC SQL step creates a new data set to hold the results of an inner join on two input data sets. This animation provides a conceptual view of how PROC SQL works rather than a literal depiction of the join process. In reality, SAS employs optimization routines that make the process more efficient.

```sql
proc sql;
    create table work.data4 as
    select *
    from data1, data2
    where data1.x=data2.x;
quit;
```

2 Conceptually, PROC SQL first creates a Cartesian product of the two input data sets, where each observation from the first data set is combined with each observation from the second data set. PROC SQL starts by taking the first observation from `Work.Data1` and combining it with the first observation of `Work.Data2`.

3 Next, PROC SQL takes the first observation from `Work.Data1` and combines it with the second observation from `Work.Data2`.

4 PROC SQL continues in this manner until it has combined each observation from `Work.Data1` with every observation from `Work.Data2`. This is the Cartesian product of the two input data sets.

5 Finally, PROC SQL eliminates from the output data set those observations that do not satisfy the condition in the WHERE clause of the program. In this example, observations that do not have matching values for `x` are eliminated so that the two columns for `x` have identical values for each observation.

6 The results are written to the output data set. Only one of the `x` columns is included in the output data set; because they have identical values for each observation, it does not matter which `x` column is kept and which is discarded. In
this example, the output data set `Work.Data4` contains three variables and four observations. None of the observations in `Work.Data4` contains any missing values.

Earlier in this chapter, you learned that a DATA step match-merge will probably not produce the desired results when the data sources that you want to combine have a many-to-many match. You also learned that PROC SQL and the DATA step match-merge do not, by default, produce the same results when you are combining data sources that contain nonmatching data. Now that you have seen how DATA step match-merges and PROC SQL joins work, let’s take a look at an example of using each of these techniques to combine data from a many-to-many match that also contains nonmatching data.

**Example: Combining Data from a Many-to-Many Match**

Suppose you want to combine the data from `Sasuser.Flightschedule` and `Sasuser.Flightattendants`. The `Sasuser.Flightschedule` data set contains information about flights that have been scheduled for a fictional airline. The data set `Sasuser.Flightattendants` contains information about the flight attendants of a fictional airline. A partial listing of each of these data sets is shown below.

**Table 15.3  SASuser.Flightschedule (Partial Listing)**

<table>
<thead>
<tr>
<th>Date</th>
<th>Destination</th>
<th>FlightNumber</th>
<th>EmpID</th>
</tr>
</thead>
<tbody>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>132</td>
<td>1739</td>
</tr>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>132</td>
<td>1478</td>
</tr>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>132</td>
<td>1130</td>
</tr>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>132</td>
<td>1983</td>
</tr>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>132</td>
<td>1111</td>
</tr>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>182</td>
<td>1076</td>
</tr>
<tr>
<td>01MAR2000</td>
<td>YYZ</td>
<td>182</td>
<td>1118</td>
</tr>
</tbody>
</table>

**Table 15.4  Sasuser.Flightattendants (Partial Listing)**

<table>
<thead>
<tr>
<th>EmpID</th>
<th>JobCode</th>
<th>LastName</th>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1350</td>
<td>FA3</td>
<td>Arthur</td>
<td>Barbara</td>
</tr>
<tr>
<td>1574</td>
<td>FA2</td>
<td>Cahill</td>
<td>Marshall</td>
</tr>
<tr>
<td>1437</td>
<td>FA3</td>
<td>Carter</td>
<td>Dorothy</td>
</tr>
<tr>
<td>1988</td>
<td>FA3</td>
<td>Dean</td>
<td>Sharon</td>
</tr>
<tr>
<td>1983</td>
<td>FA2</td>
<td>Dunlap</td>
<td>Donna</td>
</tr>
<tr>
<td>1125</td>
<td>FA2</td>
<td>Eaton</td>
<td>Alicia</td>
</tr>
<tr>
<td>1475</td>
<td>FA1</td>
<td>Fields</td>
<td>Diana</td>
</tr>
<tr>
<td>1422</td>
<td>FA1</td>
<td>Fletcher</td>
<td>Marie</td>
</tr>
</tbody>
</table>

Suppose you want to combine all variables from the `Sasuser.Flightschedule` data set with the first and last names of each flight attendant who is scheduled to work on each flight. `Sasuser.Flightschedule` contains data for 45 flights, and three flight attendants
are scheduled to be on each flight. Therefore, your output data set should contain 135 observations (three for each flight).

You could use the following PROC SQL step to combine 
_Sasuser.Flightschedule_ with 
_Sasuser.Flightattendants_.

```sas
proc sql;
  create table flightemps as
    select flightschedule.*, firstname, lastname
    from sasuser.flightschedule, sasuser.flightattendants
    where flightschedule.empid=flightattendants.empid;
quit;
```

The resulting _Flightemps_ data set contains 135 observations.

Now, suppose you use the following DATA step match-merge to combine these two data sets.

```sas
proc sort data=sasuser.flightattendants out=fa;
  by empid;
run;

proc sort data=sasuser.flightschedule out=fs;
  by empid;
run;

data flightemps2;
  merge fa fs;
  by empid;
run;
```

The resulting _Flightemps2_ data set contains 272 observations. The DATA step match-merge does not produce the correct results because it combines the data sequentially. In the correct results, there are three observations for each unique flight from _Sasuser.Flightschedule_, and there are no missing values in any of the observations. By contrast, the results from the DATA step match-merge contain six observations for each unique flight and many observations that have missing values.

In the last example, data from two data sets that have a many-to-many match was combined. The PROC SQL join produced the correct results, but the DATA step match-merge did not. However, you can produce the correct results in a DATA step.

First, let's look at using multiple SET statements to combine data.

---

**Using Multiple SET Statements**

You can use multiple SET statements to combine observations from several SAS data sets.

For example, the following DATA step creates a new data set named _Combine_. Each observation in _Combine_ contains data from one observation in _Dataset1_ and data from one observation in _Dataset2_.

```sas
data combine;
  set dataset1;
  set dataset2;
run;
```

When you use multiple SET statements

- processing stops when SAS encounters the end-of-file (EOF) marker on either data set (even if there is more data in the other data set). Therefore, the output data set contains the same number of observations as the smallest input data set.
the variables in the program data vector (PDV) are not reinitialized when a second
SET statement is executed.

- for any variables that are common to both input data sets, the value or values
  from the data set in the second SET statement will overwrite the value or values
  from the data set in the first SET statement in the PDV.

Keep in mind that using multiple SET statements to combine data from multiple
input sources that do not have a one-to-one match can be complicated. By default, the
first observation from each data set is combined, the second observation from each data
set is combined, and so on until the first EOF marker is reached in one of the data sets.
Therefore, if you are working with data sources that do not have a one-to-one match, or
that contain nonmatching data, you will need to add additional DATA step syntax in
order to produce the results that you want.

Example: Using Multiple SET Statements with a Many-to-Many Match

Remember that in the previous example you wanted to combine
Sasuser.Flightschedule with Sasuser.Flightattendants. Your resulting data set should
contain all variables from the Sasuser.Flightschedule data set with the first and last
names of each flight attendant who is scheduled to work on each flight.
Sasuser.Flightschedule contains data for 45 flights, and three flight attendants are
scheduled to be on each flight. Therefore, your output data set should contain 135
observations (three for each flight).

You can use the following DATA step to perform this table lookup operation. In this
program, the first SET statement reads an observation from the Sasuser.Flightschedule
data set. Then the DO loop executes, and the second SET statement reads each
observation in Sasuser.Flightattendants. The EmpID variable in
Sasuser.Flightattendants is renamed so that it does not overwrite the value for EmpID
that has been read from Sasuser.Flightschedule. Instead, these two values are used for
comparison to control which observations from Sasuser.Flightattendants should be
included in the output data set for each observation from Sasuser.Flightschedule.

```
data flightemps3(drop=empnum jobcode);
  set sasuser.flightschedule;
  do i=1 to num;
    set sasuser.flightattendants
      (rename=(empid=empnum))
    nobs=num point=i;
    if empid=empnum then output;
  end;
run;
```

The resulting Flightemps3 data set contains 135 observations and no missing values.
Keep in mind that although it is possible to use a DATA step to produce the same
results that a PROC SQL join creates by default, the PROC SQL step might be much
more efficient.

Combining Summary Data and Detail Data

You’ve seen how to combine data from multiple data sets. Suppose you want to
calculate percentages based on individual values from a data set as compared to a
summary statistic of the data. You need to

- create a summary statistic
combine the summary information with the detail rows of the original data set
calculate the percentages.

For example, the data set `Sasuser.Monthsum` has one row for every value of `SaleMon` (month and year) from 1997 to 1999. Each row contains information about the revenue generated by an airline.

*Note:* Note that the `SaleMon` variable has a label of `Sales Month` in the `Sasuser.Monthsum` data set.

**Table 15.5** SAS Data Set `Sasuser.Monthsum` (Partial Listing)

<table>
<thead>
<tr>
<th>Sales Month</th>
<th>RevCargo</th>
<th>MonthNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>1</td>
</tr>
<tr>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>1</td>
</tr>
<tr>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>1</td>
</tr>
<tr>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>2</td>
</tr>
<tr>
<td>FEB1998</td>
<td>$215,959,695.50</td>
<td>2</td>
</tr>
<tr>
<td>FEB1999</td>
<td>$253,999,924.00</td>
<td>2</td>
</tr>
</tbody>
</table>

Suppose you want to produce a report that shows what percentage of the total cargo revenue for the three-year period was generated in each month. You could summarize the data to get the total revenue for cargo for the three-year period and assign that value to a new variable called `Cargosum` in a summary data set.

**Table 15.6** Summary Data Set

<table>
<thead>
<tr>
<th>Cargosum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8,593,432,002.35</td>
</tr>
</tbody>
</table>

Then you would need to combine the summary data (Cargosum) with the detail data in `Sasuser.Monthsum` so that you could calculate percentages of the total cargo revenue for each month.

**Table 15.7** Partial Listing of the Combined Data Set

<table>
<thead>
<tr>
<th>Sales Month</th>
<th>RevCargo</th>
<th>MonthNo</th>
<th>Cargosum</th>
<th>PetRev</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>1</td>
<td>$8,593,432,002.35</td>
<td>&lt;RevCargo/Cargosum&gt;</td>
</tr>
<tr>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>1</td>
<td>$8,593,432,002.35</td>
<td>&lt;RevCargo/Cargosum&gt;</td>
</tr>
<tr>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>1</td>
<td>$8,593,432,002.35</td>
<td>&lt;RevCargo/Cargosum&gt;</td>
</tr>
<tr>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>2</td>
<td>$8,593,432,002.35</td>
<td>&lt;RevCargo/Cargosum&gt;</td>
</tr>
<tr>
<td>FEB1998</td>
<td>$215,959,695.50</td>
<td>2</td>
<td>$8,593,432,002.35</td>
<td>&lt;RevCargo/Cargosum&gt;</td>
</tr>
<tr>
<td>FEB1999</td>
<td>$253,999,924.00</td>
<td>2</td>
<td>$8,593,432,002.35</td>
<td>&lt;RevCargo/Cargosum&gt;</td>
</tr>
</tbody>
</table>

Let’s examine this task more closely.
The MEANS Procedure

You should already know how to use the MEANS procedure for producing summary statistics. By default, PROC MEANS generates a report that contains descriptive statistics. The descriptive statistics can be routed to a SAS data set by using an OUTPUT statement, and the default report can be suppressed by using the NOPRINT option.

General form, PROC MEANS with OUTPUT statement:

```
PROC MEANS DATA=original-SAS-data-set NOPRINT;
     <VAR variable(s)>;
     OUTPUT OUT=output-SAS-data-set
             statistic=output-variable(s);
RUN;
```

where

- `original-SAS-data-set` identifies the data set on which the summary statistic is generated.
- `variable(s)` is the name(s) of the variable(s) that is being analyzed.
- `output-SAS-data-set` names the data set where the descriptive statistics will be stored.
- `statistic` is one of the summary statistics generated.
- `output-variable(s)` names the variable(s) in which to store the value(s) of `statistic` in the output data set.

The output data set that a PROC MEANS step creates contains the requested statistics as values for `output-variable(s)`, as well as two additional variables that are automatically included, as follows:

- `_TYPE_` contains information about the class variables
- `_FREQ_` contains the number of observations that an output level represents.

Example

The following program creates a summary data set named `Sasuser.Summary`. `Summary` contains the sum of the values of `Revcargo` from `Sasuser.Monthsum`, stored in the variable `Cargosum`.

```
proc means data=sasuser.monthsum noprint;
   var revcargo;
   output out=sasuser.summary sum=Cargosum;
run;

proc print data=sasuser.summary;
run;
```

Because of the NOPRINT option, the PROC MEANS step does not produce any output. Printing the `Sasuser.Summary` data set produces the following output.
Once you have created the summary statistic, you need to combine this summary information with the detail rows of the data set so that you can calculate the percentages. Remember that you can use multiple SET statements to combine data horizontally. Let's take a closer look at how this works by using multiple set statements to combine the detail rows of `Sasuser.Monthsum` with the detail data that we created in `Sasuser.Summary`.

### Example

This example creates a new data set named `Percent1` that combines

- summary data (total revenue for cargo from the three-year period) from `Sasuser.Summary`
- detail data (month and total cargo for the month) from `Sasuser.Monthsum`.

`Percent1` also contains a new variable named `PctRev` that records the calculated percentage of the total revenue that each observation represents.

Remember, the automatic variable `_N_` keeps track of how many times the DATA step has begun to execute. The following DATA step uses `_N_` to keep SAS from reaching the EOF marker for `Sasuser.Summary` after the first iteration of the step. Since the variables in the PDV will not be reinitialized on each iteration, the first value of `Summary.Cargosum` will be retained in the PDV for each observation that is read from `Sasuser.Monthsum`.

1. This example shows the compilation and execution of the DATA step below. This DATA step creates a new data set that combines summary data from one input data set (`Sasuser.Summary`) and detail data from a second input data set (`Sasuser.Monthsum`).

   ```
   data sasuser.percent1(drop=cargosum);
   if _N_=1 then set sasuser.summary(keep=cargosum);
   set sasuser.monthsum(keep=salemon revcargo);
   PctRev=revcargo/cargosum;
   run;
   ```

2. During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. `_N_` is a temporary variable that is included in the PDV although it will not be included in the output data set.

3. Execution begins. On the first iteration of the DATA step, `_N_` has a value of 1. The IF statement evaluates as true, so the first SET statement executes and SAS reads the value of `Cargosum` from `Sasuser.Summary` and records it in the PDV.

4. Next, the second SET statement executes. SAS reads the first observation in `Sasuser.Monthsum` and records the values in the PDV.

5. SAS calculates the value of `PctRev` and records it in the PDV.

6. At the end of the DATA step, SAS write the values in the PDV to the output data set. `_N_` is not included in the output data set since it is a temporary variable. `CargoSum` is dropped from the output data set as well.

7. On the second iteration of the DATA step, the value of `_N_` is 2, so the IF statement evaluates to false and the first SET statement does not execute. However, the value of `CargoSum` is retained in the PDV.
8 The second SET statement executes. SAS reads the second observation from Sasuser.Monthsum and records the values in the PDV.

9 The value for PctRev is calculated and recorded in the PDV. SAS will write the values in the PDV to the output data set (except for _N_ and CargoSum).

10 The DATA step will continue to execute until all observations have been read from Sasuser.Monthsum.

Another method of combining summary data and detail data is to create the summary statistic in a DATA step and combine it with the detail data in the same step. To do this you must

- read the data once and calculate the summary statistic
- re-read the data to combine the summary statistic with the detail data and calculate the percentages.

### The Sum Statement

You can use the sum statement to obtain a summary statistic within a DATA step. The sum statement adds the result of an expression to an accumulator variable.

**General form, sum statement:**

```sas
variable+expression;
```

**where**

- `variable` specifies the name of the accumulator variable. This variable must be numeric. The variable is automatically set to 0 before the first observation is read. The variable's value is retained from one DATA step execution to the next.

- `expression` is any valid SAS expression.

#### CAUTION:

If the expression produces a missing value, the sum statement ignores it.

(Remember, however, that assignment statements assign a missing value if the expression produces a missing value.)

#### Note:
The sum statement is one of the few SAS statements that doesn’t begin with a keyword.

The sum statement adds the result of the expression that is on the right side of the plus sign (+) to the numeric variable that is on the left side of the plus sign. At the top of the DATA step, the value of the numeric variable is not set to missing as it usually is when reading raw data. Instead, the variable retains the new value in the program data vector for use in processing the next observation.

### Example

The following example uses a sum statement to generate the summary statistic in a DO UNTIL loop. On the first execution of the DATA step, the DO UNTIL loop reads each observation of Sasuser.Monthsum and keeps a running tally of the total value of RevCargo from each observation. On each subsequent execution of the DATA step, this
tally (stored in the variable `TotalRev`) is divided into `RevCargo` in order to calculate the new variable `PctRev`.

*Note:* Remember that the `END=` data set option creates a temporary variable that contains an end-of-file indicator.

1. This example shows the execution of the DATA step below. This DATA step reads the same data set, `Sasuser.Monthsum`, twice: first, to create a summary statistic; second, to merge the summary statistic back into the detail data to create a new data set, `Sasuser.Percent2`.

```sas
data sasuser.percent2(crop=totalrev);
  if _N_=1 then do until (LastObs);
    set sasuser.monthsum(keep=revcargo) end=lastobs;
    TotalRev+revcargo;
  end;
  set sasuser.monthsum(keep=salemon revcargo);
  PctRev=revcargo/totalrev;
run;
```

2. During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. `_N_`, `LastObs`, and `TotalRev` are temporary variables that are included in the PDV although they will not be included in the output data set.

3. Execution begins. The temporary variables are initialized with values. The IF statement resolves to true on the first iteration of the DATA step, so the DO UNTIL loop begins to execute. Remember, in a DO UNTIL loop, the condition is evaluated at the bottom of the loop.

4. SAS reads the first observation from `Sasuser.Monthsum` and writes the value for `RevCargo` to the PDV.

5. The value of `TotalRev` is increased by the value of `RevCargo` and recorded in the PDV.

6. At the bottom of the DO loop, SAS evaluates the UNTIL expression. It resolves to false since the value of `LastObs` is 0, so the loop continues to execute.

7. SAS reads the second observation from `Sasuser.Monthsum`, overwriting the value for `RevCargo` in the PDV and adding this value to the accumulator variable `TotalRev`.

8. The DO UNTIL loop continues to execute until SAS reads the last observation from `Sasuser.Monthsum` and the value of `LastObs` is set to 1. At this point, the value for `TotalRev` in the PDV is the sum of all values for `RevCargo` in `Sasuser.Monthsum`. The loop is satisfied.

9. The second SET statement reads data from the same data set as the first SET statement did. However, this time values for both `SaleMon` and `RevCargo` are recorded in the PDV. There is already a value for `TotalRev` in the PDV.

10. A value for `PctRev` is calculated for the observation and recorded in the PDV. Then, SAS writes the values in the PDV to the output data set `Sasuser.Percent2`, except for the temporary variables and the variable `TotalRev`.

11. On the second iteration of the DATA step, the value of `_N_` increases to 2, so the IF expression is false. The second SET statement executes and values from the second observation of `Sasuser.Monthsum` are read into the PDV.

12. The value for the accumulator variable `TotalRev` is retained from the last iteration and is used to calculate a new value for `PctRev`, which is recorded in the PDV. SAS writes the values in the PDV to the output data set and the DATA step iterates.

13. The DATA step iterates until SAS has read the last observation from `Sasuser.Monthsum` and has written the new observation to the output data set `Sasuser.Percent2`. 
Using an Index to Combine Data

Suppose you want to combine data from two data sets, and one of the data sets is much larger than the other. Also, suppose you want to use only those observations from the larger data set that match an observation from the smaller data set according to the value of one or more common variables.

You should already know how to create an index on a SAS data set. You have learned that PROC SQL can take advantage of an index to improve performance on a join. You can also take advantage of an index in a DATA step to combine data from matching observations in multiple data sets if the index is built on variables that are common to all input data sets.

For example, suppose you want to combine data from the matching observations in Sasuser.Dnunder and Sasuser.Sale2000. Only a portion of the flights that are in Sasuser.Sale2000 (which has 156 observations) are also in Sasuser.Dnunder (which has only 57 observations). Suppose you want to use only data from Sasuser.Sale2000 about flights that are in both data sets.

Assume that Sasuser.Sale2000 has a composite index named Flightdate associated with it. The values for Flightdate are unique and are based on the values of the variables FlightID and Date. You can use Flightdate to combine data from only the observations in both data sets that have matching values for FlightID and Date.

The next few sections show how to use the Flightdate index to combine matching observations from the Sasuser.Sale2000 data set and the Sasuser.Dnunder data set.

### The KEY= Option

You have seen how to use multiple SET statements in a DATA step in order to combine summary data and detail data in a new data set. You can also use multiple
SET statements to combine data from multiple data sets if you want to combine only data from observations that have matching values for particular variables.

You specify the KEY= option in the SET statement to use an index to retrieve observations from the input data set that have key values equal to the key variable value that is currently in the program data vector (PDV).

General form, SET statement with KEY= option:

```
SET SAS-data-set-name KEY= index-name;
```

where

- `index-name` is the name of an index that is associated with the `SAS-data-set-name` data set.

To use the SET statement with the KEY= option to perform a lookup operation, your lookup values must be stored in a SAS data set that has an index. This technique is appropriate only when you are working with one-to-one matches, and you can use it with a lookup table of any size. It is possible to return multiple values with this technique, and you can use other DATA step syntax with it as well.

When SAS encounters the SET statement that includes the KEY= option, there must already be a value in the PDV for the value or values of the key variable(s) on which the KEY= index is built. SAS can then use the index to retrieve an observation that has a value for the key variable that matches the key value from the PDV.

For example, if the `Sasuser.Sale2000` data set has an index named `Flightdate` associated with it, the following SET statement uses the `Flightdate` index to locate observations in `Sale2000` that have specific values for `FlightID` and `Date`:

```
set sasuser.sale2000 key=flightdate;
```

When the SET statement in the example above begins to execute, there must already be a value for `FlightID` and a value for `Date` in the PDV. SAS then uses the `Flightdate` index to retrieve an observation from `Sasuser.Sale2000` that has values for `FlightID` and `Date` that match the values of `FlightID` and `Date` that are already in the PDV.

In order to assign a key value in the PDV before the SET statement with the KEY= option executes, you precede that SET statement with another SET statement in the DATA step. Let’s look more closely at this example in context.

---

**Example**

Remember that you want to combine `Sasuser.Sale2000` and `Sasuser.Dnunder`, and that `Sasuser.Sale2000` has an index named `Flightdate` that is based on the values of the `FlightID` and the `Date` variables. You can use two SET statements to combine these two data sets, and you can use the KEY= option on the second SET statement to take advantage of the index.

In the following example,

- the first SET statement reads an observation sequentially from the `Sasuser.Dnunder` data set. SAS writes the values from this observation to the PDV, and then moves to the second SET statement.
- SAS will use the `Flightdate` index on `Sasuser.Sale2000` to find an observation in `Sasuser.Sale2000` that has values for `FlightID` and `Date` that match the values of `FlightID` and `Date` that are currently in the PDV.
- `Work.Profit` is the output data set.
CAUTION:
If you use the KEY= option to read a SAS data set, you cannot use WHERE processing on that data set in the same DATA step.

This example shows the execution of a DATA step that uses two SET statements to combine data from two input data sets (Sasuser.Sale2000 and Sasuser.Dnunder) into one output data set (Work.Profit). The DATA step also uses an index on the larger of the two input data sets, which is Sasuser.Sale2000, to find matching observations.

data work.profit;
set sasuser.dnunder;
set sasuser.sale2000(keep=routeid flightid date rev1st
  revbusiness revecon revcargo)
  key=flightdate;
Profit=sum(rev1st, revbusiness, revecon, revcargo,
  -expenses);
run;

SAS sets up the new data set by reading the descriptor portions of the input data sets and creating the PDV.

The first SET statement executes. SAS reads the first observation in Sasuser.Dnunder and records the values in the PDV.

When the second SET statement executes, the KEY= option causes SAS to use the Flightdate index to directly access the observation in Sasuser.Sale2000 that has values for FlightID and Date that match the values already in the PDV. SAS reads the observation and records the values to the PDV.

SAS calculates the value for Profit and records it in the PDV. Then, SAS writes the values from the PDV to the output data

The DATA step continues to iterate. Only the variable Profit is reinitialized to missing. SAS reads the second observation in Sasuser.Dnunder and records the values in the PDV, overwriting the values that have been retained.

SAS uses the Flightdate index to find a matching observation in Sasuser.Sale2000. Then, SAS records the values from that observation in the PDV, overwriting the values that have been retained. A new value for Profit is calculated and recorded, and the values are written to the output data set.

The DATA step continues to iterate until all observations have been read from Sasuser.Dnunder.

Remember that when SAS encounters the SET statement that includes the KEY= option, there must already be a value in the PDV for the value or values of the key variable(s) on which the KEY= index is built. Otherwise, the step will generate errors in the output data set.

Example

The following step is identical to the last example except that the order of the SET statements has been reversed:

data work.profit2;
set sasuser.sale2000(keep=routeid flightid date
  rev1st revbusiness revecon revcargo)
  key=flightdate;
set sasuser.dnunder;
Profit=sum(rev1st, revbusiness, revecon, revcargo,
On the first iteration of this DATA step, there are no values for the key values in the PDV when SAS encounters the SET statement with the KEY= option. Therefore, SAS does not know what to look up in the index, and no observation is read from the Sasuser.Sale2000 data set. SAS proceeds to the second SET statement, reads an observation from the Sasuser.Dnunder data set, and writes the values to the PDV and to the Work.Profit2 data set.

Since no data was read from the Sasuser.Sale2000 data set, there are missing values in the first observation of the output data set. Also, if you examine the values for Revlst, RevBusiness, RevEcon and RevCargo in Work.Profit2 and compare them with the values for these variables in Work.Profit, you will notice that there are differences between these two data sets.

Remember that the values in the PDV are not reinitialized after each iteration of the DATA step. On the second iteration of the DATA step, SAS uses the values from the first observation of Sasuser.Dnunder to match an observation from Sasuser.Sale2000. But before these values are written to the Work.Profit2 data set, a new observation is read from Sasuser.Dnunder and written to the PDV. Therefore, none of the observations in Work.Profit2 actually contains correctly matched data from the two input data sets.

You have seen how to use a SET statement with the KEY= option in conjunction with a second SET statement to create a data set that combines data from matching observations of two input data sets. Remember that when you use multiple SET statements, the variables in the PDV are not reinitialized when the second SET statement is executed. This can lead to problems in the output data set.

Suppose SAS reads an observation from the first input data set on the second iteration of the DATA step (that is, when _N_ =2) and does not find a matching observation in the second input data set. Because the DATA step has already iterated once, and the values in the PDV have not been reinitialized, there are already values in the PDV for all variables. Therefore, the resulting observation in the output data set will contain values from the second observation of the first input data set, combined with values from the first observation of the second input data set.

---

### Example

If you examine the Work.Profit output data set closely, you will notice that the final observation in the output data set contains values for several variables that are identical to values in the previous observation. This duplication of data is incorrect, although the error might not be obvious.

The error in the output data set is caused by a data error in one of the input data sets. If you examine the Sasuser.Dnunder data set closely, you will find that all of the values for FlightID begin with the characters IA10 except the value in the last observation of the data set. Instead, the value for FlightID in the last observation begins with the characters IA11. This is a data error. Because of the data error, when the DATA step executes SAS will not be able to find a matching observation in Sasuser.Sale2000 for the last observation in Sasuser.Dnunder, and will write an observation to the output data set that contains data from the last observation in Sasuser.Dnunder and data from the previous DATA step iteration for Sasuser.Sale2000.

The SAS log provides an additional indication that the final observation in the output data set contains nonmatching data. The observation that contains unmatched data is printed to the log. As you can see in the log sample below, the unmatched observation includes an _Error_ variable whose value is 1, which indicates that there is an error. The _N_ variable indicates the iteration of the DATA step in which the error occurred.
You could check the SAS log for observations that contain errors in order to ensure that your output data set does not contain bad data, but there is a better way. Notice that the observation that is printed in the SAS log above also contains a variable named `_IORC_`. You can use the value of the `_IORC_` variable to prevent the observations that contain errors from being written to your output data set.

### The `_IORC_` Variable

When you use the KEY= option, SAS creates an automatic variable named `_IORC_`, which stands for INPUT/OUTPUT Return Code. You can use `_IORC_` to determine whether the index search was successful. If the value of `_IORC_` is zero, SAS found a matching observation. If the value of `_IORC_` is not zero, SAS did not find a matching observation.

*Note:* The `_IORC_` variable is also automatically created when you use the MODIFY statement with the KEY= option in the DATA step.

In the previous section, you saw an example in which a data error was included in the output data set and was written to the SAS log. To prevent writing the data error to the log (and to your output data set)

- check the value of `_IORC_` to determine whether a match has been found
- set `ERROR=` to 0 if there is no match
- delete the nonmatching data or write the nonmatching data to an errors data set.

### Example

The following example uses the Flightdate index to combine data from Sasuser.Sale2000 with data from Sasuser.Dnunder, and writes the combined data to a new data set named Work.Profit3. If any unmatched observations are read from Sasuser.Dnunder, the resulting combined observation will be written to Work.Errors. No observations should be written to the SAS log.

```sas
data work.profit3 work.errors;
  set sasuser.dnunder;
  set sasuser.sale2000(keep=routeid flightid date revlst
    revbusiness revecon revcargo)key=flightdate;
  if _iorc_=0 then do;
    Profit=sum(revlst, revbusiness, revecon, revcargo,
      -expenses);
    output work.profit3;
  end;
  else do;
```

---

**Table 15.8** SAS Log

| FlightID=IA11802 RouteID=0000108 Date=30DEC2000 Expenses=3720  |
|-------------------------|-------------------------|
| Revlst=1270 RevBusiness=. RevEcon=5292 RevCargo=1940 Profit=4782 |

_`ERROR_=1 _IORC_=1230015 _N_=57_

NOTE: There were 57 observations read from the data set SASUSER.DNUNDER.
NOTE: The data set WORK.PROFIT has 57 observations and 9 variables.
NOTE: DATA statement used (Total process time):
  real time 0.38 seconds
  cpu time 0.04 seconds
Using a Transactional Data Set

Sometimes, rather than just combining data from two data sets, you might want to update the data in one data set with data that is stored in another data set. That is, you might want to update a master data set by overwriting certain values in it with values that are stored in a transactional data set.

For example, suppose the data set `Mylib.Empmaster` contains some information that is outdated. You have the current data stored in another data set named `Mylib.Empchanges`. `Mylib.Empmaster` contains 148 observations, and `Mylib.Empchanges` contains 6 observations. The variable `EmpID` contains unique values in both data sets.

A partial listing of `Mylib.Empmaster` and the full listing of `Mylib.Empchanges` is shown below. Notice that there is one observation in each data set with a value of 1065 for `EmpID`, and that the values of `JobCode` and `Salary` are different in for this observation.

**Table 15.9  Mylib.Empmaster (Partial Listing)**

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>05MAR1957</td>
<td>30MAR1990</td>
<td>1009</td>
<td>M</td>
<td>TA1</td>
<td>$40,432</td>
</tr>
<tr>
<td>01JAN1956</td>
<td>20OCT1979</td>
<td>1017</td>
<td>M</td>
<td>TA3</td>
<td>$57,201</td>
</tr>
<tr>
<td>23MAY1963</td>
<td>27OCT1982</td>
<td>1036</td>
<td>F</td>
<td>TA3</td>
<td>$55,149</td>
</tr>
<tr>
<td>14APR1962</td>
<td>17SEP1990</td>
<td>1037</td>
<td>F</td>
<td>TA1</td>
<td>$39,98</td>
</tr>
<tr>
<td>13NOV1967</td>
<td>26NOV1989</td>
<td>1038</td>
<td>F</td>
<td>TA1</td>
<td>$37,146</td>
</tr>
<tr>
<td>17JUL1961</td>
<td>27AUG1984</td>
<td>1050</td>
<td>M</td>
<td>ME2</td>
<td>$49,234</td>
</tr>
<tr>
<td>29JAN1942</td>
<td>10JAN1985</td>
<td>1065</td>
<td>M</td>
<td>ME2</td>
<td>$49,126</td>
</tr>
<tr>
<td>18OCT1970</td>
<td>06OCT1989</td>
<td>1076</td>
<td>M</td>
<td>PT1</td>
<td>$93,181</td>
</tr>
</tbody>
</table>

**Table 15.10  Mylib.Empchanges**

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>30JUN1955</td>
<td>31JAN1982</td>
<td>1639</td>
<td>F</td>
<td>TA3</td>
<td>$59,164</td>
</tr>
<tr>
<td>29JAN1942</td>
<td>10JAN1985</td>
<td>1065</td>
<td>M</td>
<td>ME3</td>
<td>$53,326</td>
</tr>
<tr>
<td>03DEC1961</td>
<td>10Oct1985</td>
<td>1561</td>
<td>M</td>
<td>TA3</td>
<td>$51,120</td>
</tr>
<tr>
<td>25SEP1965</td>
<td>07OCT1989</td>
<td>1221</td>
<td>F</td>
<td>FA3</td>
<td>$41,854</td>
</tr>
</tbody>
</table>
Combining Data Horizontally

Using the UPDATE Statement

You use the UPDATE statement to update a master data set with a transactional data set. The UPDATE statement can perform the following tasks:

- change the values of variables in the master data set
- add observations to the master data set
- add variables to the master data set.

General form, UPDATE statement:

```sas
DATA master-data-set;
UPDATE master-data-set transaction-data-set;
BY by-variable(s);
RUN;
```

where

- `master-data-set` names the SAS data set used as the master file.
- `transaction-data-set` names the SAS data set that contains the changes to be applied to the master data set.
- `by-variable(s)` names a variable that appears in both `master-data-set` and in `transaction-data-set`. Each observation in `master-data-set` must have a unique value for `by-variable`, but `transaction-data-set` can contain more than one observation with the same `by-variable` value.

The UPDATE statement replaces values in the master data set with values from the transactional data set for each observation with a matching value of the BY variable. Any observations in either the master data set or the transactional data set that have non-matching values for the BY variable are included in the output data set. Also, by default, SAS does not replace existing values in the master data set with missing values if those values are coded as periods (for numeric variables) or blanks (for character variables) in the transactional data set.

When you use the UPDATE statement, keep in mind the following restrictions.

- Only two data set names can appear in the UPDATE statement.
- The master data set must be listed first.
- A BY statement that gives the matching variable must be used.

<table>
<thead>
<tr>
<th>DateOfBirth</th>
<th>DateOfHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>11AUG1970</td>
<td>01NOV2000</td>
<td>1447</td>
<td>F</td>
<td>FA1</td>
<td>$30,340</td>
</tr>
<tr>
<td>13SEP1968</td>
<td>05NOV2000</td>
<td>1998</td>
<td>M</td>
<td>SCP</td>
<td>$32,240</td>
</tr>
</tbody>
</table>
Both data sets must be sorted by or have indexes based on the BY variable.
In the master data set, each observation must have a unique value for the BY variable.

**Example**

Remember that you want to update the master data set *Mylib.Empmaster* with the transactional data set *Mylib.Empchanges*. You can use the UPDATE statement to accomplish this task, as shown the program below. Remember, both data sets must be sorted by or indexed on the BY variable.

```plaintext
proc sort data=mylib.empmaster;
   by empid;
run;

proc sort data=mylib.empchanges;
   by empid;
run;

data mylib.empmaster;
   update mylib.empmaster mylib.empchanges;
      by empid;
run;
```

The first 8 observations of the updated *Mylib.Empmaster* data set are shown below. Notice that the observation that has a value of 1065 for EmpID now contains the updated values for *JobCode* and *Salary*.

<table>
<thead>
<tr>
<th>DateofBirth</th>
<th>DateofHire</th>
<th>EmpID</th>
<th>Gender</th>
<th>JobCode</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>05MAR1957</td>
<td>30MAR1990</td>
<td>1009</td>
<td>M</td>
<td>TA1</td>
<td>$40,432</td>
</tr>
<tr>
<td>01JAN1956</td>
<td>20OCT1979</td>
<td>1017</td>
<td>M</td>
<td>TA3</td>
<td>$57,201</td>
</tr>
<tr>
<td>23MAY1963</td>
<td>27OCT1982</td>
<td>1036</td>
<td>F</td>
<td>TA3</td>
<td>$55,149</td>
</tr>
<tr>
<td>14APR1962</td>
<td>17SEP1990</td>
<td>1037</td>
<td>F</td>
<td>TA1</td>
<td>$39,981</td>
</tr>
<tr>
<td>13NOV1967</td>
<td>25NOV1989</td>
<td>1038</td>
<td>F</td>
<td>TA1</td>
<td>$37,146</td>
</tr>
<tr>
<td>17JUL1961</td>
<td>27AUG1984</td>
<td>1050</td>
<td>M</td>
<td>ME2</td>
<td>$49,234</td>
</tr>
<tr>
<td>29JAN1942</td>
<td>10JAN1985</td>
<td>1065</td>
<td>M</td>
<td>ME3</td>
<td>$53,326</td>
</tr>
<tr>
<td>18OCT1970</td>
<td>06OCT1989</td>
<td>1076</td>
<td>M</td>
<td>PT1</td>
<td>$63,181</td>
</tr>
</tbody>
</table>
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Reviewing Terminology

You can review definitions of terms that are important in this chapter. You can also review diagrams and descriptions of the various relationships between input sources for a table lookup operation.

Working with Lookup Values Outside of SAS Data Sets

You can use the IF-THEN/ELSE statement in the DATA step to combine data from a base table with lookup values that are not stored in a SAS data set. You can also use the FORMAT procedure or the ARRAY statement to combine data from a base table with lookup values that are not stored in a SAS data set.

Combining Data with the DATA Step Match-Merge

You can use the MERGE statement in the DATA step to combine data from multiple data sets as long as the input data sets have a common variable. You can merge more than two data sets that lack a common variable in multiple DATA steps if each input data set contains at least one variable in it that is also in at least one other input data set.

Using PROC SQL to Join Data

You can also use PROC SQL to join data from multiple data sets if there is no single variable that is common to all input data sets. In a PROC SQL step, you can choose only the specific variables from each input data set that you want to include in the new data set. If you create a new table with the results of an inner join in a PROC SQL step, the results can be very similar to the results of a DATA step match-merge.

Comparing DATA Step Match-Merges and PROC SQL Joins

It is possible to create identical results with a basic DATA step match-merge and a PROC SQL join. However, there are significant differences between these two methods, as well as advantages and disadvantages to each. In some cases, such as when there is a one-to-one or a one-to-many match on values of the BY variables in the input data sets, these two methods produce identical results. In other cases, such as when there is a many-to-many match on values of the BY variables, or if there are nonmatching values of the BY variables, these two methods will produce different results. These differences reflect the fact that the processing is different for a DATA step match-merge and a PROC SQL join. Even if you are working with many-to-many matches or nonmatching data, it is possible to use other DATA step techniques such as multiple SET statements to create results that are identical to the results that a PROC SQL step creates.
Combining Summary Data and Detail Data

In order to perform tasks such as calculating percentages based on individual values from a data set as compared to a summary statistic of the data, you need to combine summary data and detail data. One way to create a summary data set is to use PROC MEANS. Once you have a summary data set, you can use multiple SET statements to combine the summary data with the detail data in the original data set. It is also possible to create summary data with a sum statement and to combine it with detail data in one DATA step.

Using an Index to Combine Data

You can use an index to combine data from matching observations in multiple data sets if the index is built on variables that are common to all input data sets. Especially if one of the input data sets is very large, an index can improve the efficiency of the merge. You use the KEY= option in a SET statement in conjunction with another SET statement to use an index to combine data. However, this method might result in data errors in the output data set. You can use the _IORC_ variable to prevent unmatched data from being included in the output data set.

Using a Transactional Data Set

Sometimes, you might want to update the data in one data set with data that is stored in another data set. You use the UPDATE statement to update a master data set with a transactional data set. The UPDATE statement replaces values in the master data set with values from the transactional data set for each observations with a matching value of the BY variable.

Syntax

```
PROC MEANS DATA=original-SAS-data-set NOPRINT;
  <VAR variable(s)>;
  OUTPUT OUT=output-SAS-data-set statistic=output-variable(s);
RUN;

DATA libref.data-set-name;
  SET SAS-data-set-name;
  SET SAS-data-set-name KEY= index-name;
  variable+expression;
RUN;

DATA master-data-set;
  UPDATE master-data-set transaction-data-set;
  BY by-variables;
RUN;
```

Sample Programs

Combining Data with the IF-THEN/ELSE Statement

```
data mylib.employees_new;
  set mylib.employees;
```
Combining Data Horizontally

```sas
if IDnum=1001 then Birthdate='01JAN1963'd;
else if IDnum=1002 then Birthdate='08AUG1946'd;
else if IDnum=1003 then Birthdate='23MAR1950'd;
else if IDnum=1004 then Birthdate='17JUN1973'd;
run;
```

Combining Data with the ARRAY Statement

```sas
data mylib.employees_new;
  array birthdates{1001:1004} _temporary_ ('01JAN1963'd
    '08AUG1946'd '23MAR1950'd '17JUN1973'd);
  set mylib.employees;
  Birthdate=birthdates(IDnum);
run;
```

Combining Data with the FORMAT Procedure

```sas
proc format;
  value $birthdate '1001' = '01JAN1963'
    '1002' = '08AUG1946'
    '1003' = '23MAR1950'
    '1004' = '17JUN1973';
run;

data mylib.employees_new;
  set mylib.employees;
  Birthdate=input(put(IDnum,$birthdate.),date9.);
run;
```

Performing a DATA Step Match-Merge

```sas
proc sort data=sasuser.expenses out=expenses;
  by flightid date;
run;

proc sort data=sasuser.revenue out=revenue;
  by flightid date;
run;

data revexpns (drop=rev1st revbusiness revecon expenses);
  merge expenses(in=e) revenue(in=r);
  by flightid date;
  if e and r;
  Profit=sum(rev1st, revbusiness, revecon,
            -expenses);
run;

proc sort data=revexpns;
  by dest;
run;

proc sort data=sasuser.acities out=acities;
  by code;
run;
```
data sasuser.alldata;
merge revexpns(in=r) acities
  (in=a rename=(code=dest))
  keep=city name code);
by dest;
if r and a;
run;

**Performing a PROC SQL Join**

proc sql;
create table sqljoin as
select revenue.flightid,
    revenue.date format=date9.,
    revenue.origin, revenue.dest,
    sum(revenue.rev1st,
        revenue.revbusiness,
        revenue.reveon)
    -expenses.expenses as Profit,
    acities.city, acities.name
from sasuser.expenses, sasuser.revenue,
    sasuser.acities
where expenses.flightid=revenue.flightid
    and expenses.date=revenue.date
    and acities.code=revenue.dest
order by revenue.dest, revenue.flightid,
    revenue.date;
quit;

**Working with a Many-to-Many Match**

proc sql;
create table flightemp as
select flightschedule.*, firstname, lastname
from sasuser.flightschedule, sasuser.flightattendants
where flightschedule.empid=flightattendants.empid;
quit;
data flightemps3(drop=empnum jobcode)
set sasuser.flightschedule;
do i=1 to num;
    set sasuser.flightattendants
        (rename=(empid=empnum))
        nob=num point=1;
    if empid=empnum then output;
end;
run;

**Combining Summary Data and Detail Data**

proc means data=sasuser.monthsum noprint;
var revcargo;
output out=sasuser.summary sum=Cargosum;
run;
data sasuser.percent1;
  if _n_=1 then set sasuser.summary
       (keep=cargosum);
  set sasuser.monthsum
       (keep=salemon revcargo);
  PctRev=revcargo/cargosum;
run;

data sasuser.percent2(drop=totalrev);
  if _n_=1 then do until(lastobs);
    set sasuser.monthsum(keep=revcargo)
       end=lastobs;
    totalrev+revcargo;
  end;
  set sasuser.monthsum (keep=salemon revcargo);
  PctRev=revcargo/totalrev;
run;

Using an Index to Combine Data

data work.profit work.errors;
  set sasuser.dnunder;
  set sasuser.sale2000(keep=routeid
                   flightid date revlst revbusiness
                   revecon revcargo)key=flightdate;
  if _iorc_=0 then do;
    Profit=sum(revlst, revbusiness, revecon,
               revcargo, -expenses);
    output work.profit;
  end;
  else do;
    _error_=0;
    output work.errors;
  end;
run;

Using a Transactional Data Set

proc sort data=mylib.empmaster;
  by empid;
run;

proc sort data=mylib.empchanges;
  by empid;
run;

data mylib.empmaster;
  update mylib.empmaster mylib.empchanges;
  by empid;
run;
Points to Remember

- In a DATA step match-merge, you can use the RENAME= option to give identical names to variables in input data sets if those variables record the same information in values that have the same type and length.
- You use the OUTPUT statement and the NOPRINT option with the MEANS procedure if you want the results to be routed to an output data set and the default report to be suppressed.
- The automatic variable _N_ keeps track of how many times a DATA step has iterated. The _N_ variable is useful when you are combining data from a summary data set with data from a larger detail data set.
- When you use the UPDATE statement, both data sets must be sorted by or have indexes based on the BY variable.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. According to the data set descriptions below, which of the variables listed qualify as BY variables for a DATA step match-merge?

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>char</td>
<td>5</td>
<td>Department code</td>
</tr>
<tr>
<td>Totemps</td>
<td>num</td>
<td>3</td>
<td>Total number of employees</td>
</tr>
<tr>
<td>Region</td>
<td>char</td>
<td>4</td>
<td>Location of the department</td>
</tr>
<tr>
<td>Manager</td>
<td>num</td>
<td>5</td>
<td>Employee ID number</td>
</tr>
<tr>
<td>IDnum</td>
<td>num</td>
<td>5</td>
<td>Employee ID number</td>
</tr>
<tr>
<td>Name</td>
<td>char</td>
<td>20</td>
<td>Employee name</td>
</tr>
<tr>
<td>Division</td>
<td>char</td>
<td>3</td>
<td>Division abbreviation</td>
</tr>
<tr>
<td>Hiredate</td>
<td>num</td>
<td>8</td>
<td>Date of hire</td>
</tr>
<tr>
<td>Supervisor</td>
<td>char</td>
<td>20</td>
<td>Name of supervisor</td>
</tr>
</tbody>
</table>

- a Code and IDnum
- b Manager and Supervisor
- c Manager and IDnum
- d There are no variables that are common to both of these data sets.

2. Suppose you want to merge Dataset1, Dataset2, and Dataset3. Also suppose that Dataset1 and Dataset2 have the common variable Startdate, Dataset2 and Dataset3 have the common variable Instructor, and that these data sets have no other common variables. How can you use a DATA step to merge these three data sets into one new data set?

- a You use a MERGE statement in one DATA step to merge Dataset1, Dataset2, and Dataset3 by Startdate and Instructor.
b You sort Dataset1 and Dataset2 by Startdate and merge them into a temporary data set in a DATA step. Then you sort the temporary data set and Dataset3 by Instructor and merge them into a new data set in a DATA step.
c You can merge these data sets only with a PROC SQL step.
d You cannot merge these three data sets at all because they do not have a common variable.

3 Which of the following programs correctly creates a table with the results of a PROC SQL inner join matched on the values of empcode?

a proc sql;
    select newsals.empcode allemps.lastname
        newsals.salary contrib.amount
    from sasuser.allemps, sasuser.contrib,
        sasuser.newsals
    where empcode=allemps.empid
        and empcode=contrib.empid;
quit;
b proc sql;
    create table usesql as
        select newsals.empcode allemps.lastname
            newsals.salary contrib.amount
        from sasuser.allemps, sasuser.contrib,
            sasuser.newsals;
quit;
c proc sql;
    create table usesql as;
        select newsals.empcode allemps.lastname,
            newsals.salary, contrib.amount;
        from sasuser.allemps, sasuser.contrib,
            sasuser.newsals;
    where empcode=allemps.empid
        and empcode=contrib.empid;
quit;
d proc sql;
    create table usesql as
        select newsals.empcode, allemps.lastname,
            newsals.salary, contrib.amount
        from sasuser.allemps, sasuser.contrib,
            sasuser.newsals
    where empcode=allemps.empid
        and empcode=contrib.empid;
quit;

4 To process a default DATA step match-merge, SAS first reads the descriptor portion of each data set and sets up the PDV and the descriptor portion of the new data set. Which of the following accurately describes the rest of this process?

a Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. When the BY value changes in all the input data sets, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are written to the new data set.
b Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. After each DATA step iteration, the PDV is initialized to missing. Missing values for variables, as
well as missing values that result from unmatched observations, are omitted from the new data set.

c Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. Then SAS goes through the new data set and eliminates all observations that do not have matching values of the BY variable.

d Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. The new data set is then ordered by values of the BY variable.

5 Which of the following statements is false about using multiple SET statements in one DATA step?

a You can use multiple SET statements to combine observations from several SAS data sets.

b Processing stops when SAS encounters the end-of-file (EOF) marker on either data set (even if there is more data in the other data set).

c You can use multiple SET statements in one DATA step only if the data sets in each SET statement have a common variable.

d The variables in the PDV are not reinitialized when a second SET statement is executed.

6 Select the program that correctly creates a new data set named Sasuser.Summary that contains one observation with summary data created from the Salary variable of the Sasuser.Empdata data set.

a proc sum data=sasuser.emdata noprint;
   output out=sasuser.summary sum=Salarysum;
run;

b proc means data=sasuser.empdata noprint;
   var salary;
   output out=sasuser.summary sum=Salarysum;
run;

c proc sum data=sasuser.empdata noprint;
   var salary;
   output out=sasuser.summary sum=Salarysum;
run;

d proc means data=sasuser.empdata noprint;
   output=sasuser.summary sum=Salarysum;
run;

7 If the value of Cargosum is $1000 at the end of the first iteration of the DATA step shown below, what is the value of Cargosum in the PDV when the DATA step is in its third iteration?

data sasuser.percent1;
   if _n_=1 then set sasuser.summary (keep=cargosum);
   set sasuser.monthsum (keep=salemon revcargo);
   PctRev=revcargo/cargosum;
run;

a $1000
b $3000
c The value is missing.

d The value cannot be determined without seeing the data that is in Sasuser.Summary.
According to the data set shown, what is the value of Totalrev in the PDV at the end of the fourth iteration of the DATA step?

```
data sasuser.percent2(drop=totalrev);
  if _n_=1 then do until(lastobs);
    set sasuser.monthsum2(keep=revcargo) end=lastobs;
    totalrev+revcargo;
  end;
  set sasuser.monthsum2 (keep=salemon revcargo);
  PctRev=revcargo/totalrev;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>SaleMon</th>
<th>RevCargo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAN1997</td>
<td>$530.00</td>
</tr>
<tr>
<td>2</td>
<td>JAN1998</td>
<td>$230.00</td>
</tr>
<tr>
<td>3</td>
<td>JAN1999</td>
<td>$380.00</td>
</tr>
<tr>
<td>4</td>
<td>FEB1997</td>
<td></td>
</tr>
</tbody>
</table>

a. The value is missing.
b. $350.00
c. $520.00
d. $1100.00

Which of the following programs correctly uses an index to combine data from two input data sets?

```
a. data work.profit;
  set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo)
    key=flightdate;
  set sasuser.dnunder;
  Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;
```

```
b. data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo)
    key=flightdate;
  where routeid='0000103';
  Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;
```

```
c. data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo);
  key=flightdate;
  Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;
```

```
d. data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo)
    key=flightdate;
  Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;
```
10 Which of the following statements about the _IORC_ variable is false?

a. It is automatically created when you use either a SET statement with the KEY= option or the MODIFY statement with the KEY= option in a DATA step.

b. A value of zero for _IORC_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) did not execute successfully.

c. A value of zero for _IORC_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) executed successfully.

d. You can use the _IORC_ variable to prevent nonmatching data from being included when you use an index to combine data from multiple data sets.
CHAPTER 16

Using Lookup Tables to Match Data

Overview 560
   Introduction 560
   Objectives 560
Using Multidimensional Arrays 561
   Review of the Multidimensional Array Statement 561
   Example 562
Using Stored Array Values 564
   Example 564
   Creating an Array 565
   Loading the Array Elements 566
   Reading the Actual Values 569
Using PROC TRANSPOSE 570
   Example 572
   Adding Descriptive Variable Names 573
Merging the Transposed Data Set 574
   Structuring the Data for a Merge 574
   Using a BY Statement with PROC TRANSPOSE 575
   Sorting the Work.Ctarget2 Data Set 576
   Reorganizing the Sasuser.Monthsum Data Set 576
   Sorting the Work.Monthsum2 Data Set 577
   Completing the Merge 578
Using Hash Objects as Lookup Tables 579
   The Structure of a Hash Object 580
   Example 580
   Data Step Component Objects 581
   Declaring the Hash Object 582
   Instantiating the Hash Object 582
   Declaring and Instantiating the Hash Object in a Single Step 583
   Defining Keys and Data 583
   Using the Call Missing Routine 584
   Loading Key and Data Values 585
   Retrieving Matching Data 585
   Hash Object Processing 585
   Creating a Hash Object From a SAS Data Set 586
   Using a Non-Executing SET Statement 587
   Working with Multiple Data Variables 587
   Retrieving Multiple Data Values 588
   Using Return Codes with the FIND Method 589
   Example 589
Summary 592
   Text Summary 592
Overview

Introduction

Sometimes, you need to combine data from two or more sets into a single observation in a new data set according to the values of a common variable. When the data sources are two or more data sets that have a common structure, you can use a match-merge to combine the data sets. However, in some cases the data sources do not share a common structure. When data sources do not have a common structure, you can use a lookup table to match them. A lookup table is a table that contains key values.

The technique that you use to perform a table lookup is dependent on your data. This chapter focuses on using multidimensional arrays to perform table lookups and transposing SAS data sets in preparation for a match-merge.

Objectives

In this chapter, you learn to

- use a multidimensional array to match data
- use stored array values to match data
Using Lookup Tables to Match Data

- use PROC TRANSPOSE to transpose a SAS data set and prepare it for a table lookup
- merge a transposed SAS data set
- use a hash object as a lookup table (for SAS 9 and later).

Using Multidimensional Arrays

Review of the Multidimensional Array Statement

When a lookup operation depends on more than one numerical factor, you can use a multidimensional array. You use an ARRAY statement to create an array. The ARRAY statement defines a set of elements that you plan to process as a group.

General form, multidimensional ARRAY statement:

```
ARRAY array-name {rows,cols,...} <$> <length>
   <array-elements> <(initial values)>;
```

where

- `array-name` names the array.
- `rows` specifies the number of array elements in a row arrangement.
- `cols` specifies the number of array elements in a column arrangement.
- `array-elements` names the variables that make up the array.
- `initial values` specifies initial values for the corresponding elements in the array that are separated by commas or spaces.

Note: The keyword _TEMPORARY_ may be used instead of `array-elements` to avoid creating new data set variables. Only temporary array elements are produced as a result of using _TEMPORARY_.

When you are working with arrays, remember that

- the name of the array must be a SAS name that is not the name of a SAS variable in the same DATA step
- the variables listed as array elements must all be the same type (either all numeric or all character)
- the initial values specified can be numbers or character strings. You must enclose all character strings in quotation marks.

Note: If you use the _TEMPORARY_ keyword in an array statement, remember that temporary data elements behave like DATA step variables with the following exceptions:

- They do not have names. Refer to temporary data elements by the array name and dimension.
- They do not appear in the output data set.
Example

Suppose you need to determine the wind chill values for the flights represented in the SAS data set Sasuser.Flights. The data set contains three variables: Flight (the flight number), Temp (the average outdoor temperature during the flight), and Wspeed (the average wind speed during the flight).

Figure 16.1  SAS Data Set Sasuser.Flights

<table>
<thead>
<tr>
<th>Obs</th>
<th>Flight</th>
<th>Temp</th>
<th>Wspeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA2736</td>
<td>-8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>IA6352</td>
<td>-4</td>
<td>16</td>
</tr>
</tbody>
</table>

Wind chill values are derived from the air temperature and wind speed as shown in the following wind chill lookup table. To determine the wind chill for each flight, you can create a multidimensional array that stores the wind chill values shown in the table. You can then match the values of Temp and Wspeed with the wind chill values stored in the array.

Figure 16.2  Temperature (in degrees Fahrenheit)

<table>
<thead>
<tr>
<th>Wind Speed (in miles per hour)</th>
<th>-10</th>
<th>-5</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-22</td>
<td>-16</td>
<td>-11</td>
<td>-5</td>
<td>1</td>
<td>7</td>
<td>13</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>-28</td>
<td>-22</td>
<td>-16</td>
<td>-10</td>
<td>-4</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>-32</td>
<td>-26</td>
<td>-19</td>
<td>-13</td>
<td>-7</td>
<td>0</td>
<td>6</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>25</td>
<td>-37</td>
<td>-31</td>
<td>-24</td>
<td>-17</td>
<td>-11</td>
<td>-4</td>
<td>3</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>-39</td>
<td>-33</td>
<td>-26</td>
<td>-19</td>
<td>-12</td>
<td>-5</td>
<td>1</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>35</td>
<td>-41</td>
<td>-34</td>
<td>-27</td>
<td>-21</td>
<td>-14</td>
<td>-7</td>
<td>0</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>40</td>
<td>-43</td>
<td>-36</td>
<td>-29</td>
<td>-22</td>
<td>-15</td>
<td>-8</td>
<td>-1</td>
<td>6</td>
<td>13</td>
</tr>
</tbody>
</table>

In the following program, the ARRAY statement creates the two-dimensional array WC and specifies the dimensions of the array: four rows and two columns. No variables are created from the array because the keyword _TEMPORARY_ is used. The initial values specified correspond to the values in the wind chill lookup table. For this example, only the values in the first two columns and four rows in the wind chill lookup table are included in the array.

```sas
data work.wndchill (drop = column row);
array WC{4,2} _temporary_
```

- You cannot use the special subscript astersisk (*) to refer to all the elements.
- Temporary data element values are always automatically retained, rather than being reset to missing at the beginning of the next iteration of the DATA step.
Using Lookup Tables to Match Data

Example

```sas
((-22,-16,-28,-22,-32,-26,-35,-29));

set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc(row,column);
run;
```

**Figure 16.3** Temperature (in degrees Fahrenheit)

The value of **WindChill** for each flight is determined by referencing the array based on the values of **Wspeed** and **Temp** in the **Sasuser.Flights** data set. The row number for the array reference is determined by the value of **Wspeed**. The column number for the array reference is determined by the value of **Temp**.

**Table Representation of the WC Array**

```sas
data work.wndchill (drop = column row);
array WC{4,2} _temporary_
(-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc(row,column);
run;
```

The rounding unit for the value of **Wspeed** is 5 because the values for wind speed in the wind chill table are rounded to every 5 miles-per-hour. **Wspeed** is then divided by 5 to derive the row number for the array reference.

Like the value for **Wspeed**, the value of **Temp** is rounded to the nearest 5, then divided by 5. The offset of 3 is added to the value because the third column in the wind chill lookup table represents 0 degrees.

```sas
data work.wndchill (drop = column row);
array WC(4,2) _temporary_
(-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
run;
```
Using Stored Array Values

In the previous section, the wind chill values were loaded into the WC array when the array was created. In some cases, you may need to store array values in a SAS data set rather than loading them in an ARRAY statement. Array values should be stored in a SAS data set when

- there are too many values to initialize easily in the array
- the values change frequently
- the same values are used in many programs.

Example

Suppose you want to compare the actual cargo revenue values in the SAS data set Sasuser.Monthsum to the target cargo revenue values in the SAS data set Sasuser.Ctargets. Sasuser.Monthsum contains the actual cargo and passenger revenue figures for each month from 1997 through 1999.

Table 16.1 SAS Data Set Sasuser.Monthsum (first five observations of selected variables)

<table>
<thead>
<tr>
<th>Obs</th>
<th>SaleMon</th>
<th>RevCargo</th>
<th>MonthNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>2</td>
</tr>
</tbody>
</table>

The SAS data set Sasuser.Ctargets contains the target cargo revenue figures for each month from 1997 through 1999.

WindChill= wc{row,column};
run;

PROC PRINT output shows the completed data set.

proc print data=work.wndchill;
run;
Table 16.2  SAS Data Set Sasuser.Ctargets

<table>
<thead>
<tr>
<th>Obs</th>
<th>Year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1997</td>
<td>192284420</td>
<td>86376721</td>
<td>28526103</td>
<td>260386468</td>
<td>109975326</td>
<td>102833104</td>
</tr>
<tr>
<td>2</td>
<td>1998</td>
<td>108645734</td>
<td>147656369</td>
<td>202158055</td>
<td>41160707</td>
<td>264294440</td>
<td>267135485</td>
</tr>
<tr>
<td>3</td>
<td>1999</td>
<td>85730444</td>
<td>74168740</td>
<td>39955768</td>
<td>312654811</td>
<td>318149340</td>
<td>187270927</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>196728648</td>
<td>236996122</td>
<td>112413744</td>
<td>125401565</td>
<td>72551855</td>
<td>136042505</td>
</tr>
<tr>
<td>2</td>
<td>208694865</td>
<td>83456868</td>
<td>286846554</td>
<td>275721406</td>
<td>230488351</td>
<td>24901752</td>
</tr>
<tr>
<td>3</td>
<td>123394421</td>
<td>34273985</td>
<td>151565752</td>
<td>141528519</td>
<td>178043261</td>
<td>181668256</td>
</tr>
</tbody>
</table>

You want to create a new SAS data set, Work.Lookup1, that lists the actual and target values for each month. Work.Lookup1 should have the same structure as Sasuser.Monthsum: an observation for each month and year, as well as a new variable, Ctarget (target cargo revenues). The value of Ctarget is derived from the target values in Sasuser.Ctargets.

Table 16.3  SAS Data Set Work.Lookup1 (first five observations of selected variables)

<table>
<thead>
<tr>
<th>Obs</th>
<th>SaleMon</th>
<th>RevCargo</th>
<th>Ctarget</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>$192,284,420.00</td>
</tr>
<tr>
<td>2</td>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>$108,645,734.00</td>
</tr>
<tr>
<td>3</td>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>$85,730,444.00</td>
</tr>
<tr>
<td>4</td>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>$86,376,721.00</td>
</tr>
<tr>
<td>5</td>
<td>FEB1998</td>
<td>$215,959,695.50</td>
<td>$147,656,369.00</td>
</tr>
</tbody>
</table>

Sasuser.Monthsum and Sasuser.Ctargets cannot be merged because they have different structures:

- Sasuser.Monthsum has an observation for each month and year.
- Sasuser.Ctargets has one column for each month and one observation for each year.

However, the data sets have two common factors: month and year. You can use a multidimensional array to match the actual values for each month and year in Sasuser.Monthsum with the target values for each month and year in Sasuser.Ctargets.

Creating an Array

The first step is to create an array to hold the values in the target data set, Sasuser.Ctargets. The array needs two dimensions: one for the year values and one for the month values. In the following program, the first ARRAY statement creates the two-dimensional array, Targets.

Remember that the index of an array does not have to range from one to the number of elements. You can specify a range for the values for the index when you define the
array. In this case, the dimensions of the array are specified as three rows (one for each year: 1997, 1998, and 1999) and 12 columns (one for each month).

```sas
data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.ctargets;
  array mnth{*} Jan--Dec;
  do j=1 to dim(mnth);
    targets{year,j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarget dollar15.2;
run;
```

The following table represents the `Targets` array. Notice that the array is not populated. The next step is to load the array elements from `Sasuser.Ctargets`.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The row dimension for the `Targets` array could have been specified using the value 3. For example,

```sas
array Targets{3,12} _temporary_;
```

However, using the notation `1997:1999` simplifies the program by eliminating the need to map numeric values to the year values.

---

### Loading the Array Elements

The `Targets` array needs to be loaded with the values in `Sasuser.Ctargets`. One method for accomplishing this task is to load the array within a DO loop.
Using Lookup Tables to Match Data

Table 16.5  SAS Data Set Sasuser.Ctargets

<table>
<thead>
<tr>
<th>Year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>192284420</td>
<td>86376721</td>
<td>28526103</td>
<td>260386468</td>
<td>109975326</td>
<td>102833104</td>
</tr>
<tr>
<td>1998</td>
<td>108645734</td>
<td>147656369</td>
<td>202158055</td>
<td>41160707</td>
<td>264294440</td>
<td>267135485</td>
</tr>
<tr>
<td>1999</td>
<td>85730444</td>
<td>74168740</td>
<td>39955768</td>
<td>312654811</td>
<td>318149340</td>
<td>187270927</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>196728648</td>
<td>236996122</td>
<td>112413744</td>
<td>125401565</td>
<td>72551855</td>
<td>136042505</td>
</tr>
<tr>
<td>208694865</td>
<td>83456868</td>
<td>286846554</td>
<td>275721406</td>
<td>230488351</td>
<td>24901752</td>
</tr>
<tr>
<td>123394421</td>
<td>34273985</td>
<td>151565752</td>
<td>141528519</td>
<td>178043261</td>
<td>181668256</td>
</tr>
</tbody>
</table>

The IF-THEN statement specifies that the Targets array is loaded only once, during the first iteration of the DATA step. The DO loop executes three times, once for each observation in Sasuser.Ctargets.

The ARRAY statement within the DO loop creates the Mnth array, which will be used to store the elements from Sasuser.Ctargets. The dimensions of the Mnth array are specified using an asterisk, which enables SAS to automatically count the array elements.

Note: If you use an asterisk to specify the dimensions of an array, you must list the array elements. You cannot use an asterisk to specify an array’s dimensions if the elements of the array are specified with the _TEMPORARY_ keyword.

The array elements Jan through Dec are listed using a double hyphen (- -). The double hyphen (- -) is used to read the specified values based on their positions in the PDV instead of alphabetically.

```
data work.lookup1;
  array Targets{1997:1999,12} _temporary_;
  if _n_=1 then do i= 1 to 3;
    set sasuser.ctargets;
    array Mnth{*} Jan--Dec;
    do j=1 to dim(mnth);
      targets{year,j}=mnth{j};
    end;
  end;
  set sasuser.monthsum(keep=salemon revcargo monthno);
  year=input(substr(salemon,4),4.);
  Ctarget=targets{year,monthno};
  format ctarget dollar15.2;
run;
```

The following table shows the values in the Mnth array after the first iteration of the DO loop.
Table 16.6  Table Representation of Mnth Array (after the first iteration of the DO loop)

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar...</th>
<th>...Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>192284420</td>
<td>86376721</td>
<td>260386468</td>
<td>125401565</td>
<td>72551855</td>
<td>136042505</td>
</tr>
</tbody>
</table>

Within the nested DO loop, the **Targets** array reference is matched to the **Mnth** array reference in order to populate the **Targets** array. The DIM function returns the number of elements in the **Mnth** array (in this case 12) and provides an ending point for the nested DO loop.

```sas
data work.lookup1;
  array Targets{1997:1999,12} _temporary_;
  if _n_=1 then do i= 1 to 3;
    set sasuser.ctargets;
    array Mnth{*} Jan--Dec;
    do j=1 to dim(mnth);
      targets{year,j}=mnth{j};
    end;
  end;
  set sasuser.monthsum(keep=salemon revcargo monthno);
  year=input(substr(salemon,4),4.);
  Ctarget=targets{year,monthno};
  format ctarget dollar15.2;
run;
```

Table 16.7  Table Representation of Mnth Array (after the second iteration of the DO loop)

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar...</th>
<th>...Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>108645734</td>
<td>147656369</td>
<td>202158055</td>
<td>275721406</td>
<td>230488351</td>
<td>24901752</td>
</tr>
</tbody>
</table>

Table 16.8  Table Representation of Mnth Array (after the third iteration of the DO loop)

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar...</th>
<th>...Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>85730444</td>
<td>74168740</td>
<td>39955768</td>
<td>141528519</td>
<td>178043261</td>
<td>181688256</td>
</tr>
</tbody>
</table>

Table 16.9  Table Representation of Populated Targets Array

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3...</th>
<th>...10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>192284420</td>
<td>86376721</td>
<td>260386468</td>
<td>125401565</td>
<td>72551855</td>
</tr>
<tr>
<td>1998</td>
<td>108645734</td>
<td>147656369</td>
<td>202158055</td>
<td>275721406</td>
<td>230488351</td>
</tr>
<tr>
<td>1999</td>
<td>85730444</td>
<td>74168740</td>
<td>39955768</td>
<td>141528519</td>
<td>178043261</td>
</tr>
</tbody>
</table>

**Note:** The dimension of the **Mnth** array could also be specified using the numeric value 12. However, the asterisk notation enables the program to be more flexible. For example, using the asterisk, the program would not need to be edited if the target data set contained data for only eleven months. Remember that if you use an asterisk to count the array elements, you must list the array elements. △
Reading the Actual Values

The last step is to read the actual values stored in `Sasuser.Monthsum`. Remember that you need to know the month and year values for each observation in order to locate the correct target revenue values.

Table 16.10 SAS Data Set Sasuser.Monthsum (first five observations of selected variables)

<table>
<thead>
<tr>
<th>SaleMon</th>
<th>RevCargo</th>
<th>MonthNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>1</td>
</tr>
<tr>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>1</td>
</tr>
<tr>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>1</td>
</tr>
<tr>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>2</td>
</tr>
<tr>
<td>FEB1998</td>
<td>$215,959,695.50</td>
<td>2</td>
</tr>
</tbody>
</table>

The values for month are read in from `MonthNo`. The year values are contained within the values of `SaleMon` and can be extracted using the SUBSTR function. In this example, the SUBSTR function brings in four characters from `SaleMon`, starting at the fourth character. Note that the INPUT function is used to convert the value that is extracted from `SaleMon` from character to numeric in the assignment statement for `Year`. A numeric format must be used because the value of `Year` will be used as an array reference.

The values of `Ctarget` are then read in from the `Targets` array based on the value of `Year` and `MonthNo`.

```sas
data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
set sasuser.ctargets;
array Mnth{*} Jan--Dec;
do j=1 to dim(mnth);
  targets{year,j}=mnth{j};
end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarget dollar15.2;
run;
```

Table 16.11 Table Representation of Targets Array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>192284420</td>
<td>86376721</td>
<td>260386468</td>
<td>125401565</td>
<td>72551855</td>
<td>136042505</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>108645734</td>
<td>147656369</td>
<td>202158055</td>
<td>275721406</td>
<td>230488351</td>
<td>24901752</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>85730444</td>
<td>74168740</td>
<td>39955768</td>
<td>141528519</td>
<td>178043261</td>
<td>181668256</td>
<td></td>
</tr>
</tbody>
</table>

PROC PRINT output shows the new data set `Work.Lookup1`, which contains the actual cargo values (`RevCargo`) and the target cargo values (`Ctarget`).
**Using PROC TRANSPOSE**

In the previous section, we compared actual revenue values to target revenue values using an array as a lookup table. Remember that

- **Sasuser.Monthsum** has an observation for each month and year.

<table>
<thead>
<tr>
<th>TABLE 16.12</th>
<th>SAS Data Set Sasuser.Monthsum (first five observations of selected variables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SaleMon</td>
<td>RevCargo</td>
</tr>
<tr>
<td>JAN1997</td>
<td>$171,520,869.10</td>
</tr>
<tr>
<td>JAN1998</td>
<td>$238,786,807.60</td>
</tr>
<tr>
<td>JAN1999</td>
<td>$280,350,393.00</td>
</tr>
<tr>
<td>FEB1997</td>
<td>$177,671,530.40</td>
</tr>
<tr>
<td>FEB1998</td>
<td>$215,959,695.50</td>
</tr>
</tbody>
</table>

- **Sasuser.Ctargets** has one variable for each month and one observation for each year.

<table>
<thead>
<tr>
<th>TABLE 16.13</th>
<th>SAS Data Set Sasuser.Ctargets (selected variables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>Jan</td>
</tr>
<tr>
<td>1997</td>
<td>192284420</td>
</tr>
<tr>
<td>1998</td>
<td>108645734</td>
</tr>
<tr>
<td>1999</td>
<td>85730444</td>
</tr>
</tbody>
</table>

Using arrays was a good solution because the orientation of the data sets differed. An alternate solution is to transpose **Sasuser.Ctargets** using PROC TRANSPOSE, and then merge the transposed data set with **Sasuser.Monthsum** by the values of **Year** and **Month**.
General form, PROC TRANSPOSE:

```
PROC TRANSPOSE <DATA=input-data-set>
   <OUT=output-data-set>
   <NAME=variable-name>
   <PREFIX=variable-name>;
BY <DESCENDING> variable-1
   <....<DESCENDING> variable-n>
   <NOTSORTED>;
VAR variable(s);
RUN;
```

where

- `DATA=input-data-set` names the SAS data set to transpose.
- `OUT=output-data-set` names the output data set.
- `NAME=variable-name` specifies the name for the variable in the output data set that contains the name of the variable that is being transposed to create the current observation.
- `PREFIX=variable-name` specifies a prefix to use in constructing names for transposed variables in the output data set. For example, if `PREFIX=VAR`, the names of the variables are VAR1, VAR2, ..., VARn.
- `BY` statement is used to transpose each BY group.
- `VAR variable(s)` names one or more variables to transpose.

**Note:** If `output-data-set` does not exist, PROC TRANSPOSE creates it by using the DATA n naming convention.

**Note:** If you omit the `VAR` statement, the TRANSPOSE procedure transposes all of the numeric variables in the input data set that are not listed in another statement.

**Note:** You must list character variables in a `VAR` statement if you want to transpose them.

The TRANSPOSE procedure creates an output data set by restructuring the values in a SAS data set. When the data set is restructured, selected variables are transposed into observations. The TRANSPOSE procedure can often eliminate the need to write a lengthy DATA step to achieve the same result. The output data set can be used in subsequent DATA or PROC steps for analysis, reporting, or further data manipulation.

PROC TRANSPOSE does not print the output data set. Use PROC PRINT, PROC REPORT, or some other SAS reporting tool to print the output data set.
The following program transposes the SAS data set Sasuser.Ctargets. The OUT= option specifies the name of the output data set, Work.Ctarget2. All of the variables in Sasuser.Ctargets are transposed because all of the variables are numeric and a VAR statement is not used in the program.

```
proc transpose data=sasuser.ctargets 
   out=work.ctarget2;
run;
```

<table>
<thead>
<tr>
<th>Table 16.14</th>
<th>Input Data Set Sasuser.Ctargets (selected variables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Obs</strong></td>
<td><strong>Year</strong></td>
</tr>
<tr>
<td>1</td>
<td>1997</td>
</tr>
<tr>
<td>2</td>
<td>1998</td>
</tr>
<tr>
<td>3</td>
<td>1999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 16.15</th>
<th>Output Data Set: Work.Ctarget2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Obs</strong></td>
<td><strong><em>NAME</em></strong></td>
</tr>
<tr>
<td>1</td>
<td>Year</td>
</tr>
<tr>
<td>2</td>
<td>Jan</td>
</tr>
<tr>
<td>3</td>
<td>Feb</td>
</tr>
<tr>
<td>4</td>
<td>Mar</td>
</tr>
</tbody>
</table>
Using Lookup Tables to Match Data

Adding Descriptive Variable Names

Notice that in the output data set, the variables are named `_NAME_`, `COL1`, `COL2`, and `COL3`.

>_NAME_ is the default name of the variable that PROC TRANSPOSE creates to identify the source of the values in each observation in the output data set. This variable is a character variable whose values are the names of the variables that are transposed from the input data set. For example, in `Work.Ctarget2` the values in the first observation in the output data set come from the values of the variable `Year` in the input data set.

The remaining transposed variables are named `COL1`...`COLn` by default. In `Work.Ctarget2`, the values of the variables `COL1`, `COL2`, and `COL3` represent the target cargo revenue for each month in the years 1997, 1998, and 1999.

**Adding Descriptive Variable Names**

You can use PROC TRANSPOSE options to give the variables in the output data set more descriptive names. The `NAME=` option specifies a name for the `_NAME_` variable.

The `PREFIX=` option specifies a prefix to use in constructing names for transposed variables in the output data set. For example, if `PREFIX=Ctarget`, the names of the variables are `Ctarget1`, `Ctarget2`, and `Ctarget3`.

```
proc transpose data=sasuser.ctargets
  out=work.ctarget2
  name=Month
  prefix=Ctarget;
run;
```

**Table 16.16**  Output Data Set: `Work.Ctarget2`

<table>
<thead>
<tr>
<th>Obs</th>
<th>Month</th>
<th>Ctarget1</th>
<th>Ctarget2</th>
<th>Ctarget3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Year</td>
<td>1997</td>
<td>1997</td>
<td>1997</td>
</tr>
<tr>
<td>2</td>
<td>Jan</td>
<td>192284420</td>
<td>108645734</td>
<td>85730444</td>
</tr>
<tr>
<td>3</td>
<td>Feb</td>
<td>86376721</td>
<td>147656369</td>
<td>74168740</td>
</tr>
<tr>
<td>4</td>
<td>Mar</td>
<td>28526103</td>
<td>202158055</td>
<td>39955768</td>
</tr>
<tr>
<td>5</td>
<td>Apr</td>
<td>260386468</td>
<td>41160707</td>
<td>31265481</td>
</tr>
<tr>
<td>6</td>
<td>May</td>
<td>109975326</td>
<td>264294440</td>
<td>318149340</td>
</tr>
<tr>
<td>7</td>
<td>Jun</td>
<td>102833104</td>
<td>267135485</td>
<td>187270927</td>
</tr>
</tbody>
</table>
### Merging the Transposed Data Set

#### Structuring the Data for a Merge

Remember that the transposed data set, `Work.Ctarget2`, needs to be merged with `Sasuser.Monthsum` by the values of `Year` and `Month`. Neither data set is currently structured correctly for the merge.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Month</th>
<th>Ctarget1</th>
<th>Ctarget2</th>
<th>Ctarget3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Year</td>
<td>1997</td>
<td>1998</td>
<td>1999</td>
</tr>
<tr>
<td>2</td>
<td>Jan</td>
<td>192284420</td>
<td>108645734</td>
<td>85730444</td>
</tr>
<tr>
<td>3</td>
<td>Feb</td>
<td>86376721</td>
<td>147656369</td>
<td>74168740</td>
</tr>
</tbody>
</table>

*Note:* The RENAME=data set option can also be used with PROC TRANSPOSE to change variable names.

```sas
proc transpose data=sasuser.ctargets
  out=work.ctarget2 (rename=(col1=Ctarget1 col2=Ctarget2 col3=Ctarget3))
  name=Month;
run;
```

The default label for the `_NAME_` variable is `NAME OF FORMER VARIABLE`. To see this, print the transposed data set using PROC PRINT with the LABEL option. You can use a LABEL statement to override the default label.

```sas
proc transpose data=sasuser.ctargets
  out=work.ctarget2 name=Month prefix=Ctarget;
run;
proc print data=work.ctarget2 label;
label Month=MONTH;
run;
```
Using Lookup Tables to Match Data

Using a BY Statement with PROC TRANSPOSE

In order to correctly structure Work.Ctarget2 for the merge, a BY statement needs to be used with PROC TRANSPOSE. For each BY group, PROC TRANSPOSE creates one observation for each variable that it transposes. The BY variable itself is not transposed.

The following program transposes Sasuser.Ctargets by the value of Year. The resulting output data set, Work.Ctarget2, now contains 12 observations for each each year (1997, 1998, and 1999).

```sas
proc transpose data=sasuser.ctargets
   out=work.ctarget2
   name=Month
   prefix=Ctarget;
   by year;
run;
```

Table 16.19 Input Data Set Sasuser.Ctargets (selected variables)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1997</td>
<td>192284420</td>
<td>86376721</td>
<td>28526103</td>
<td>260386468</td>
<td>109975326</td>
<td>102833104</td>
</tr>
<tr>
<td>2</td>
<td>1998</td>
<td>108645734</td>
<td>147656369</td>
<td>202158055</td>
<td>41160707</td>
<td>264294440</td>
<td>267135485</td>
</tr>
<tr>
<td>3</td>
<td>1999</td>
<td>85730444</td>
<td>74168740</td>
<td>39955768</td>
<td>31265481</td>
<td>318149340</td>
<td>187270927</td>
</tr>
</tbody>
</table>

Table 16.20 Output Data Set Work.Ctarget2 (first 12 observations)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Year</th>
<th>Month</th>
<th>Ctarget1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1997</td>
<td>Jan</td>
<td>192284420</td>
</tr>
<tr>
<td>2</td>
<td>1997</td>
<td>Feb</td>
<td>86376721</td>
</tr>
<tr>
<td>3</td>
<td>1997</td>
<td>Mar</td>
<td>28526103</td>
</tr>
<tr>
<td>4</td>
<td>1997</td>
<td>Apr</td>
<td>260386468</td>
</tr>
<tr>
<td>5</td>
<td>1997</td>
<td>May</td>
<td>109975326</td>
</tr>
</tbody>
</table>
Chapter 16

SORTING THE WORK.CTARGET2 DATA SET

The last step in preparing Work.Ctarget2 for the merge is to use the SORT procedure to sort the data set by Year and Month as shown in the following program:

```
proc sort data=work.ctarget2;
    by year month;
run;
```

Notice that in the sorted version of Work.Ctarget2, the values of month are sorted alphabetically by year.

**Table 16.21** SAS Data Set Work.Ctarget2 (sorted, first 12 observations)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Year</th>
<th>Month</th>
<th>Ctarget1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1997</td>
<td>Apr</td>
<td>260386468</td>
</tr>
<tr>
<td>2</td>
<td>1997</td>
<td>Aug</td>
<td>236996122</td>
</tr>
<tr>
<td>3</td>
<td>1997</td>
<td>Dec</td>
<td>136042505</td>
</tr>
<tr>
<td>4</td>
<td>1997</td>
<td>Feb</td>
<td>86376721</td>
</tr>
<tr>
<td>5</td>
<td>1997</td>
<td>Jan</td>
<td>192284420</td>
</tr>
<tr>
<td>6</td>
<td>1997</td>
<td>Jul</td>
<td>196728648</td>
</tr>
<tr>
<td>7</td>
<td>1997</td>
<td>Jun</td>
<td>102833104</td>
</tr>
<tr>
<td>8</td>
<td>1997</td>
<td>Mar</td>
<td>28526103</td>
</tr>
<tr>
<td>9</td>
<td>1997</td>
<td>May</td>
<td>109975326</td>
</tr>
<tr>
<td>10</td>
<td>1997</td>
<td>Nov</td>
<td>725518555</td>
</tr>
<tr>
<td>11</td>
<td>1997</td>
<td>Oct</td>
<td>125401565</td>
</tr>
<tr>
<td>12</td>
<td>1997</td>
<td>Sep</td>
<td>112413744</td>
</tr>
</tbody>
</table>

CAUTION:

The original SAS data set must be sorted or indexed prior to using a BY statement with PROC TRANSPOSE unless you use the NOTSORTED option.

**Reorganizing the Sasuser.Monthsum Data Set**

The data in Sasuser.Monthsum must also be reorganized for the merge because the month and year values in that data set are combined in the variable SaleMon.
Table 16.22 SAS Data Set Sasuser.Monthsum (first five observations of selected variables)

<table>
<thead>
<tr>
<th>Obs</th>
<th>SaleMon</th>
<th>RevCargo</th>
<th>MonthNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>FEB1998</td>
<td>$215,959,695.50</td>
<td>2</td>
</tr>
</tbody>
</table>

The following program creates two new variables, *Year* and *Month*, to hold the year and month values. The values for *Year* are created from *MonthSum* using the INPUT and SUBSTR functions. The values for *Month* are extracted from *MonthSum* using the LOWCASE and SUBSTR functions.

```sas
data work.mnthsum2;
  set sasuser.monthsum(keep=SaleMon RevCargo);
  length Month $ 8;
  Year=input(substr(SaleMon,4),4.);
  Month=substr(SaleMon,1,1)||lowcase(substr(SaleMon,2,2));
run;
```

Table 16.23 SAS Data Set Work.Mnthsum2 (first six observations)

<table>
<thead>
<tr>
<th>Obs</th>
<th>SaleMon</th>
<th>RevCargo</th>
<th>Month</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAN1997</td>
<td>$171,520,869.10</td>
<td>Jan</td>
<td>1997</td>
</tr>
<tr>
<td>2</td>
<td>JAN1998</td>
<td>$238,786,807.60</td>
<td>Jan</td>
<td>1998</td>
</tr>
<tr>
<td>3</td>
<td>JAN1999</td>
<td>$280,350,393.00</td>
<td>Jan</td>
<td>1999</td>
</tr>
<tr>
<td>4</td>
<td>FEB1997</td>
<td>$177,671,530.40</td>
<td>Feb</td>
<td>1997</td>
</tr>
<tr>
<td>5</td>
<td>FEB1998</td>
<td>$215,959,695.50</td>
<td>Feb</td>
<td>1998</td>
</tr>
<tr>
<td>6</td>
<td>FEB1999</td>
<td>$253,999,924.00</td>
<td>Feb</td>
<td>1999</td>
</tr>
</tbody>
</table>

**Sorting the Work.Mnthsum2 Data Set**

As with *Work.Ctarget2*, the last step in preparing for the merge is to sort the data set by the values of *Year* and *Month* as shown in the following program:

```sas
proc sort data=work.mnthsum2;
  by year month;
run;
```

Notice that in the sorted version of *Work.Mnthsum2*, the values of month are sorted alphabetically by year.
Completing the Merge

When the data is structured correctly, `Work.Mnthsum2` and `Work.Ctarget2` can be merged by the values of `Year` and `Month` as shown in the following program:

```sas
data work.merged;
  merge work.mnthsum2 work.ctarget2;
  by year month;
run;
```

| Table 16.24 SAS Data Set Work.Mnthsum3 (sorted, first twelve observations) |
|---|---|---|---|
| Obs | SaleMon | RevCargo | Month | Year |
| 1   | APR1997  | $380,804,120.20 | Apr   | 1997 |
| 3   | DEC1997  | $196,504,413.00 | Dec   | 1997 |
| 4   | FEB1997  | $177,671,530.40 | Feb   | 1997 |
| 5   | JAN1997  | $171,520,869.10 | Jan   | 1997 |
| 6   | JUL1997  | $197,163,278.20 | Jul   | 1997 |
| 7   | JUN1997  | $190,560,828.50 | Jun   | 1997 |
| 8   | MAR1997  | $196,591,378.20 | Mar   | 1997 |
| 9   | MAY1997  | $196,261,573.20 | May   | 1997 |
| 10  | NOV1997  | $190,228,066.70 | Nov   | 1997 |
| 12  | SEP1997  | $190,535,012.50 | Sep   | 1997 |

| Table 16.25 SAS Data Set Work.Mnthsum2 (first five observations) |
|---|---|---|---|
| Obs | SaleMon | RevCargo | Month | Year |
| 1   | APR1997  | $380,804,120.20 | Apr   | 1997 |
| 3   | DEC1997  | $196,504,413.00 | Dec   | 1997 |
| 4   | FEB1997  | $177,671,530.40 | Feb   | 1997 |
| 5   | JAN1997  | $171,520,869.10 | Jan   | 1997 |

| Table 16.26 SAS Data Set Work.Ctarget2 (first five observations) |
|---|---|---|
| Obs | Year | Month | Ctarget1 |
| 1   | 1997 | Apr   | 260386468 |
| 2   | 1997 | Aug   | 236996122 |
| 3   | 1997 | Dec   | 136042505 |
PROC PRINT output shows the resulting data set `Work.Merged`. The values of `RevCargo` represent the actual cargo revenue for each month. The values of `Ctarget1` represent the target cargo values for each month.

```
proc print data=work.merged;
  format ctarget1 dollar15.2;
  var month year revcargo ctarget1;
run;
```

Using Hash Objects as Lookup Tables

Beginning with SAS 9, the hash object is available for use in a DATA step. The hash object provides an efficient, convenient mechanism for quick data storage and retrieval.

Unlike an array, which uses a series of consecutive integers to address array elements, a hash object can use any combination of numeric and character values as addresses. A hash object can be loaded from hard-coded values or a SAS data set, is sized dynamically, and exists for the duration of the DATA step.

The hash object is a good choice for lookups using unordered data that can fit into memory because it provides in-memory storage and retrieval and does not require the data to be sorted or indexed.
The Structure of a Hash Object

When a lookup operation depends on one or more key values, you can use the hash object. A hash object resembles a table with rows and columns and contains a key component and a data component.

The key component
- might consist of numeric and character values
- maps key values to data rows
- must be unique
- can be a composite.

The data component
- can contain multiple data values per key value
- can consist of numeric and character values.

Example

Suppose you have a data set, named Sasuser.Contrib, that lists the quarterly contributions to a retirement fund. You can use the hash object to calculate the difference between the actual contribution and the goal amount.
The following program creates a hash object that stores the quarterly goal for employee contributions to the retirement fund. To calculate the difference between actual contribution and the goal amount, the program retrieves the goal amount from the hash object based on the key values.

```sas
data work.difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash goal( );
    goal.definekey("QtrNum");
    goal.definedata("GoalAmount");
    goal.done( );
    call missing(qtrnum, goalamount);
    goal.add(key:'qtr1', data:10 );
    goal.add(key:'qtr2', data:15 );
    goal.add(key:'qtr3', data: 5 );
    goal.add(key:'qtr4', data:15 );
  end;
set sasuser.contrib;
goal.find();
  Diff = amount - goalamount;
run;
```

Let’s see how the hash object is set up.

### Data Step Component Objects

The hash object is a DATA step component option. Component objects are data elements that consist of attributes and methods. Attributes are the properties that specify the information that is associated with an object. An example is size. Methods define the operations that an object can perform.

To use a DATA step component object in your SAS program, you must first declare and create (instantiate) the object.
Declaring the Hash Object

You declare a hash object using the DECLARE statement.

```
DECLARE object variable <(<argument_tag-1: value-1<, ...argument_tag-n: value-n>>)>
```

where

- `object` specifies the component object.
- `variable` specifies the variable name for the component object.
- `arg_tag` specifies the information that is used to create an instance of the component object.
- `value` specifies the value for an argument tag. Valid values depend on the component object.

Valid values for `object` are as follows:
- `hash` indicates a hash object.
- `hiter` indicates a hash iterator object.

**Note:** The hash iterator object retrieves data from the hash object in ascending or descending key order.

The following DECLARE statement creates a hash object named `Goal`.

```
data work.difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash goal;
  end;
```

At this point, you have only declared the variable `Goal`. It has the potential to hold a component object of the type hash.

**Note:** The DECLARE statement is an executable statement.

Instantiating the Hash Object

You use the `_NEW_` statement to instantiate the hash object.
Using Lookup Tables to Match Data

General form, _NEW_ statement:

\[ \text{variable} = \text{NEW}\text{object(</argument_tag-1: value-1</argument_tag-n: value-n>>>)}; \]

where

- \text{variable} specifies the variable name for the component object.
- \text{object} specifies the component object.
- \text{argument_tag} specifies the information that is used to create an instance of the component object.
- \text{value} specifies the value for an argument tag. Valid values depend on the component object.

Valid values for \text{object} are as follows:
- \text{hash} indicates a hash object.
- \text{hiter} indicates a hash iterator object.

The following _NEW_ statement creates an instance of the hash object and assigns it to the variable \text{Goal}.

```plaintext
data work.difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash goal;
    goal= _new_ hash();
end;
```

**Declaring and Instantiating the Hash Object in a Single Step**

As an alternative to the two-step process of using the DECLARE and _NEW_ statements to declare and instantiate a component object, you can use the DECLARE statement to declare and instantiate the component object in one step.

```plaintext
data Work.Difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash Goal();
  end;
```

**Defining Keys and Data**

Remember that the hash object uses lookup keys to store and retrieve data. The keys and the data are DATA step variables that you use to initialize the hash object by using dot notation method calls.
General form, dot notation method calls:

\[
\text{object.method(}<\text{argument_tag-1}: \text{value-1}<, \ldots, \text{argument_tag-n}: \text{value-n}>>)\]

where

- object specifies the variable name for the DATA step component object.
- method specifies the name of the method to invoke.
- argument-tag identifies the arguments that are passed to the method.
- value specifies the argument value.

A key is defined by passing the key variable name to the DEFINEKEY method. Data is defined by passing the data variable name to the DEFINEDATA method. When all key and data variables are defined, the DEFINEDONE method is called. Keys and data can consist of any number of character or numeric data step variables.

The following code initializes the key variable `QtrNum` and the data variable `GoalAmount`.

```
data work.difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash goal();
    goal.definekey("QtrNum");
    goal.definedata("GoalAmount");
    goal.definedone();
```

**Using the Call Missing Routine**

The hash object does not assign values to key variables, and the SAS compiler cannot detect the implicit key and data variable assignments done by the hash object. Therefore, if no explicit assignment to a key or data variable appears in the program, SAS issues a note stating that the variables are uninitialized.

To avoid receiving these notes, use the CALL MISSING routine with the key and data variables as parameters. The CALL MISSING routine assigns a missing value to the specified character or numeric variables.

```
data Work.Difference (drop= goalamount);
  length GoalAmount 8;
  if _N_ = 1 then do; declare hash goal();
    goal.definekey("QtrNum");
    goal.definedata("GoalAmount");
    goal.definedone();
    call missing(qtrnum, goalamount);
```

*Note:* Another way to avoid receiving notes stating that the variables are uninitialized is to provide an initial assignment statement that assigns a missing value to each key and data variable.
Loading Key and Data Values

So far, you've declared and instantiated the hash object, and initialized the hash object's key and data variables. You are now ready to store data in the hash object using the ADD method. The following code uses the ADD method to load the key values qtr1, qtr2, qtr3, and qtr4 and the corresponding data values 10, 15, 5, and 15 into the hash object.

```sas
data work.difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash goal();
    goal.definekey("QtrNum");
    goal.definedata("GoalAmount");
    goal.definedone();
    call missing(qtrnum, goalamount);
    goal.add(key:'qtr1', data:10);
    goal.add(key:'qtr2', data:15);
    goal.add(key:'qtr3', data:5);
    goal.add(key:'qtr4', data:15);
  end;
```

Retrieving Matching Data

You use the FIND method to retrieve matching data from the hash object. The FIND method returns a value that indicates whether the key is in the hash object. If the key is in the hash object, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call.

```sas
   set sasuser.contrib;
   goal.find();
   Diff = amount - goalamount;
run;
```

Hash Object Processing

Let's take a closer look at what happens when the program is submitted for execution.

```sas
  data Work.Difference (drop= goalamount);
    length goalamount 8;
    if _N_ = 1 then do;
```
declare hash goal( );
goal.definekey("QtrNum");
goal.definedata("GoalAmount");
goal.definedone( );
call missing(qtrnum, goalamount);
goal.add(key:'qtr1', data:10 );
goal.add(key:'qtr2', data:15 );
goal.add(key:'qtr3', data: 5 );
goal.add(key:'qtr4', data:15 );
end;
set sasuser.contrib;
goal.find();
Diff = amount - goalamount;
run;

The program executes until the DATA step encounters the end of the line. PROC PRINT output shows the completed data set.

proc print data=work.difference;
run;

Table 16.27 SAS Data Set Sasuser.Revenue (first five observations)

<table>
<thead>
<tr>
<th>Origin</th>
<th>Dest</th>
<th>FlightID</th>
<th>Date</th>
<th>Rev1st</th>
<th>RevBusiness</th>
<th>RevEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANC</td>
<td>RDU</td>
<td>IA03400</td>
<td>02DEC1999</td>
<td>15829</td>
<td>28420</td>
<td>68688</td>
</tr>
<tr>
<td>ANC</td>
<td>RDU</td>
<td>IA03400</td>
<td>14DEC1999</td>
<td>20146</td>
<td>26460</td>
<td>72981</td>
</tr>
<tr>
<td>ANC</td>
<td>RDU</td>
<td>IA03400</td>
<td>26DEC1999</td>
<td>20146</td>
<td>23520</td>
<td>59625</td>
</tr>
</tbody>
</table>
Using Lookup Tables to Match Data

Working with Multiple Data Variables

<table>
<thead>
<tr>
<th>Origin</th>
<th>Dest</th>
<th>FlightID</th>
<th>Date</th>
<th>Rev1st</th>
<th>RevBusiness</th>
<th>RevEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANC</td>
<td>RDU</td>
<td>IA03401</td>
<td>09DEC1999</td>
<td>15829</td>
<td>22540</td>
<td>58671</td>
</tr>
<tr>
<td>ANC</td>
<td>RDU</td>
<td>IA03401</td>
<td>21DEC1999</td>
<td>20146</td>
<td>22540</td>
<td>65826</td>
</tr>
</tbody>
</table>

Table 16.28 SAS Data Set Sasuser.Acities (first five observations)

<table>
<thead>
<tr>
<th>City</th>
<th>Code</th>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auckland</td>
<td>AKL</td>
<td>International</td>
<td>New Zealand</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>AMS</td>
<td>Schiphol</td>
<td>Netherlands</td>
</tr>
<tr>
<td>Anchorage, AK</td>
<td>ANC</td>
<td>Anchorage International Airport</td>
<td>USA</td>
</tr>
<tr>
<td>Stockholm</td>
<td>ARN</td>
<td>Arlanda</td>
<td>Sweden</td>
</tr>
<tr>
<td>Athens (Athinai)</td>
<td>ATH</td>
<td>Hellinikon International Airport</td>
<td>Greece</td>
</tr>
</tbody>
</table>

To create the report, you can use a hash object to retrieve matching airport data from Sasuser.Acities.

Using a Non-Executing SET Statement

To initialize the attributes of hash variables that originate from an existing SAS data set, you can use a non-executing SET statement.

Because the IF condition is false during execution, the SET statement is compiled but not executed. The PDV is created with the variable `Code`, `City`, and `Name` from Sasuser.Acities.

```sas
data work.report;
  if _N_=1 then do;
    if 0 then
      set sasuser.acities (keep=Code City Name);
  end;
```

When you use this technique, the MISSING routine is not required.

Working with Multiple Data Variables

The hash object that you worked with earlier in this chapter contains one key variable and one data variable. In this example, you need to associate more than one data value with each key.

In the following code, the DECLARE statement creates the `Airports` hash object and loads it from Sasuser.Acities. The DEFINEKEY method call defines the key to be the value of the variable `Code`. The DEFINEDATA method call defines the data to be the values of the variables `City` and `Name`.

```sas
data work.report;
  if 0 then
    set sasuser.acities (keep=Code City Name);
  if _N_=1 then do;
    declare hash airports (dataset: "sasuser.acities")
      airports.definekey ("Code");
    airports.definedata ("City", "Name");
    airports.definedone();
  end;
```
### Table 16.29 Hash Object Airports

<table>
<thead>
<tr>
<th>Key: Code</th>
<th>Data: City</th>
<th>Data: Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANC</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
</tr>
<tr>
<td>BNA</td>
<td>Nashville, TN</td>
<td>Nashville International Airport</td>
</tr>
<tr>
<td>CDG</td>
<td>Paris</td>
<td>Charles de Gaulle</td>
</tr>
<tr>
<td>LAX</td>
<td>Los Angeles, CA</td>
<td>Los Angeles International Airport</td>
</tr>
<tr>
<td>RDU</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
</tbody>
</table>

**Note:** To define all data set variables as data variables for the hash object, use the `ALL: "YES"` option.

```sas
HashObject.DEFINEDATA (ALL: "YES");
```

**Note:** The hash object can store multiple key variables as well as multiple data variables.

## Retrieving Multiple Data Values

You can use multiple `FIND` method calls to retrieve multiple data values. In the following program, the `FIND` method calls retrieve the value of `City` and `Name` from the Airports hash object based on the value of `Origin`.

```sas
data work.report;
  if _N_=1 then do;
    if 0 then set sasuser.acities (keep=Code City Name);
    declare hash airports (dataset: "sasuser.acities");
    airports.definekey ("Code");
    airports.definedata ("City", "Name");
    airports.definedone();
  end;
  set sasuser.revenue;
  airports.find (key: origin);
  OriginCity=city;
  OriginAirport=name;
  airports.find (key: dest);
  DestCity=city;
  DestAirport=name;
run;
```

**PROC PRINT** output shows the completed data set.

```sas
proc print data=work.report;
  var origin dest flightid date origincity originairport
testcity destairport;
run;
```
Using Return Codes with the FIND Method

Method calls create a return code that is a numeric value. The value specifies whether the method succeeded or failed. A value of 0 indicates that the method succeeded. A non-zero value indicates that the method failed.

If the program does not contain a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

To store the value of the return code in a variable, specify the variable name `rc` at the beginning of the method call. For example:

```plaintext
rc=hashobject.find (key:keyvalue);
```

The return code variable value can be used in conditional logic to ensure that the FIND method found a KEY value in the hash object that matches the KEY value from the PDV.

Example

Error messages are written to the log when the following program is submitted.

```plaintext
data work.report;
  if _N_=1 then do;
    if 0 then set sasuser.acities (keep=Code City Name);
    declare hash airports (dataset: "sasuser.acities");
    airports.definekey ("Code");
    airports.definedata ("City", "Name");
    airports.definedone();
  end;
set sasuser.revenue;
airports.find(key:origin);
OriginCity=city;
OriginAirport=name;
airports.find(key:dest);
DestCity=city;
DestAirport=name;
run;
```
A closer examination of the output shows that the data set Work.Reporting contains errors. For example, notice that in observations 6 through 8 the value of both OriginCity and DestCity is Canberra, Australian C and the values of OriginAirport and DestAirport are missing.

The errors occur because the Airports hash object does not include the key value WLG or a corresponding Name value for the key value CBR.

Conditional logic can be added to the program to create blank values if the values loaded from the input data set, Sasuser.Revenue, cannot be found in the Airports hash object:

- If the return code for the FIND method call has a value of 0, indicating that the method succeeded, the value of City and Name are assigned to the appropriate variables (OriginCity and OriginAirport or DestCity and DestAirport).
- If the return code for the FIND method call has a non-zero value, indicating the method failed, the value of City and Name are assigned blank values.

data work.report;
  if _N_=1 then do;
    if 0 then set sasuser.acities(keep=Code City Name);
    declare hash airports (dataset: "sasuser.acities");
    airports.definekey ("Code");
    airports.definedata ("City", "Name");
    airports.definedone();
  end;
  set sasuser.revenue;
  rc=airports.find(key:origin);
  if rc=0 then do;
    OriginCity=city;
    OriginAirport=name;
  end;
else do;

Using Lookup Tables to Match Data

Example 591

```plaintext
OriginCity='';
OriginAirport='';
end;
rc=airports.find(key:dest);
if rc=0 then do;
   DestCity=city;
   DestAirport=name;
end;
else do;
   DestCity='';
   DestAirport='';
end;
run;
```

PROC PRINT output shows the completed data set. Notice that in observations 6 through 8, the value of DestCity is now blank and no error messages appear in the log.

```plaintext
proc print data=work.report;
   var origin dest flightid date origincity originairport
destcity destairport;
run;
```

Figure 16.6 SAS Data Set Work.Report (first eight observations of selected variables)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Origin</th>
<th>Dest</th>
<th>FlightID</th>
<th>Date</th>
<th>OriginCity</th>
<th>OriginAirport</th>
<th>DestCity</th>
<th>DestAirport</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ANC</td>
<td>RDU</td>
<td>IAA03400</td>
<td>02DEC1999</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>2</td>
<td>ANC</td>
<td>RDU</td>
<td>IAA03400</td>
<td>14DEC1999</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>3</td>
<td>ANC</td>
<td>RDU</td>
<td>IAA03400</td>
<td>26DEC1999</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>4</td>
<td>ANC</td>
<td>RDU</td>
<td>IAA03401</td>
<td>06DEC1999</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>5</td>
<td>ANC</td>
<td>RDU</td>
<td>IAA03401</td>
<td>21DEC1999</td>
<td>Anchorage, AK</td>
<td>Anchorage International Airport</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>6</td>
<td>CBR</td>
<td>WLG</td>
<td>IA10500</td>
<td>04DEC1999</td>
<td>Canberra, Australian C</td>
<td>Canberra, Australian C</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>7</td>
<td>CBR</td>
<td>WLG</td>
<td>IA10500</td>
<td>16DEC1999</td>
<td>Canberra, Australian C</td>
<td>Canberra, Australian C</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
<tr>
<td>8</td>
<td>CBR</td>
<td>WLG</td>
<td>IA10500</td>
<td>28DEC1999</td>
<td>Canberra, Australian C</td>
<td>Canberra, Australian C</td>
<td>Raleigh-Durham, NC</td>
<td>Raleigh-Durham International Airport</td>
</tr>
</tbody>
</table>

Table 16.31 SAS Log

NOTE: There were 50 observations read from the data set SASUSER.ACITIES.
NOTE: There were 142 observations read from the data set SASUSER.REVENUE.
NOTE: The data set WORK.REPORT2 has 142 observations and 15 variables.
Summary

This section contains the following:

- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Introduction

Sometimes, you need to combine data from two or more sets into a single observation in a new data set according to the values of a common variable. When data sources do not have a common structure, you can use a lookup table to match them.

Using Multidimensional Arrays

When a lookup operation depends on more than one numerical factor, you can use a multidimensional array. You use an ARRAY statement to create an array. The ARRAY statement defines a set of elements that you plan to process as a group.

Using Stored Array Values

In some cases, you might need to store array values in a SAS data set rather than loading them in an ARRAY statement. Array values should be stored in a SAS data set when

- there are too many values to initialize easily in the array
- the values change frequently
- the same values are used in many programs.

The first step in using stored array values is to create an array to hold the values in the target data set. The next step is to load the array elements. One method for accomplishing this task is to load the array within a DO loop. The last step is to read the actual values stored in the other source data set.

Using PROC TRANSPOSE

The TRANSPOSE procedure can also be used to match data when the orientation of the data sets differs. PROC TRANSPOSE creates an output data set by restructuring the values in a SAS data set, thereby transposing selected variables into observations. The transposed (output) data set can then be merged with the other data set in order to match the data.

The output data set contains several default variables.

- _NAME_ is the default name of the variable that PROC TRANSPOSE creates to identify the source of the values in each observation in the output data set. This variable is a character variable whose values are the names of the variables that are transposed from the input data set. To override the default name, use the NAME= option.

- The remaining transposed variables are named col1...coln by default. To override the default names, use the PREFIX= option.
Merging the Transposed Data Set

You might need to use a BY statement with PROC TRANSPOSE in order to correctly structure the data for a merge. For each BY group, PROC TRANSPOSE creates one observation for each variable that it transposes. The BY variable itself is not transposed. In order to structure the data for a merge, you might also need to sort the output data set. Any other source data sets might need to be reorganized and sorted as well. When the data is structured correctly, the data sets can be merged.

Using Hash Objects as Lookup Tables

Beginning with SAS 9, the hash object is available for use in a DATA step. The hash object provides an efficient, convenient mechanism for quick data storage and retrieval.

A hash object resembles a table with rows and columns; it contains a key component and a data component. Unlike an array, which uses a series of consecutive integers to address array elements, a hash object can use any combination of numeric and character values as addresses.

The hash object is a DATA step component option. Component objects are data elements that consist of attributes and methods. To use a DATA step component object in your SAS program, you must first declare and create (instantiate) the object. After you define the hash object’s key and data variables, you can load the variables from hard-coded values or a SAS data set.

You use the FIND method call return code that is a numeric value. The value specifies whether the method succeeded or failed. A value of 0 indicates that the method succeeded. A nonzero value indicates that the method failed. The return code variable value can be used in conditional logic to ensure that the FIND method found KEY value in the hash object that matches the KEY value from the PDV.

Syntax

Using a Multidimensional Array

LIBNAME libref 'SAS-data-library';
DATA libref.sas-data-set(DROP=variable(s));
ARRAY array-name {rows,cols,...} <$><length>
   <array-elements> <(initial values)>;
SET <SAS-data-set(s) <(data-set-options(s))>> <options>;
variable=expression;
variable=array-name {rows,cols,...};
RUN;

Using Stored Array Values

LIBNAME libref 'SAS-data-library';
DATA libref.sas-data-set;
ARRAY array-name1 {rows,cols,...} <$><length>
   <array-elements> <(initial values)>;
IF expression THEN DO index-variable=specification;
   SET <SAS-data-set(s) <(data-set-options(s))>> <options>;
   ARRAY array-name2 {rows,cols,...} <$><length>
      <array-elements> <(initial values)>;
   END;
DO index-variable=specification;
    array-name1 {rows,cols,...}=array-name2 {rows,cols,...};
    END;
    END;
RUN;

Using PROC TRANSPOSE and a Merge
LIBNAME libref 'SAS-data-library';
PROC TRANSPOSE <DATA=input-data-set>
    <OUT=output-data-set>
    <NAME=name>
    <PREFIX=prefix>;
    BY <DESCENDING> variable-1
    <...<DESCENDING> variable-n> <NOTSORTED>;
    RUN;

PROC SORT;
    BY <DESCENDING> variable-1
    <...<DESCENDING> variable-n>;
RUN;

DATA libref.sas-data-set;
    SET <SAS-data-set(s) <(data-set-option(s)))> <options>;
    LENGTH variable(s) <$> length;
    variable=expression;
    variable=expression;
RUN;

PROC SORT;
    BY <DESCENDING> variable-1 <...<DESCENDING> variable-n>;
RUN;

DATA libref.sas-data-set;
    MERGE SAS-data-set-1 <(data-set-options)>
    SAS-data-set-2 <(data-set-options)>;
    BY <DESCENDING> variable-1 <...<DESCENDING> variable-n>;
RUN;

Using a Hash Object as a Lookup Table
DATA libref.sas-data-set;
    IF expression THEN statement;
    DECLARE object variable <(argument_tag-1: value-1 <, ...argument_tag-n: value-n>)>;
variable = _NEW_ object(<argument_tag-1: value-1, ...argument_tag-n: value-n>);
object.DEFINEKEY('keyvarname-1', 'keyvarname-n');
object.DEFINEDATA('datavarname-1', 'datavarname-n');
object.DEFINEONE();
CALL MISSING(varname1, varname2, ...);
object.ADD(<KEY: keyvalue-1, KEY: keyvalue-n, DATA: datavalue-1, ...
DATA: datavalue-n>);
END;
SET <SAS-data-set(s) <(data-set-options(s))>> <options>;
object.FIND(<KEY: keyvalue-1, KEY: keyvalue-n>);
RUN;

Sample Programs

Using a Multidimensional Array

data work.wndchill (drop = column row);
array WC{4,2} _temporary_
  (-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill = wc{row,column};
run;

Using Stored Array Values

data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.ctargets;
  array Mnth{*} Jan--Dec;
  do j=1 to dim(mnth);
    targets{year,j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarget dollar15.2;
run;

Using PROC TRANSPOSE and a Merge

proc transpose data=sasuser.ctargets
  out=work.ctarget2
  name=Month
  prefix=Ctarget;
  by year;
run;

proc sort data=work.ctarget2;
by year month;
run;

data work.mnthsum2;
  set sasuser.monthsum(keep=SaleMon RevCargo);
  length Month $ 8;
  Year=input(substr(SaleMon,4),4.)
  Month=substr(SaleMon,1,1)
    ||lowcase(substr(SaleMon,2,2));
run;

proc sort data=work.mnthsum2;
  by year month;
run;

data work.merged;
  merge work.mnthsum2 work.ctarget2;
  by year month;
run;

Using a Hash Object That Is Loaded From Hard-Coded Values

data work.difference (drop= goalamount);
  length goalamount 8;
  if _N_ = 1 then do;
    declare hash goal( );
    goal.definekey("QtrNum");
    goal.definedata("GoalAmount");
    goal.definedone();
    call missing(qtrnum, goalamount);
    goal.add(key:'qtr1', data:10 );
    goal.add(key:'qtr2', data:15 );
    goal.add(key:'qtr3', data: 5 );
    goal.add(key:'qtr4', data:15 );
  end;
  set sasuser.contrib;
  goal.find();
  Diff = amount - goalamount;
run;

Using a Hash Object That Is Loaded From a SAS Data Set

data work.report;
  if _N_=1 then do;
    if 0 then set sasuser.acities(keep=Code City Name);
    declare hash airports (dataset: "sasuser.acities");
    airports.definekey("Code");
    airports.definedata ("City", "Name");
    airports.definedone();
  end;
  set sasuser.revenue;
  rc=airports.find(key:origin);
  if rc=0 then do;
    OriginCity=city;
    OriginAirport=name;
  end;
end;
else do;
  OriginCity='';
  OriginAirport='';
end;
rc=airports.find(key:dest);
if rc=0 then do;
  DestCity=city;
  DestAirport=name;
end;
else do;
  DestCity='';
  DestAirport='';
end;
run;

Points to Remember

- The name of an array must be a SAS name that is not the name of a SAS variable in the same DATA step.
- Array elements must be either all numeric or all character.
- The initial values specified for an array can be numbers or character strings. You must enclose all character strings in quotation marks.
- The original SAS data set must be sorted or indexed prior to using a BY statement with PROC TRANSPOSE unless you use the NOTSORTED option.
- The hash object is a good choice for lookups using unordered data that can fit into memory because it provides in-memory storage and retrieval and does not require the data to be sorted.
- The hash object is sized dynamically, and exists for the duration of the DATA step.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which SAS statement correctly specifies the array `Sales` as illustrated in the following table?

   Table Representation of Sales Array
   
<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
</tr>
</thead>
<tbody>
<tr>
<td>m5</td>
<td>m6</td>
<td>m7</td>
<td>m8</td>
</tr>
<tr>
<td>m9</td>
<td>m10</td>
<td>m11</td>
<td>m12</td>
</tr>
</tbody>
</table>

   a. array Sales{3,4} m1-m12;
   b. array Sales{4,3} m1-m12;
2 Which of the following statements creates temporary array elements?

a array new (*) _temporary_

b array new (6) _temporary_

c array new (*) _temporary_ Jan Feb Mar Apr May Jun;

d array _temporary_ new (6) Jan Feb Mar Apr May Jun;

3 Which DO statement processes all of the elements in the \texttt{Yearx} array?

\begin{verbatim}
array Yearx{12} Jan--Dec;
do i=1 to dim(yearx);
do i=1 to 12;
do i=Jan to Dec;
da and b
\end{verbatim}

4 Given the following program, what is the value of \texttt{Points} in the fifth observation in the data set \texttt{Work.Results}?

\begin{verbatim}
data work.results;
    array score{2,4} _temporary_ (40,50,60,70,40,50,60,70);
    set work.contest;
    Points=score{week,finish};
run;
\end{verbatim}

\begin{verbatim}
Obs Name Week Finish
1 Tuflle 1 1
2 Gomez 1 2
3 Chapman 1 3
4 Venter 1 4
5 Van aux 2 1
6 Tittle 2 2
7 Venter 7 3
8 Gomez 7 4
\end{verbatim}

a 40
b 50
c 60
d 70

5 Array values should be stored in a SAS data set when

a there are too many values to initialize easily in an array.
b the values change frequently.
c the same values are used in many programs.
d all of the above

6 Given the following program, which statement is not true?

\begin{verbatim}
data work.lookup1;
    array Targets{1997:1999,12} _temporary_; 
    if _n_=1 then do i= 1 to 3;
        set sasuser.ctargets;
        array Mnth{*} Jan--Dec;
    \end{verbatim}
7 Given the following program, which variable names will appear in the data set *Work.New*?

```
proc transpose data=work.revenue;
  out=work.new;
run;
```

<table>
<thead>
<tr>
<th>SAS Data Set Work.Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

| a Year, Jan, Feb, Mar, Apr |
| b Year, 2000, 2001, 2002 |
| c _NAME_, Col1, Col2, Col3 |
| d _NAME_, Jan, Feb, Mar, Apr |

8 Which program creates the output data set *Work.Temp2*?

```
sas data set Work.Temp
```

<table>
<thead>
<tr>
<th>SAS Data Set Work.Temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAS Data Set Work.Temp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

| a proc transpose data=work.temp
  out=work.temp2
  prefix=Quarter;
  run;
| b proc transpose data=work.temp
  out=work.temp2
  name=Month |
9 Which version of the data set `Work.Sales2` is created by the following program?

```sas
proc transpose data=work.sales
  out=work.sales2
  name=Week;
by employee;
run;
```

**SAS Data Set Work.Sales**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee</th>
<th>Week1</th>
<th>Week2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Almers</td>
<td>3393.50</td>
<td>2192.25</td>
</tr>
<tr>
<td>2</td>
<td>Bonaventure</td>
<td>5093.75</td>
<td>2247.50</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>1813.30</td>
<td>2082.75</td>
</tr>
<tr>
<td>4</td>
<td>LaMance</td>
<td>1572.50</td>
<td>2960.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee</th>
<th>Week</th>
<th>COL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Almers</td>
<td>Week1</td>
<td>3393.50</td>
</tr>
<tr>
<td>2</td>
<td>Almers</td>
<td>Week2</td>
<td>2192.25</td>
</tr>
<tr>
<td>3</td>
<td>Bonaventure</td>
<td>Week1</td>
<td>5093.75</td>
</tr>
<tr>
<td>4</td>
<td>Bonaventure</td>
<td>Week2</td>
<td>2247.50</td>
</tr>
<tr>
<td>5</td>
<td>Johnson</td>
<td>Week1</td>
<td>1813.30</td>
</tr>
<tr>
<td>6</td>
<td>Johnson</td>
<td>Week2</td>
<td>2028.75</td>
</tr>
<tr>
<td>7</td>
<td>LaMance</td>
<td>Week1</td>
<td>1572.50</td>
</tr>
<tr>
<td>8</td>
<td>LaMance</td>
<td>Week2</td>
<td>2960.00</td>
</tr>
</tbody>
</table>
c

<table>
<thead>
<tr>
<th>Obs</th>
<th>Week</th>
<th>Almers</th>
<th>Bonnaventure</th>
<th>Johnson</th>
<th>LaMance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Week1</td>
<td>3393.50</td>
<td>5093.75</td>
<td>1813.30</td>
<td>1572.5</td>
</tr>
<tr>
<td>2</td>
<td>Week2</td>
<td>2192.25</td>
<td>2247.50</td>
<td>2028.75</td>
<td>2960.0</td>
</tr>
</tbody>
</table>

d

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee</th>
<th><em>NAHE</em></th>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Almers</td>
<td>Week1</td>
<td>3393.50</td>
</tr>
<tr>
<td>2</td>
<td>Bonnaventure</td>
<td>Week1</td>
<td>5093.75</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>Week1</td>
<td>1813.30</td>
</tr>
<tr>
<td>4</td>
<td>LaMance</td>
<td>Week1</td>
<td>1572.50</td>
</tr>
<tr>
<td>5</td>
<td>Almers</td>
<td>Week2</td>
<td>2192.25</td>
</tr>
<tr>
<td>6</td>
<td>Bonnaventure</td>
<td>Week2</td>
<td>2247.50</td>
</tr>
<tr>
<td>7</td>
<td>Johnson</td>
<td>Week2</td>
<td>2028.75</td>
</tr>
<tr>
<td>8</td>
<td>LaMance</td>
<td>Week2</td>
<td>2960.00</td>
</tr>
</tbody>
</table>

10 Which program creates the data set `Work.Fishsize`?
SAS Data Set Work.Fishdata

<table>
<thead>
<tr>
<th>Obs</th>
<th>Location</th>
<th>Date</th>
<th>Length1</th>
<th>Weight1</th>
<th>Length2</th>
<th>Weight2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>31</td>
<td>0.25</td>
<td>32</td>
<td>0.30</td>
</tr>
<tr>
<td>2</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>29</td>
<td>0.23</td>
<td>30</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>32</td>
<td>0.35</td>
<td>32</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>33</td>
<td>0.30</td>
<td>33</td>
<td>0.28</td>
</tr>
</tbody>
</table>

SAS Data Set Work.Fishsize

<table>
<thead>
<tr>
<th>Obs</th>
<th>Location</th>
<th>Date</th>
<th><em>NAME</em></th>
<th>Measurement1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Length1</td>
<td>31.00</td>
</tr>
<tr>
<td>2</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Weight1</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Length2</td>
<td>32.00</td>
</tr>
<tr>
<td>4</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Weight2</td>
<td>0.30</td>
</tr>
<tr>
<td>5</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Length1</td>
<td>29.00</td>
</tr>
<tr>
<td>6</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Weight1</td>
<td>0.23</td>
</tr>
<tr>
<td>7</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Length2</td>
<td>30.00</td>
</tr>
<tr>
<td>8</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Weight2</td>
<td>0.25</td>
</tr>
<tr>
<td>9</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Length1</td>
<td>32.00</td>
</tr>
<tr>
<td>10</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Weight1</td>
<td>0.35</td>
</tr>
<tr>
<td>11</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Length2</td>
<td>32.00</td>
</tr>
<tr>
<td>12</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Weight2</td>
<td>0.25</td>
</tr>
<tr>
<td>13</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Length1</td>
<td>33.00</td>
</tr>
<tr>
<td>14</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Weight1</td>
<td>0.30</td>
</tr>
<tr>
<td>15</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Length2</td>
<td>33.00</td>
</tr>
<tr>
<td>16</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Weight2</td>
<td>0.28</td>
</tr>
</tbody>
</table>

a) proc transpose data=work.fishdata
daout=work.fishsize
b) prefix=Measurement;
c) run;
d) by location;
e) run;
f) by date;
g) run;
h) by location date;
i) run;
CHAPTER 17
Formatting Data

Overview 604
  Introduction 604
  Objectives 604
Creating Custom Formats Using the VALUE Statement 605
  Review of Creating Non-Overlapping Formats 605
  Creating a Format with Overlapping Ranges 606
  Example 606
Creating Custom Formats Using the PICTURE Statement 608
  Ways to Specify Pictures 608
  Example 609
  Guidelines for Specifying Directives 610
  Example 611
Managing Custom Formats 612
  Using FMTLIB with PROC FORMAT to Document Formats 612
  Example 612
  Using PROC CATALOG to Manage Formats 613
  Example 614
Using Custom Formats 615
  Example 615
  Using a Permanent Storage Location for Formats 616
  Example 616
  Substituting Formats to Avoid Errors 617
  Example 617
Creating Formats from SAS Data Sets 618
  Example 619
  Rules for Control Data Sets 619
  Example 619
    Step 1: List Data Set Variables 620
    Step 2: Restructure the Data 620
    Apply the Format 622
Creating SAS Data Sets from Custom Formats 623
  Example 623
    Step 1: Create a SAS Data Set from the Format 623
    Step 2: Edit the Data Set 624
    Step 3: Create a Format from the SAS Data Set 624
Summary 626
  Text Summary 626
  Creating Custom Formats Using the VALUE Statement 626
  Creating Formats with Overlapping Ranges 626
  Creating Custom Formats Using the PICTURE Statement 626
  Documenting Formats 626
Overview

Introduction

Custom formats are used to display variable values in certain ways, such as formatting a product number so that it is displayed as descriptive text. You should already be familiar with using the FORMAT procedure to create and store formats.

<table>
<thead>
<tr>
<th>SAS Variable Value</th>
<th>Your Format</th>
<th>SAS Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>35mm camera</td>
<td>$itemfmt</td>
<td>35mm camera</td>
</tr>
</tbody>
</table>

In this chapter you learn to document formats, use formats located in any catalog, create formats with overlapping ranges, and use the PICTURE statement to create a format for printing numbers. You also learn an easy way to update formats by creating a SAS data set from a format, updating the data set, and then creating a format from the SAS data set.

Objectives

In this chapter, you learn to
- create formats with overlapping ranges
- create custom formats using the PICTURE statement
- use the FMTLIB keyword with the FORMAT procedure to document formats
- use the CATALOG procedure to manage format entries
- use the DATASETS procedure to associate a format with a variable
- use formats located in any catalog
- substitute formats to avoid errors
- create a format from a SAS data set
- create a SAS data set from a custom format.
Creating Custom Formats Using the VALUE Statement

Review of Creating Non-Overlapping Formats

As you learned in Creating and Applying User-Defined Formats, you can use the VALUE statement in the FORMAT procedure to create a custom format for displaying data in a particular way. For example, suppose you have airline data and you want to create several custom formats that you can use for your report-writing tasks. You need formats that

- group airline routes into zones
- label airport codes as International or Domestic
- group cargo revenue figures into ranges.

The following PROC FORMAT step creates these three formats:

```sas
libname library 'c:\sas\newfmts';
proc format lib=library;
  value $routes
    'Route1' = 'Zone 1'
    'Route2' - 'Route4' = 'Zone 2'
    'Route5' - 'Route7' = 'Zone 3'
    ' ' = 'Missing'
    other = 'Unknown';
  value $dest
    'AKL','AMS','ARN','ATH','BKK','BRU',
    'CBR','CCU','CDG','CPH','CPT','DEL',
    'DXB','FBU','FCO','FRA','GLA','GVA',
    'HEL','HKG','HND','JED','JNB','JRS',
    'LHR','LIS','MAD','NBO','PEK','PRG',
    'SIN','SYD','VIE','WLG' = 'International'
    'ANC','BHM','BNA','BOS','DFW','HNL',
    'IAD','IND','JFK','LAX','MCI','MIA',
    'MSY','ORD','PWM','RDU','SEA','SFO' = 'Domestic';
  value revfmt
    . = 'Missing'
    low - 10000 = 'Up to $10,000'
    10000 <= 20000 = '$10,000+ to $20,000'
    20000 <= 30000 = '$20,000+ to $30,000'
    30000 <= 40000 = '$30,000+ to $40,000'
    40000 <= 50000 = '$40,000+ to $50,000'
    50000 <= 60000 = '$50,000+ to $60,000'
    60000 <= HIGH = 'More than $60,000';
run;
```

Tip: If you choose to run this example code, be sure to change the path in the LIBNAME statement to a storage location in your operating environment.

The PROC FORMAT step creates three formats: $ROUTES, and $DEST., which are character formats, and REVFMT., which is numeric. $ROUTES, groups airline routes into zones. In $ROUTES, both single values and ranges are assigned labels.

missing values are designated by a space in quotation marks and are assigned the label “Missing”
the keyword *OTHER* is used to assign the label “Unknown” to any values that are not addressed in the range.

$DEST.$ labels airport codes as either International or Domestic. In $DEST.$,
- unique character values are enclosed in quotation marks and separated by commas
- missing values and values not included in the range are not handled in this format.

REVFMT. groups cargo revenue figures into ranges. In REVFMT.,
- the “less than” operator (<) is used to show a non-inclusive range (10000<20000 indicates that the first value is not included in the range)
- the keyword *LOW* is used to specify the lower limit of a variable’s value range, but it does not include missing values
- missing values are designated with a period (.) and assigned the label “Missing”
- the keyword *HIGH* is used to specify the upper limit of a variable’s value range.

### Creating a Format with Overlapping Ranges

There are times when you need to create a format that groups the same values into different ranges. To create overlapping ranges, use the MULTILABEL option in the VALUE statement in PROC FORMAT.

<table>
<thead>
<tr>
<th>General form, VALUE statement with the MULTILABEL option:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VALUE</strong> format-name <em>(MULTILABEL);</em>*</td>
</tr>
<tr>
<td>where</td>
</tr>
<tr>
<td><code>format-name</code></td>
</tr>
<tr>
<td>is the name of the character or numeric format that is being created.</td>
</tr>
</tbody>
</table>

### Example

Suppose you want to create a format that groups dates into overlapping categories. In the table below, notice that each month appears in two groups.

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan - Mar</td>
<td>1st Quarter</td>
</tr>
<tr>
<td>Apr - Jun</td>
<td>2nd Quarter</td>
</tr>
<tr>
<td>Jul - Sep</td>
<td>3rd Quarter</td>
</tr>
<tr>
<td>Oct - Dec</td>
<td>4th Quarter</td>
</tr>
<tr>
<td>Jan - Jun</td>
<td>First Half of Year</td>
</tr>
<tr>
<td>Jul - Dec</td>
<td>Second Half of Year</td>
</tr>
</tbody>
</table>

In the PROC FORMAT step below, the MULTILABEL option has been added to indicate that the DATES. format will have values with overlapping ranges:

```plaintext
proc format;
    value dates (multilabel)
```
'01jan2000'd - '31mar2000'd = '1st Quarter'
'01apr2000'd - '30jun2000'd = '2nd Quarter'
'01jul2000'd - '30sep2000'd = '3rd Quarter'
'01oct2000'd - '31dec2000'd = '4th Quarter'
'01jan2000'd - '30jun2000'd = 'First Half of Year'
'01jul2000'd - '31dec2000'd = 'Second Half of Year';

run;

Multilabel formatting allows an observation to be included in multiple rows or categories. To use the multilabel formats, you can specify the MLF option on class variables in procedures that support it:

- PROC TABULATE
- PROC MEANS
- PROC SUMMARY.

The MLF option activates multilabel format processing when a multilabel format is assigned to a class variable. For example, in the following TABULATE procedure,

- the FORMAT= option specifies DOLLAR15.2 as the default format for the value in each table cell
- the FORMAT statement references the new format DATES. for the variable Date
- the CLASS statement identifies Date as the class variable and uses the MLF option to activate multilabel format processing
- the row dimension of the TABLE statement creates a row for each formatted value of Date.

```
proc tabulate data = sasuser.sale2000 format = dollar15.2;
format Date dates.;
class Date / mlf;
var RevCargo;
  table Date, RevCargo*(mean median);
run;
```

<table>
<thead>
<tr>
<th>Date</th>
<th>RevCargo</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Quarter</td>
<td></td>
<td>$24,839.08</td>
<td>$4,939.00</td>
</tr>
<tr>
<td>2nd Quarter</td>
<td></td>
<td>$14,949.77</td>
<td>$4,579.50</td>
</tr>
<tr>
<td>3rd Quarter</td>
<td></td>
<td>$19,836.00</td>
<td>$3,591.00</td>
</tr>
<tr>
<td>4th Quarter</td>
<td></td>
<td>$20,403.13</td>
<td>$1,940.00</td>
</tr>
<tr>
<td>First Half of Year</td>
<td></td>
<td>$19,894.42</td>
<td>$4,823.00</td>
</tr>
<tr>
<td>Second Half of Year</td>
<td></td>
<td>$20,261.35</td>
<td>$2,100.00</td>
</tr>
</tbody>
</table>

*Tip:* For more information about using the MULTILABEL option, see the SAS documentation for the FORMAT procedure.
Creating Custom Formats Using the PICTURE Statement

You have learned that you can use the VALUE statement to convert output values to a different form. Suppose one of the variables in your data set had numeric values that you wanted to format a certain way. For example, you might have a phone number listed in your data set as 111231234 and you want to format it as (111) 123-1234. You can use the PICTURE statement to create a template for printing numbers.

General form, PROC FORMAT with the PICTURE statement:

```plaintext
PROC FORMAT;
  PICTURE format-name
      value-or-range='picture';
RUN;
```

where

- `format-name` is the name of the format you are creating.
- `value-or-range` is the individual value or range of values you want to label.
- `picture` specifies a template for formatting values of numeric variables. The template is a sequence of characters enclosed in quotation marks. The maximum length for a picture is 40 characters.

### Ways to Specify Pictures

Pictures are specified with three types of characters:

- **digit selectors**
- **message characters**
- **directives**.

Let’s look at using digit selectors and message characters first. You’ll learn about directives a little later.

**Digit selectors** are numeric characters (0 through 9) that define positions for numeric values. If you use nonzero digit selectors, zeros are added to the formatted value as needed. If you use zeros as digit selectors, no zeros are added to the formatted value.

In the picture definitions below you can see the difference between using nonzero digit selectors (99) and zero digit selectors (00) on the formatted values.

<table>
<thead>
<tr>
<th>Picture Definition</th>
<th>Data Values</th>
<th>Formatted Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>picture month 1-12='99'</code></td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td><code>picture month 1-12='00'</code></td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
**Message characters** are nonnumeric characters that print as specified in the picture. They are inserted into the picture after the numeric digits are formatted. Digit selectors must come before message characters in the picture definition. In the picture definition below, the text string JAN is made up of message characters.

<table>
<thead>
<tr>
<th>Picture Definition</th>
<th>Data Value</th>
<th>Formatted Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>picture month 1='99 JAN';</td>
<td>1</td>
<td>01 JAN</td>
</tr>
</tbody>
</table>

**Example**

The following PICTURE statement contains both digit selectors and message characters. Because the RAINAMT. format has nonzero digit selectors, values are printed with leading zeros. The keyword OTHER is used to print values and message characters for any values that do not fall into the specified range.

```
proc format;
    picture rainamt
      0-2='9.99 slight'
      2<-4='9.99 moderate'
      4<-<10='9.99 heavy'
      other='999 check value';
run;
data rain;
    input Amount;
datalines;
   4
   3.9
  20
  .5
   6;
run;
proc print data=rain;
    format amount rainamt.;
run;
```

The following output shows the values with the RAINAMT. format applied.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.00 moderate</td>
</tr>
<tr>
<td>2</td>
<td>3.90 moderate</td>
</tr>
<tr>
<td>3</td>
<td>020 check value</td>
</tr>
<tr>
<td>4</td>
<td>0.50 slight</td>
</tr>
<tr>
<td>5</td>
<td>6.00 heavy</td>
</tr>
</tbody>
</table>

Next you learn about using *directives* to specify picture formats.
The final way to specify a picture is with a directive. *Directives* are special characters that you can use in the picture to format *date*, *time*, or *datetime* values. If you use a directive, you must specify the *DATATYPE= option* in the PICTURE statement. This option specifies that the picture applies to a SAS date, SAS time, or SAS datetime value.

General form, PICTURE statement with the DATATYPE= option:

```
PICTURE format-name
value-or-range, 'picture' (DATATYPE=SAS-date-value-type);
```

where

- *format-name* is the name of the format you are creating.
- *value-or-range* is the individual value or range of values you want to label.
- *picture* specifies a template with directives for formatting values of numeric variables.
- *SAS-date-value-type* is either *DATE*, *TIME*, or *DATETIME*.

---

**Guidelines for Specifying Directives**

The percent sign (%) followed by a letter indicates a directive. Directives that you can use to create a picture format are listed in the table below.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>abbreviated weekday name</td>
</tr>
<tr>
<td>%A</td>
<td>full weekday name</td>
</tr>
<tr>
<td>%b</td>
<td>abbreviated month name</td>
</tr>
<tr>
<td>%B</td>
<td>full month name</td>
</tr>
<tr>
<td>%d</td>
<td>day of the month as a number 1-31, with no leading zero</td>
</tr>
<tr>
<td>%H</td>
<td>24-hour clock as a number 0-23, with no leading zero</td>
</tr>
<tr>
<td>%I</td>
<td>12-hour clock as a number 1-12, with no leading zero</td>
</tr>
<tr>
<td>%j</td>
<td>day of the year as a number 1-366, with no leading zero</td>
</tr>
<tr>
<td>%m</td>
<td>month as a number 1-12, with no leading zero</td>
</tr>
<tr>
<td>%M</td>
<td>minute as a decimal number 0-59, with no leading zero</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM</td>
</tr>
<tr>
<td>%S</td>
<td>second as a number 0-59, with no leading zero</td>
</tr>
<tr>
<td>%U</td>
<td>week number of the year (Sunday is the first day of the week) as a number 0-53, with no leading zero</td>
</tr>
<tr>
<td>%w</td>
<td>weekday as a number (1=Sunday, to 7)</td>
</tr>
</tbody>
</table>
Although directives generally return numeric values with no leading zeros, you can add 0 in the directive so that if a one-digit numeric value is returned, it is preceded by a 0.

As shown below, when you create a picture with directives, the number of characters inside quotation marks is the maximum length of the formatted value. You must add trailing blanks to the directive if your values will contain more characters than the picture. The formatted value will be truncated if you do not.

<table>
<thead>
<tr>
<th>Formatted Result</th>
<th>Picture Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d %m %y</td>
<td>'%0d-%m%b%Y'</td>
</tr>
<tr>
<td>%Y</td>
<td>'%d %b %Y'</td>
</tr>
</tbody>
</table>

Example

Suppose you want to display values for employee hire dates in the format *dd-mmm-yyyy* (such as *25-JAN2000*). This format requires spaces for 10 characters.

The following code creates this format. There are a few things you should notice about the picture definition:

- The keywords LOW and HIGH are used to include all values.
- The 0 in the %d directive indicates that if the day of the month is one digit, it should be preceded by a 0.
- Because there are only eight characters inside the single quotation marks, you must add two blank spaces to set the length to 10.

```sas
proc format;
  picture mydate
    low-high='%0d-%b%Y' (datatype=date);
run;
proc print data=sasuser.empdata
  (keep=division hireDate lastName obs=5);
  format hiredate mydate.;
run;
```

The output below shows the values for *HireDate* formatted with the MYDATE. picture format.
Using FMTLIB with PROC FORMAT to Document Formats

When you have created a large number of permanent formats, it can be easy to forget the exact spelling of a specific format name or its range of values. Remember that adding the keyword FMTLIB to the PROC FORMAT statement displays a list of all the formats in the specified catalog, along with descriptions of their values.

```
libname library 'c:sas\newfmt';
proc format lib=library fmtlib;
run;
```

You can also use the SELECT and EXCLUDE statements to process specific formats rather than an entire catalog.

General form, PROC FORMAT with FMTLIB and the SELECT and EXCLUDE statements:

```
PROC FORMAT LIB=library FMTLIB;
   SELECT format-name;
   EXCLUDE format-name;
RUN;
```

where

- `library` is the name of the library where the formats are stored. If you do not specify the LIB= option, formats in the Work library are listed.
- `format-name` is the name of the format you want to select or exclude.

Example

The following code displays only the documentation for the $ROUTES. format. Notice that you do not use a period at the end of the format name when you specify the format in the SELECT statement.
libname library 'c:sas\newfmt';
proc format lib=library fmtlib;
    select $routes;
run;

Table 17.1 SAS Output

<table>
<thead>
<tr>
<th>FORMAT NAME : $ROUTES LENGTH: 7 NUMBER OF VALUES: 5 MIN LENGTH: 1 MAX LENGTH: 40 DEFAULT LENGTH: 7 FUZZ: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Missing</td>
</tr>
<tr>
<td>Route1</td>
</tr>
<tr>
<td>Route2</td>
</tr>
<tr>
<td>Route5</td>
</tr>
<tr>
<td><strong>OTHER</strong></td>
</tr>
</tbody>
</table>

Tip: If you specify more than one format on the SELECT or EXCLUDE statement, separate each format name with a space as follows:

    select $routes newdate;

Using PROC CATALOG to Manage Formats

Because formats are saved as catalog entries, you can use the CATALOG procedure to manage your formats. Using PROC CATALOG, you can

- create a listing of the contents of a catalog
- copy a catalog or selected entries within a catalog
- delete or rename entries within a catalog.
General form, PROC CATALOG step:

```
PROC CATALOG CATALOG=libref.catalog;
   CONTENTS <OUT=SAS-data-set>;
   COPY OUT=libref.catalog <options>;
   SELECT entry-name.entry-type(s);
   EXCLUDE entry-name.entry-type(s);
   DELETE entry-name.entry-type(s);
RUN;
QUIT;
```

where

- `libref.catalog` with the CATALOG= argument is the SAS catalog to be processed.
- `SAS-data-set` is the name of the data set that will contain the list of the catalog contents.
- `libref.catalog` with the OUT= argument is the SAS catalog to which the catalog entries will be copied.
- `entry-name.entry-type(s)` are the full names of catalog entries (in the form name.type) that you want to process.

**CAUTION:**

If you are using the SAS Learning Edition, you will not be able to submit a PROC CATALOG step because the CATALOG procedure is not included in the software.

---

**Example**

The first PROC CATALOG step below copies the $ROUTES. format from the `Library.Formats` catalog to the `Work.Formats` catalog. Notice that in the SELECT statement, you specify the $ROUTES. character format using the full catalog entry name, `ROUTES.FORMATC`.

```
proc catalog catalog=library.formats;
   copy out=work.formats;
   select routes.formatc;
run;
proc catalog cat=work.formats;
   contents;
run;
quit;
```

The second PROC CATALOG step displays the contents of the `Work.Formats` catalog. A note is written to the log when the format is copied from one catalog to another.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Create Date</th>
<th>Modified Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ROUTES FORMATC</td>
<td>17OCT2002:10:30:24</td>
<td>17OCT2002:10:30:24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tip: For more information about PROC CATALOG, including other statements and options that you can use, see the SAS documentation.

Using Custom Formats

After you have created a custom format, you can use SAS statements to permanently assign the format to a variable in a DATA step, or you can temporarily specify a format in a PROC step to determine the way that the data values appear in output. You should already be familiar with referencing a format in a FORMAT statement.

Another way to assign, change, or remove the format associated with a variable in an existing SAS data set is to use the DATASETS procedure to modify the descriptor portion of a data set.

General form, DATASETS procedure with the MODIFY and FORMAT statements:

```
PROC DATASETS LIB=SAS-library <NOLIST>;
    MODIFY SAS-data-set;
    FORMAT variable(s) format;
QUIT;
```

where

- **SAS-library** is the name of the SAS library that contains the data you want to modify.
- **NOLIST** suppresses the directory listing.
- **SAS-data-set** is the name of the SAS data set you want to modify.
- **variable** is the name of one or more variables whose format you want to assign, change, or remove.
- **format** is the name of a format to apply to the variable or variables listed before it. If you do not specify a format, any format associated with the variable is removed.

Note: The DATASETS procedure is interactive and will run until you issue the QUIT statement.

Example

In the following code, two variables in the SAS data set Flights are changed. The format $DEST. is associated with the variable Dest and the format is removed from the variable Baggage.

```
proc datasets lib=Mylib;
    modify flights;
    format dest $dest.;
    format baggage;
quit;
```
Using a Permanent Storage Location for Formats

When you permanently associate a format with a variable in a data set, it is important to be sure that the format you are referencing is stored in a permanent location. Remember that the storage location for the format is determined when the format is created in the FORMAT procedure.

When you create formats that you want to use in subsequent SAS sessions, it is useful to

1. assign the Library libref to a SAS library in the SAS session in which you are running the PROC FORMAT step
2. specify LIB=LIBRARY in the PROC FORMAT step that creates the format
3. include a LIBNAME statement in the program that references the format to assign the Library libref to the library that contains the permanent format catalog.

You can store formats in any catalog you choose; however, you must identify the format catalogs to SAS before you can access them. You learn about this in the next topic.

When a format is referenced, SAS automatically looks through the following libraries in this order:

- Work.Formats
- Library.Formats.

The Library libref is recommended for formats because it is automatically searched when a format is referenced. If you store formats in libraries or catalogs other than those in the default search path, you must use the FMTSEARCH= system option to tell SAS where to find your formats.

General form, FMTSEARCH= system option:

```
OPTIONS FMTSEARCH= (catalog-1 catalog-2...catalog-n);
```

where

catalog

is the name of one or more catalogs to search. The value of catalog can be either libref or libref.catalog. If only the libref is given, SAS assumes that Formats is the catalog name.

The Work.Formats catalog is always searched first, and the Library.Formats catalog is searched next, unless one or both catalogs appear in the FMTSEARCH= list.

Example

Suppose you have formats stored in the Rpt library and in the Prod.Newfmt catalog. The following OPTIONS statement tells SAS where to find your formats:

```
options fmtsearch=(rpt prod.newfmt);
```

Because no catalog is specified with the Rpt libref, the default catalog name Formats is assumed. This OPTIONS statement creates the following search order:
Because the Work and Library librefs were not specified in the FMTSEARCH= option, they are searched in default order.

Substituting Formats to Avoid Errors

Let's look at what happens if you forget to specify a catalog in the FMTSEARCH= option, misspell a format name, or make some other mistake that causes SAS to fail to locate the format you have specified.

By default, the FMTERR system option is in effect. If you use a format that SAS cannot load, SAS issues an error message and stops processing the step. To prevent this, you must change the system option FMTERR to NOFMTERR. When NOFMTERR is in effect, SAS substitutes a format for the missing format and continues processing.

General form, FMTERR system option:

```
OPTIONS FMTERR | NOFMTERR;
```

where

- **FMTERR** specifies that when SAS cannot find a specified variable format, it generates an error message and stops processing. Substitution does not occur.
- **NOFMTERR** replaces missing formats with the $w. or $w. default format and continues processing.

Example

Suppose the FMTERR system option is in effect. In a previous example, we created the $ROUTES. format to group airline routes into zones. In the following code, the $ROUTES. format is misspelled:

```
proc print data=sasuser.cargorev(obs=10);
  format route $route.;
run;
```

Because FMTERR is in effect, the format cannot be located and SAS stops processing. An error message is written to the log.
Creating Formats from SAS Data Sets

You have seen that you can create a format by specifying values and labels in a PROC FORMAT step. You can also create a format from a SAS data set that contains value information (called a control data set). To do this, you use the CNTLIN= option to read the data and create the format.
General form, CNTLIN= option in PROC FORMAT:

```
PROC FORMAT LIBRARY=libref.catalog
    CNTLIN=SAS-data-set;
```

where

- `libref.catalog` is the name of the catalog in which you want to store the format.
- `SAS-data-set` is the name of the SAS data set that you want to use to create the format.

Example

Suppose you have a SAS data set named `Routes` that has the variables required to create a format. You specify the data set in the CNTLIN= option as follows:

```
proc format lib=library.formats cntlin=sasuser.routes;
run;
```

As you can see, the code for creating a format from a SAS data set is simple. However, the control data set must contain certain variables before it can be used to create a format, and most data sets must be restructured before they can be used.

Rules for Control Data Sets

When you create a format using programming statements, you specify the name of the format, the range or value, and the label for each range or value as in the VALUE statement below:

```
value rainfall 0='none';
```

The control data set you use to create a format must contain variables that supply this same information. That is, the data set specified in the CNTLIN= option

- must contain the variables `FmtName`, `Start`, and `Label`, which contain the format name, value or beginning value in the range, and label.
- must contain the variable `End` if a range is specified. If there is no `End` variable, SAS assumes that the ending value of the format range is equal to the value of `Start`.
- must contain the variable `Type` for character formats, unless the value for `FmtName` begins with a $.
- must be sorted by `FmtName` if multiple formats are specified.

Now let’s see how you create a correctly structured data set.

Example

Suppose you want to create a format that labels a three-letter airport code with the name of the city where the airport is located. You have a data set, `Sasuser.Acities`, that contains airport codes and airport cities. However, the data does not have the required variables for the CNTLIN= option.
Table 17.3  SAS Data Set Sasuser.Acities (Partial Listing)

<table>
<thead>
<tr>
<th>City Where Airport is Located</th>
<th>Start Point</th>
<th>Airport Name</th>
<th>Country Where Airport is Located</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auckland</td>
<td>AKL</td>
<td>International</td>
<td>New Zealand</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>AMS</td>
<td>Schiphol</td>
<td>Netherlands</td>
</tr>
<tr>
<td>Anchorage, AK</td>
<td>ANC</td>
<td>Anchorage International Airport</td>
<td>USA</td>
</tr>
<tr>
<td>Stockholm</td>
<td>ARN</td>
<td>Arlanda</td>
<td>Sweden</td>
</tr>
<tr>
<td>Athens (Athinai)</td>
<td>ATH</td>
<td>Hellinikon International Airport</td>
<td>Greece</td>
</tr>
<tr>
<td>Birmingham, AL</td>
<td>BHM</td>
<td>Birmingham International Airport</td>
<td>USA</td>
</tr>
<tr>
<td>Bangkok</td>
<td>BKK</td>
<td>Don Muang International Airport</td>
<td>Thailand</td>
</tr>
</tbody>
</table>

To create a format from this data set, you need to
1. List data set variables
2. Restructure the data

Step 1: List Data Set Variables

Remember that you need to have the variables **FmtName**, **Start**, and **Label**. You can submit a PROC CONTENTS step to get a listing of the variables in the *Sasuser.Acities* data set.

**Partial Output**

```plaintext
proc contents data=sasuser.acities;
run;
```

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>City</td>
<td>Char</td>
<td>22</td>
<td>City Where Airport is Located</td>
</tr>
<tr>
<td>2</td>
<td>Code</td>
<td>Char</td>
<td>3</td>
<td>Start Point</td>
</tr>
<tr>
<td>4</td>
<td>Country</td>
<td>Char</td>
<td>40</td>
<td>Country Where Airport is Located</td>
</tr>
<tr>
<td>3</td>
<td>Name</td>
<td>Char</td>
<td>50</td>
<td>Airport Name</td>
</tr>
</tbody>
</table>

**Tip:** You can also get a list of variable names by using PROC DATASETS with a CONTENTS statement or by viewing the properties of the SAS data set in the SAS Explorer window.

**Tip:** You can also get a list of variable names by using PROC DATASETS with a CONTENTS statement or by viewing the properties of the SAS data set in the Properties window.

Step 2: Restructure the Data

Once you have looked at the data and know the variable names, you are ready to write a DATA step to manipulate the data. The variable **Code** is the three-letter airport code and the variable **City** is the city where the airport is located. You can rename the variable **Code** to **Start** and the variable **City** to **Label**, but you also need to create the variable **FmtName**.

The code below is an efficient way to get your data ready to use. The DATA step uses

1. the KEEP statement to write only the specified variables to the output data set
the RETAIN statement to create the variable `FmtName` and set the value to ‘$airport’

- the RENAME statement to rename the variable `Code` to `Start` (you do not need a variable named `End` because you are labeling discrete values rather than ranges) and to rename the variable `City` to `Label`.

data sasuser.aports;
  keep Start Label FmtName;
  retain FmtName ‘$airport’;
  set sasuser.acities (rename=(Code=Start
    City= Label));
run;

proc print data=sasuser.aports(obs=10) noobs;
run;

Below is the listing of the first ten observations in the new data set `Sasuser.Aports`.

<table>
<thead>
<tr>
<th>FmtName</th>
<th>Label</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>$airport</td>
<td>Auckland</td>
<td>AKL</td>
</tr>
<tr>
<td>$airport</td>
<td>Amsterdam</td>
<td>AMS</td>
</tr>
<tr>
<td>$airport</td>
<td>Anchorage, AK</td>
<td>ANC</td>
</tr>
<tr>
<td>$airport</td>
<td>Stockholm</td>
<td>ARN</td>
</tr>
<tr>
<td>$airport</td>
<td>Athens (Athina)</td>
<td>ATH</td>
</tr>
<tr>
<td>$airport</td>
<td>Birmingham, AL</td>
<td>BHM</td>
</tr>
<tr>
<td>$airport</td>
<td>Bangkok</td>
<td>BKK</td>
</tr>
<tr>
<td>$airport</td>
<td>Nashville, TN</td>
<td>BNA</td>
</tr>
<tr>
<td>$airport</td>
<td>Boston, MA</td>
<td>BOS</td>
</tr>
<tr>
<td>$airport</td>
<td>Brussels (Bruxelles)</td>
<td>BRU</td>
</tr>
</tbody>
</table>

This data set is now in the proper format to be used to create a format using the CNTLIN= option. Next, let's look again at the code that creates the format from this data set.

Once you have the data in the proper format, you can use the CNTLIN= option to create the format. The first PROC FORMAT step creates a format from the data set `Sasuser.Aports`. The second PROC FORMAT step documents the new format.

```
proc format library=sasuser cntlin=sasuser.aports;
run;

proc format library=sasuser fmtlib;
  select $airport;
run;
```

The first few lines of the output are shown below.
Table 17.4  Partial SAS Output

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| START | END | LABEL (VER. V7/V8 21OCT2002:14:13:14) |
| AKL | AKL | Auckland |
| AMS | AMS | Amsterdam |
| ANC | ANC | Anchorage, AK |
| ARN | ARN | Stockholm |
| ATH | ATH | Athens (Athinai) |
| BHM | BHM | Birmingham, AL |
| BKK | BKK | Bangkok |

Apply the Format

Let’s take a look at this format applied to the data set Sasuser.Cargo99. The following PROC PRINT code assigns the $AIRPORT. format to both the Dest and Origin variables:

```bash
proc print data=sasuser.cargo99 (obs=5);
  var origin dest cargorev;
  format origin dest $airport.;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Origin</th>
<th>Dest</th>
<th>CargoRev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Raleigh-Durham, NC</td>
<td>London, England</td>
<td>$111,720.00</td>
</tr>
<tr>
<td>2</td>
<td>Raleigh-Durham, NC</td>
<td>London, England</td>
<td>$109,270.00</td>
</tr>
<tr>
<td>3</td>
<td>Raleigh-Durham, NC</td>
<td>London, England</td>
<td>$109,270.00</td>
</tr>
<tr>
<td>4</td>
<td>Raleigh-Durham, NC</td>
<td>London, England</td>
<td>$118,180.00</td>
</tr>
<tr>
<td>5</td>
<td>Raleigh-Durham, NC</td>
<td>London, England</td>
<td>$108,290.00</td>
</tr>
</tbody>
</table>

Tip: For more information about using the CNTLIN= option, see the SAS documentation for the FORMAT procedure.

Creating SAS Data Sets from Custom Formats

You know how to create a format from a SAS data set, but what if you want to create a SAS data set from a format? To do this, you use the CNTLOUT= option.
General form, CNTLOUT= option in PROC FORMAT:

```
PROC FORMAT LIBRARY=libref.catalog CNTLOUT=SAS-data-set;
   select format-name format-name...;
   exclude format-name format-name...;
RUN;
```

where

- `libref.catalog` is the name of the catalog in which the format is located.
- `SAS-data-set` is the name of the SAS data set that you want to create.
- `format-name` is the name of the format that you want to select or exclude.

The output control data set will contain variables that completely describe all aspects of each format, including optional settings. The output data set contains one observation per range per format in the specified catalog. You can use either the `SELECT` or `EXCLUDE` statement to include specific formats in the data set.

Creating a SAS data set from a format is very useful when you need to add information to a format but no longer have the the SAS data set you used to create the format. When you need to update a format, you can:

1. create a SAS data set from the values in a format using CNTLOUT=
2. edit the data set using any number of methods
3. create a format from the updated SAS data set using CNTLIN=.

Next let’s look at each of these steps individually.

**Example**

In the last example, you created the $AIRPORT. format. Suppose you want to add the following airport codes to the format:

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYC</td>
<td>Calgary, AB</td>
</tr>
<tr>
<td>YYZ</td>
<td>Toronto, ON</td>
</tr>
<tr>
<td>YQB</td>
<td>Quebec, QC</td>
</tr>
<tr>
<td>YUL</td>
<td>Montreal, QC</td>
</tr>
</tbody>
</table>

**Step 1: Create a SAS Data Set from the Format**

First, you write the $AIRPORT. format out as a SAS data set. In the code below, the output data set is named `Sasuser.Fmtdata`. The `SELECT` statement is used so that the resulting data set has only the data for the $AIRPORT. format. Without the `SELECT` statement, the data would have values for all the formats in the `Sasuser.Formats` catalog.
When you use the CNTLOUT= option, SAS creates an output data set that has many variables for storing information about the format. The output data set Sasuser.fmtdata has 50 rows and 21 columns. In the PRINT procedure below, the VAR statement specifies only a few of the variables to print:

```sas
proc print data=sasuser.fmtdata (obs=5) noobs;
  var fmtname start end label min max
default length fuzz;
run;
```

<table>
<thead>
<tr>
<th>FMTNAME</th>
<th>START</th>
<th>END</th>
<th>LABEL</th>
<th>MIN</th>
<th>MAX</th>
<th>DEFAULT</th>
<th>LENGTH</th>
<th>FUZZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIRPORT</td>
<td>AKL</td>
<td>AKL</td>
<td>Auckland</td>
<td>1</td>
<td>40</td>
<td>22</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>AIRPORT</td>
<td>AMS</td>
<td>AMS</td>
<td>Amsterdam</td>
<td>1</td>
<td>22</td>
<td>22</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AIRPORT</td>
<td>ANC</td>
<td>ANC</td>
<td>Anchorage, AK</td>
<td>1</td>
<td>40</td>
<td>22</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>AIRPORT</td>
<td>ARN</td>
<td>ARN</td>
<td>Stockholm</td>
<td>1</td>
<td>40</td>
<td>22</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>AIRPORT</td>
<td>ATH</td>
<td>ATH</td>
<td>Athens (Athina)</td>
<td>1</td>
<td>40</td>
<td>22</td>
<td>22</td>
<td>0</td>
</tr>
</tbody>
</table>

As you can see, the data set contains **End** and other variables that were not in the original data. When you use the CNTLIN= option, if there is no **End** variable in the data set, SAS assumes that the **Start** and **End** variables have the same value. When you write the format out as a data set using the CNTLOUT= option, both variables are in the data set.

**Step 2: Edit the Data Set**

The next step in updating the format is to edit the data set. You could use PROC SQL or a DATA step to add lines to the data set, or you could add rows using VIEWTABLE. Whatever method you choose, you must add values for the **FmtName**, **Start**, **End**, and **Label** variables. If the values for **Start** and **End** are the same, you must enter values for both variables or SAS will return an error. You do not have to add values for the other variables in the data set.

**Step 3: Create a Format from the SAS Data Set**

Once the data set is edited and saved, you can create a format from the data set using the CNTLIN= option. The following code creates the $AIRPORT. format and then uses FMTLIB to document it:

```sas
proc format library=sasuser cntlin=sasuser.fmtdata;
  select $airport; run;

proc format lib=sasuser fmlib;
  select $airport;
run;
```

The partial output shown below includes the new values in the format.
Table 17.5  Partial SAS Output

<table>
<thead>
<tr>
<th>START</th>
<th>END</th>
<th>LABEL (CONT'D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYD</td>
<td>END</td>
<td>Sydney, New South Wales</td>
</tr>
<tr>
<td>VIE</td>
<td>VIE</td>
<td>Wien (Vienna)</td>
</tr>
<tr>
<td>WLG</td>
<td>WLG</td>
<td>Wellington</td>
</tr>
<tr>
<td>YQB</td>
<td>YQB</td>
<td>Quebec, QC</td>
</tr>
<tr>
<td>YUL</td>
<td>YUL</td>
<td>Montreal, QC</td>
</tr>
<tr>
<td>YYC</td>
<td>YYC</td>
<td>Calgary, AB</td>
</tr>
<tr>
<td>YYZ</td>
<td>YYZ</td>
<td>Toronto, ON</td>
</tr>
</tbody>
</table>

Tip: For more information about using the CNTLOUT= option, see the SAS documentation for the FORMAT procedure.
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Creating Custom Formats Using the VALUE Statement

Character and numeric formats are created by using VALUE statements in a FORMAT procedure. When you specify a libref on the LIBRARY= option, the format is stored in the specified library. If no catalog name is specified, the format is saved in the "Formats" catalog by default.

Creating Formats with Overlapping Ranges

Use the MULTILABEL option to create a format that has overlapping ranges. When a format has overlapping ranges, the values in the format may have more than one label. This format can be used in procedures that support the MLF option.

Creating Custom Formats Using the PICTURE Statement

The PICTURE statement is used to create a template for formatting values of numeric variables. Pictures are specified using digit selectors, message characters, and directives.

Documenting Formats

Use the FMTLIB keyword in the PROC FORMAT statement to get documentation about the formats in the specified catalog. The output displays the format name, start and end values, and the label. You can also use the SELECT and EXCLUDE statements to process specific formats rather than an entire catalog.

Managing Formats

Because formats are saved as catalog entries, you use PROC CATALOG to copy, rename, delete, or create a listing of the entries in a catalog.

Using Custom Formats

Once you have created a format, you can reference it as you would reference a SAS format. If you have stored the format in a location other than Work.Formats, you must use the FMTSEARCH= system option to add the location to the search path so that SAS can locate the format. It can be useful to change the default FMTERR system option to NOFMTERR so that if SAS does not find a format you reference, it will substitute the w. or $w. format and continue processing.

You can permanently associate a format with a variable by modifying the data set using PROC DATASETS.
Creating Formats from SAS Data Sets

Use the CNTLIN= option to specify a SAS data set that you want to use to create a format. The SAS data set must contain the variables FmtName, Start, and Label. If the values have ranges, there must also be an End variable.

Creating SAS Data Sets from Formats

Use the CNTLOUT= option to create a SAS data set from a format. This is useful for maintaining formats because you can easily update a SAS data set.

Syntax

```sas
OPTIONS FMTSEARCH=(catalog catalog);
OPTIONS FMTERR | NOFMTERR;

LIBNAME libref 'SAS-data-library';
PROC FORMAT LIBRARY=libref;
   VALUE format-name (MULTILABEL)
       range1='label1'
       rangen'labeln';
RUN;

PROC FORMAT;
   PICTURE format-name
       range1='picture1'
       rangen='picturen';
RUN;

PROC FORMAT;
   PICTURE format-name
       value-or-range='picture'
       (DATATYPE=SAS-date-value-type);
RUN;

PROC FORMAT LIB=library FMTLIB;
   SELECT format-name1 format-name2;
   EXCLUDE format-name format-name2;
RUN;

PROC FORMAT LIBRARY=libref.catalog
   CNTLIN=SAS-data-set;
   SELECT format-name1 format-name2;
   EXCLUDE format-name format-name2;
RUN;
```
PROC FORMAT LIBRARY=libref.catalog
   CNTLOUT=SAS-data-set;
   SELECT format-name1 format-name2;
   EXCLUDE format-name format-name2;
RUN;

PROC CATALOG CATALOG=libref.catalog;
   CONTENTS <OUT=SAS-data-set>;
   COPY OUT=libref.catalog<options>;
   SELECT entry-name.entry-type(s);
   EXCLUDE entry-name.entry-type(s);
   DELETE entry-name.entry-type(s);
RUN;
QUIT;

PROC DATASETS LIB=SAS-library;
   MODIFY SAS-data-set;
   FORMAT variable(s) format;
QUIT;

---

Sample Programs

Creating a Multilabel Format

proc format;
   value dates (multilabel)
      '01jan2000’d - ‘31mar2000’d = ’1st Quarter’
      '01apr2000’d - ‘30jun2000’d = ’2nd Quarter’
      '01jul2000’d - ‘30sep2000’d = ’3rd Quarter’
      '01oct2000’d - ‘31dec2000’d = ’4th Quarter’
      '01jan2000’d - ‘30jun2000’d = ’First Half of Year’
      '01jul2000’d - ‘31dec2000’d = ’Second Half of Year’;
run;

Creating a Picture Format

proc format;
   picture rainamt
      0-2=’9.99 slight’
      2<-4=’9.99 moderate’
      4<-10=’9.99 heavy’
      other=’999 check value’;
run;

Creating a Picture Format Using Directives

proc format;
   picture mydate
Restructuring a SAS Data Set and Creating a Format from the Data

data sasuser.aports;
  keep Start Label FmtName;
  retain FmtName ‘$airport’;
  set sasuser.acities (rename=(Code=Start
                      City= Label));
runk;
proc format library=sasuser cntlin=sasuser.aports;
runk;

Creating a Format from a SAS Data Set

proc format lib=sasuser cntlout=sasuser.fmtdata;
  select $airport;
runk;

Points to Remember

- By default, SAS searches for formats in the Work.Formats and Library.Formats catalogs. If you store formats in other catalogs, you must use the FMTSEARCH= system option to tell SAS where to look for your formats.
- You can use the CNTLIN= option to create a format from a SAS data set, but the data
  - must contain the variables FmtName, Start, and Label
  - must contain the variable Type for character formats, unless the value for FmtName begins with a $
  - must contain the variable End if a range is specified.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which SAS system option is used to identify format catalogs to SAS?
   - a) FMTERR
   - b) FMTLIB
   - c) NOFMTERR
   - d) FMTSEARCH=

2. Given the following PROC FORMAT step, how is the value 70 displayed when the AGEGRP format is applied?

   proc format;
   picture agegrp
     1-<13=’00 Youth’
     13-<20=’00 Teen’
     20-<70=’00 Adult’
     70-high=’000 Senior’;
run;
  a  000 Senior
  b  70 Adult
  c  70 Senior
  d  070 Senior

3 When the NOFMTERR system option is in effect, what happens when SAS
encounters a format it cannot locate?
  a  Creates the format in the default Work.Formats directory and continues
     processing.
  b  Substitutes the $w. or w. format and continues processing.
  c  Stops processing and writes an error message to the log.
  d  Skips processing at that step and continues with the next step and writes a
     note to the log.

4 Which of the following variables must be in the data set that is specified on the
CNTLIN= option?
  a  End
  b  FmtName
  c  Value
  d  Description

5 Given the following code, what option is missing?

   proc format;
   value times (?)
     '00:00'-'04:59' = 'Red Eye'
     '05:00'-'11:59' = 'Morning'
     '12:00'-'17:59' = 'Afternoon'
     '18:00'-'23:59' = 'Evening'
     '00:00'-'11:59' = 'AM'
     '12:00'-'23:59' = 'PM';
   run;

   a  MULTILABEL
   b  MULTIRANGE
   c  MLF
   d  MULTIFORMAT

6 Which PROC FORMAT option is used to create a SAS data set from a format?

   a  CNTLIN=
   b  LIB=
   c  CNTLOUT=
   d  FMTLIB

7 Given the following OPTIONS statement, in what order will SAS search to find a
user-defined format?

   options fmtsearch=(work abc.newfmt sasuser);

   c  Work.Formats ▶ Abc.Newfmt ▶ Sasuser.Format
   d  the default search order
8 What option is used with PROC FORMAT to document the formats in a particular format catalog?
   a FMTSEARCH
   b FMTERR
   c CATALOG
   d FMTLIB

9 Which set of statements would you add to the PROC CATALOG code to copy the LEVELS. and $PICKS. formats from the Sasuser.Formats catalog to the Work.Formats catalog?

   proc catalog cat=sasuser.formats;
     ?
   run;

   a copy out=sasuser.formats;
      select levels.format $picks.format;
   b copy out=work.formats;
      select levels $picks;
   c copy out=work.formats;
      select levels.format picks.formatc;
   d copy out=work.formats;
      select levels.format $picks.format;

10 Given the following PROC FORMAT step, how is the value 6.1 displayed when the SKICOND format is applied?

   proc format;
     value skicond
       0-3=’Poor’
       3<-6=’Fair’
       6<-9=’Good’
       9<-high=’Excellent’;
   run;

   a 6.1
   b Fair
   c Good
   d .
CHAPTER 18

Modifying SAS Data Sets and Tracking Changes

Overview
Introduction 634
Objectives 635
Using the MODIFY Statement 635
Modifying All Observations in a SAS Data Set 637
Example 637
Modifying Observations Using a Transaction Data Set 638
Example 639
Handling Duplicate Values 640
Handling Missing Values 641
Modifying Observations Located by an Index 641
Example 642
Handling Duplicate Values 643
Controlling the Update Process 645
Example 645
Monitoring I/O Error Conditions 646
Using _IORC_ with %SYSRC 646
Example 647
Understanding Integrity Constraints 647
General Integrity Constraints 648
Referential Integrity Constraints 648
Placing Integrity Constraints on a Data Set 649
Example 650
How Constraints Are Enforced 651
Example 651
Copying a Data Set and Preserving Integrity Constraints 652
Copying a Data Set and Preserving Integrity Constraints 652
Documenting Integrity Constraints 653
Example 653
Removing Integrity Constraints 653
Example 654
Understanding Audit Trails 654
Initiating and Reading Audit Trails 655
Example 656
Reading Audit Trail Files 657
Examples 657
Controlling Data in the Audit Trail 657
Data Set Variables 658
Audit Trail Variables 658
Values of the _ATOPCODE_ Variable 659
Using the LOG Statement to Control the Data in the Audit Trail 659
Overview

Introduction

There are times when you want to modify the observations in a SAS data set without replacing the data set. You can do this in a DATA step with the MODIFY statement. Using the MODIFY statement, you can replace, delete, or append observations in an
existing data set without creating an additional copy of the data. In this chapter, you
learn to modify all the observations in a data set, matching observations using a BY
statement, and observations located using an index.

When you modify data, it’s often essential to safeguard your data and track the
changes that are made. In this chapter you learn how to create integrity constraints to
protect your data. You will also learn two different methods of tracking changes —
audit trails and generation data sets. You use audit trails to track changes that are
made to a data set in place, and you use generation data sets to track changes that are
made when the data set is rebuilt.

Objectives

In this chapter, you learn to

- use the MODIFY statement to update all observations in a SAS data set
- use a transaction data set to make modifications to a SAS data set
- use an index to locate observations to modify in a SAS data set
- place integrity constraints on variables in a SAS data set
- initiate and manage an audit trail file
- create and process generation data sets.

Using the MODIFY Statement

When you submit a DATA step to create a SAS data set that is also named in a
MERGE, UPDATE, or SET statement, SAS creates a second copy of the input data set.
Once execution is complete, SAS deletes the original copy of the data set. As a result,
the original data set is replaced by the new data set. The new data set can contain a
different set of variables than the original data set and the attributes of the variables in
the new data set can be different from those of the original data set.
In contrast, when you submit a DATA step to create a SAS data set that is also named in the *MODIFY statement*, SAS does not create a second copy of the data but instead updates the data set in place. Any variables can be added to the program data vector (PDV), but they are not written to the data set. Therefore, the set of variables in the data set does not change when the data is modified.

When you use the MODIFY statement, there is an implied REPLACE statement at the bottom of the DATA step instead of an OUTPUT statement. Using the MODIFY statement, you can update

- every observation in a data set
- observations using a transaction data set and a BY statement
- observations located using an index.

**CAUTION:**
If the system terminates abnormally while a DATA step that is using the MODIFY statement is processing, you can lose data and possibly damage your master data set. You can recover from the failure by

- restoring the master file from a backup and restarting the step, or
- keeping an audit trail file and using it to determine which master observations have been updated.

You will learn about audit trails later in this chapter. △

First let’s take a look at using the MODIFY statement to modify all the observations in the data set.
Modifying All Observations in a SAS Data Set

When every observation in a SAS data set requires the same modification, you can use the MODIFY statement and specify the modification using an assignment statement.

General form, MODIFY statement with an assignment statement:

```
DATA SAS-data-set;
  MODIFY SAS-data-set;
  existing-variable = expression;
RUN;
```

where

- `SAS-data-set` is the name of the SAS data set that you want to modify.
- `existing-variable` is the name of the variable whose values you want to update.
- `expression` is a function or other expression that you want to apply to the variable.

Example

Suppose an airline has decided to give passengers more leg room. To do so the airline must decrease the number of seats in the business and economy classes. The SAS data set `Capacity` has the variables `CapEcon` and `CapBusiness` that hold values for the number of seats in the economy and business classes.

In the program below, the assignment statement for `CapEcon` reduces the number of seats in the economy class to 95% of the original number, and the assignment statement for `CapBusiness` reduces the number of seats in the business class to 90% of the original number. The INT function is used in both assignment statements to return the integer portion of the result.

```
Note: If you choose to run this example, you must copy the data set `Capacity` from the `Sasuser` library to the `Work` library.
```

```
proc print data=capacity (obs=4);
run;

data capacity;
  modify capacity;
  CapEcon = int(CapEcon * .95);
  CapBusiness = int(CapBusiness * .90);
run;

proc print data=capacity (obs=4);
run;
```
The following output shows the data before the MODIFY statement.

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>RouteID</th>
<th>Origin</th>
<th>Dest</th>
<th>Cap1st</th>
<th>CapBusiness</th>
<th>CapEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA00100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>2</td>
<td>IA00201</td>
<td>0000002</td>
<td>LHR</td>
<td>RDU</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>3</td>
<td>IA00300</td>
<td>0000003</td>
<td>RDU</td>
<td>FRA</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>4</td>
<td>IA00400</td>
<td>0000004</td>
<td>FRA</td>
<td>RDU</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
</tbody>
</table>

The following output shows the data after the MODIFY statement. You can see that the values in CapBusiness and CapEcon have been reduced.

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>RouteID</th>
<th>Origin</th>
<th>Dest</th>
<th>Cap1st</th>
<th>CapBusiness</th>
<th>CapEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA00100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>14</td>
<td>27</td>
<td>154</td>
</tr>
<tr>
<td>2</td>
<td>IA00201</td>
<td>0000002</td>
<td>LHR</td>
<td>RDU</td>
<td>14</td>
<td>27</td>
<td>154</td>
</tr>
<tr>
<td>3</td>
<td>IA00300</td>
<td>0000003</td>
<td>RDU</td>
<td>FRA</td>
<td>14</td>
<td>27</td>
<td>154</td>
</tr>
<tr>
<td>4</td>
<td>IA00400</td>
<td>0000004</td>
<td>FRA</td>
<td>RDU</td>
<td>14</td>
<td>27</td>
<td>154</td>
</tr>
</tbody>
</table>

Modifying Observations Using a Transaction Data Set

You can use a MODIFY statement to update all observations in a data set, but there are times when you only want to update selected observations. You can modify a master SAS data set with values in a transaction data set by using the MODIFY statement with a BY statement to apply updates by matching observations.

General form, MODIFY statement with a BY statement:

```
DATA SAS-data-set;
  MODIFY SAS-data-set transaction-data-set;
  BY key-variable;

RUN;
```

where

- **SAS-data-set**
  is the name of the SAS data set that you want to modify (also called the master data set).

- **transaction-data-set**
  is the name of the SAS data set in which the updated values are stored.

- **key-variable**
  is the name of the variable whose values will be matched in the master and transaction data sets.

**Note:** In the MODIFY statement, you must list the master data set followed by the transaction data set.
The BY statement matches observations from the transaction data set with observations in the master data set. When the MODIFY statement reads an observation from the transaction data set, it uses dynamic WHERE processing (SAS internally generates a WHERE statement) to locate the matching observation in the master data set. The matching observation in the master data set can be replaced, deleted, or appended. By default, the observation is replaced.

Note: Because the MODIFY statement uses WHERE processing to locate matching observations, neither data set requires sorting. However, having the master data set sorted or indexed and the transaction data set sorted reduces processing overhead, especially for large files.

---

**Example**

Suppose you have a master data set, *Capacity*, which has route numbers for an airline. Some of the route numbers have changed, and the changes are stored in a transaction data set, *Newrtnum*. The master data set is updated by matching values of the variable *FlightID*.

```sas
proc print data=capacity(obs=5);
run;

data capacity;
modify capacity sasuser.newrtnum;
by flightid;
run;

proc print data=capacity(obs=5);
run;
```

The following PROC PRINT output displays the first five rows of the data set *Capacity* before updates were applied.

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>RouteID</th>
<th>Origin</th>
<th>Dest</th>
<th>Cap1st</th>
<th>CapBusiness</th>
<th>CapEcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1A00100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>2</td>
<td>1A00201</td>
<td>0000002</td>
<td>LHR</td>
<td>RDU</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>3</td>
<td>1A00300</td>
<td>0000003</td>
<td>RDU</td>
<td>FRA</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>4</td>
<td>1A00400</td>
<td>0000004</td>
<td>FRA</td>
<td>RDU</td>
<td>14</td>
<td>30</td>
<td>163</td>
</tr>
<tr>
<td>5</td>
<td>1A00500</td>
<td>0000005</td>
<td>RDU</td>
<td>JFK</td>
<td>16</td>
<td></td>
<td>261</td>
</tr>
</tbody>
</table>

As you can see in this PROC PRINT output, three values of *FlightID* in *Newrtnum* have matches in *Capacity*. For each matching observation, the values for *RouteID* are updated.
Handling Duplicate Values

When you use the MODIFY and BY statements to update a data set, WHERE processing starts at the top of the master data set and finds the first match and updates it. Let’s take a look at what happens if there are duplicate values in the master or transaction data sets. Suppose you have the following code to make updates to the master data set \( M \) using the transaction data set \( T \):

```plaintext
data m;
  modify m t;
  by a;
run;
```

If duplicate values of the BY variable exist in the master data set, only the first observation in the group of duplicate values is updated because WHERE processing begins at the top of the data set and updates the first match.

If duplicate values of the BY variable exist in the transaction data set, the duplicate values overwrite each other so that the last value in the group of duplicate transaction values is the result in the master data set.

You can avoid overwriting duplicate values by writing an accumulation statement so that all the observations in the transaction data set are added to the master observation.

**Tip:** If duplicate values exist in both the master and transaction data sets, you can use PROC SQL to apply the duplicate values in the transaction data set to the duplicate values in the master data set in a one-to-one correspondence.
Handling Missing Values

If there are missing values in the transaction data set, SAS does not replace the data in the master data set with missing values unless they are special missing values.

*Note:* A special missing value is a type of numeric missing value that enables you to represent different categories of missing data by using the letters A-Z or an underscore. You designate special missing values using the MISSING statement in the DATA step. For more information, see the SAS documentation.

You can specify how missing values in the transaction data set are handled by using the UPDATEMODE= option in the MODIFY statement.

General form, MODIFY statement with the UPDATEMODE= option:

```
MODIFY master-data-set transaction-data-set
    UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK;
```

where

- `master-data-set` is the name of the SAS data set that you want to modify.
- `transaction-data-set` is the name of the SAS data set in which the updated values are stored.

**MISSINGCHECK**

prevents missing values in the transaction data set from replacing values in the master data set unless they are special missing values. MISSINGCHECK is the default.

**NOMISSINGCHECK**

allows missing values in the transaction data set to replace the values in the master data set. Special missing values in the transaction data set still replace values in the master data set.

Modifying Observations Located by an Index

You have learned that you can use a BY statement to access values you want to update in a master data set by matching. When you have an indexed data set, you can use the index to directly access the values you want to update. To do this, you use

- a MODIFY statement with the KEY= option to name an indexed variable to locate the observations for updating

- another data source (typically a SAS data set named on a SET statement or an external file read by an INPUT statement) to provide a like-named variable whose values are supplied to the index.
General form, MODIFY statement with the KEY= option:

\[ \text{MODIFY SAS-data-set KEY=\text{index-name};} \]

where

- **SAS-data-set** is the master data set, or the data set that you want to update.
- **index-name** is the name of the simple or composite index that you are using to locate observations.

Updating with an index is different from updating using a BY statement. When you use the MODIFY statement with the KEY= option to name an index,

- you must explicitly specify the update that you want to occur. No automatic overlay of non-missing values in the transaction data set occurs as it does with the MODIFY/BY method.
- each observation in the transaction data set must have a matching observation in the master data set. If you have multiple observations in the transaction data set for one master observation, only the first observation in the transaction data set is applied. The other observations generate runtime errors and terminate the DATA step (unless you use the UNIQUE option, which is discussed later in this chapter).

Example

Suppose that airline cargo weights for 1999 are stored in the master data set **Cargo99**, which has a composite index named **FlightDte** on the variables **FlightID** and **Date**. Some of the data is incorrect and the data set needs to be updated. The correct cargo data is stored in the transaction data set **Newcgnum**.

In the program below, the KEY= option specifies the **FlightDte** index. When a matching observation is found in **Cargo99**, three variables (**CapCargo**, **CargoWgt**, and **CargoRev**) are updated.

Note: If you choose to run this example, you must copy the data set **Cargo99** from the **Sasuser** library to the **Work** library.

```
proc print data=cargo99(obs=5);
run;

data cargo99;
  set sasuser.newcgnum (rename =
    (capcargo = newCapCargo
cargowgt = newCargoWgt
cargorev = newCargoRev));
  modify cargo99 key=flightdte;
capcargo = newcapcargo;
cargowgt = newcargowgt;
cargorev = newcargorev;
run;

proc print data=cargo99(obs=5);
run;
```
The output below shows the first five observations of the SAS data set Cargo99 before it was modified by Newcgnum.

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>RouteID</th>
<th>Origin</th>
<th>Dest</th>
<th>CapCargo</th>
<th>Date</th>
<th>CargoWgt</th>
<th>CargoRev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>01JAN1999</td>
<td>4600</td>
<td>$111,720.00</td>
</tr>
<tr>
<td>2</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>01AUG1999</td>
<td>4400</td>
<td>$109,270.00</td>
</tr>
<tr>
<td>3</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>20AUG1999</td>
<td>4400</td>
<td>$109,270.00</td>
</tr>
<tr>
<td>4</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>02SEP1999</td>
<td>4740</td>
<td>$115,130.00</td>
</tr>
<tr>
<td>5</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>29DEC1999</td>
<td>4420</td>
<td>$108,290.00</td>
</tr>
</tbody>
</table>

The output below shows the first five observations of the SAS data set Cargo99 after it was modified by Newcgnum. Notice that the three variables in the first observation were updated by the values in Newcgnum.

<table>
<thead>
<tr>
<th>Obs</th>
<th>FlightID</th>
<th>RouteID</th>
<th>Origin</th>
<th>Dest</th>
<th>CapCargo</th>
<th>Date</th>
<th>CargoWgt</th>
<th>CargoRev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>35055</td>
<td>01JAN1999</td>
<td>4400</td>
<td>$121,079.90</td>
</tr>
<tr>
<td>2</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>01AUG1999</td>
<td>4400</td>
<td>$109,270.00</td>
</tr>
<tr>
<td>3</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>20AUG1999</td>
<td>4400</td>
<td>$109,270.00</td>
</tr>
<tr>
<td>4</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>02SEP1999</td>
<td>4740</td>
<td>$116,130.00</td>
</tr>
<tr>
<td>5</td>
<td>AA01100</td>
<td>0000001</td>
<td>RDU</td>
<td>LHR</td>
<td>82400</td>
<td>29DEC1999</td>
<td>4420</td>
<td>$108,290.00</td>
</tr>
</tbody>
</table>

### Handling Duplicate Values

When you use an index to locate values to update, duplicate values of the indexed variable in the transaction data set may cause problems. Let’s look at what happens with various scenarios when you use the following code to update the master data set M with values from the transaction data set T. The index on the M data set is built on the variable A:

```sas
data m;
  set t (rename=(b=newb));
  modify m key=index;
  b=newb;
run;
```

If there are *duplications in the master data set*, only the first occurrence is updated.

<table>
<thead>
<tr>
<th>T</th>
<th>M</th>
<th>M updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>1</td>
</tr>
</tbody>
</table>

*Tip:* If you want all duplicates in the master data set to be updated with the transaction value, use a DO loop to execute a SET statement with the KEY= option multiple times.
If there are nonconsecutive duplications in the transaction data set, SAS updates the first observation in the master data set. The last duplicate transaction value is the result in the master data set after the update.

If there are consecutive duplications in the transaction data set, some of which do not have a match in the master data set, then SAS performs a one-to-one update until it finds a non-match. At that time, the DATA step terminates with an error.

Adding the UNIQUE option to the MODIFY statement allows you to avoid the error in the DATA step. The UNIQUE option causes the DATA step to return to the top of the index each time it looks for a match for the value from the transaction data set. The UNIQUE option can only be used with the KEY= option.

General form, MODIFY statement with the UNIQUE option:

```
MODIFY SAS-data-set KEY=index-name /UNIQUE;
```

where

- `SAS-data-set` positions the input pointer on a specified column.
- `index-name` is the name of the variable that is being created.

You can specify the UNIQUE option to
- apply multiple transactions to one master observation
- identify that each observation in the master data set contains a unique value of the index variable.

When you use the UNIQUE option and there are consecutive duplications in the transaction data set, SAS updates the first observation in the master data set. This is similar to what happens when you have nonconsecutive duplications in the transaction data set. If the values in the transaction data set should be added to the value in the master data set, you can write a statement to accumulate the values from all the duplicates.
Modifying SAS Data Sets and Tracking Changes

Controlling the Update Process

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. If no other statements are present, SAS writes the current observation to its original place in the SAS data set. This action occurs by default through an implied REPLACE statement at the bottom of the DATA step.

However, you can override this default behavior by explicitly adding the OUTPUT, REPLACE, or REMOVE statement.

General form for OUTPUT, REPLACE, and REMOVE statements:

**OUTPUT;**
**REPLACE;**
**REMOVE;**

where

**OUTPUT**
specifies that the current observation be written to the **end** of the data set.

**REPLACE**
specifies that the current observation be rewritten to the **same location** in the data set.

**REMOVE**
specifies that the current observation be **deleted** from the master data set.

Using OUTPUT, REPLACE, or REMOVE in a DATA step overrides the default replacement of observations. If you use any one of these statements in a DATA step, you must explicitly program each action that you want to take. You can use these three statements together as long as the sequence is logical.

**CAUTION:**
If you use an OUTPUT statement in conjunction with a REPLACE or REMOVE statement, be sure the OUTPUT statement is executed after any REPLACE or REMOVE statements to ensure the integrity of the index position.

**Example**

If the SAS data set *Transaction* has a variable named **code** having values of *yes*, *no*, and *new*, you can submit the following program to

- delete rows where the value of **code** is *no*
- update rows where the value of **code** is *yes*
append rows where the value of code is new.

data master;
  set transaction;
  modify master key = id;
  a = b;
  if code = 'no' then remove;
  else if code = 'yes' then replace;
  else if code = 'new' then output;
run;

Note: You cannot run this example because Transaction and Master are fictitious data sets.

---

Monitoring I/O Error Conditions

When you use the MODIFY statement with a BY statement or KEY= option to update a data set, error checking is important for several reasons. The most important reason is that these tools use nonsequential access methods, so there is no guarantee that an observation will be located that satisfies the request. Error checking enables you to perform updates or not, depending on the outcome of the I/O condition.

The automatic variable _IORC_(Input Output Return Code) is created when you use the MODIFY statement with the BY statement or KEY= option. The value of _IORC_ is a numeric return code that indicates the status of the most recently executed I/O operation. Checking the value of this variable allows you to detect abnormal I/O conditions and direct execution in particular ways rather than having the application terminate abnormally.

Using _IORC_ with %SYSRC

Because the values of the _IORC_ automatic variable are internal and subject to change, %SYSRC, an autocall macro, was created to enable you to test for specific I/O conditions while protecting your code from future changes in _IORC_ values.

General form, _IORC_ with the %SYSRC autocall macro:

```
IF_IORC_%SYSRC (mnemonic) THEN...
```

where

- `mnemonic` is a code for a specific I/O condition.

Note: %SYSRC is in the autocall library. You must have the MACRO system option in effect to use this macro.

When you use %SYSRC, you can check the value of _IORC_ by specifying one of the mnemonics listed in the table below.
Modifying SAS Data Sets and Tracking Changes  △  Understanding Integrity Constraints

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DSENMR</td>
<td>The observation in the transaction data set does not exist in the master data set (used only with the MODIFY and BY statements).</td>
</tr>
<tr>
<td>_DSEMTR</td>
<td>Multiple transaction data set observations do not exist on the master data set (used only with the MODIFY and BY statements).</td>
</tr>
<tr>
<td>_DSENOM</td>
<td>No matching observation (used with the KEY= option).</td>
</tr>
<tr>
<td>_SOK</td>
<td>The observation was located. _SOK has a value of 0.</td>
</tr>
</tbody>
</table>

Example

Suppose you are using the MODIFY statement with the KEY= option to update a SAS data set. In the program below, when _IORC_ has the value _SOK_, the observation is updated. When _IORC_ has the value _DSENOM_, no matching observation is found, so the observation is appended to the data set by the OUTPUT statement and _ERROR_ is reset to 0 in the do loop.

```sas
data master;
  set transaction;
  modify master key = id;
  if _IORC_=%sysrc(_sok) then
    do;
      a = b;
      replace;
    end;
  else
    if _IORC_=%sysrc(_dsenom) then
      do;
        output;
        _ERROR_ = 0;
      end;
  end;
run;
```

Tip: For more information about the _IORC_ automatic variable and %SYSRC, see information about error-checking tools in the SAS documentation.

Understanding Integrity Constraints

Now that you know how to modify data in place, you may be wondering how you can protect or insure the integrity of your data when it is modified. Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a variable in a data set. SAS enforces integrity constraints when values associated with a variable are added, updated, or deleted using techniques that modify data in place, such as

- a DATA step with the MODIFY statement
- an interactive data editing window
- PROC SQL with the INSERT INTO, SET, or UPDATE statements
- PROC APPEND.
When you place integrity constraints on a SAS data set, you specify the type of constraint that you want to create. Each constraint has a different action.

<table>
<thead>
<tr>
<th>Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECK</td>
<td>ensures that a specific set or range of values are the only values in a column. It can also check the validity of a value in one column based on a value in another column within the same row.</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>guarantees that a column has non-missing values in each row.</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>enforces uniqueness for the values of a column.</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>uniquely defines a row within a table, which can be a single column or a set of columns. A table can have only one primary key. The PRIMARY KEY constraint includes the attributes of the NOT NULL and UNIQUE constraints.</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>links one or more rows in a table to a specific row in another table by matching a column or set of columns in one table with the PRIMARY KEY defined in another table. This parent/child relationship limits modifications made to both tables. The only acceptable values for a FOREIGN KEY are values of the PRIMARY KEY or missing values.</td>
</tr>
</tbody>
</table>

*Note:* When you place integrity constraints on a SAS data set, you specify the type of constraint that you want to create. Each constraint has a different action. △

You can use integrity constraints in two ways, general and referential. General constraints operate within a data set, and referential constraints operate between data sets.

**General Integrity Constraints**

General integrity constraints enable you to restrict the values of variables within a single data set. The following four integrity constraints can be used as general integrity constraints:

- CHECK
- NOT NULL
- UNIQUE
- PRIMARY KEY.

*Note:* A PRIMARY KEY constraint is a general integrity constraint as long as it does not have any FOREIGN KEY constraints referencing it. When PRIMARY KEY is used as a general constraint it is simply a shortcut for assigning the NOT NULL and UNIQUE constraints. △

**Referential Integrity Constraints**

Referential constraints enable you to link the data values of a column in one data set to the data values of columns in another data set. You create a referential integrity constraint when a FOREIGN KEY integrity constraint in one data set references a PRIMARY KEY integrity constraint in another data set. To create a referential integrity constraint, you must follow two steps:

1. Define a PRIMARY KEY constraint on the first data set.
Define a FOREIGN KEY constraint on other data sets.

**Placing Integrity Constraints on a Data Set**

Integrity constraints can be created using

- the DATASETS procedure
- the SQL procedure.

Although you can use either procedure to create integrity constraints on existing data sets, you must use PROC SQL if you want to create integrity constraints at the same time that you create the data set. In this chapter you learn to use PROC DATASETS to place integrity constraints on an existing data set.

General form, DATASETS procedure with the IC CREATE statement:

```sas
PROC DATASETS LIB=libref <NOLIST>;
   MODIFY SAS-data-set;
   IC CREATE constraint-name=constraint
       <MESSAGE='Error Message'>;
QUIT;
```

where

- `libref` is the library in which the data set is stored. If you do not specify the LIB= option, the procedure uses the Work library.
- `NOLIST` suppresses the directory listing.
- `SAS-data-set` is the name of the data set to which you want to apply the integrity constraint.
- `constraint-name` is any name that you wish to give the integrity constraint.
- `constraint` is the type of constraint that you are creating, specified in the following format:
  - NOT NULL (variable)
  - UNIQUE (variables)
  - CHECK (where-expression)
  - PRIMARY KEY (variables)
  - FOREIGN KEY (variables) REFERENCES table-name.
- `Error Message` is an optional message that you want the user to see if the constraint is violated.

**Note:** You can use IC or INTEGRITY CONSTRAINT interchangeably.

**Tip:** To learn how to create integrity constraints using the SQL procedure, see Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159.
Example

Suppose you have a data set that contains route information and passenger capacity for each class in an airline. You need to create integrity constraints to ensure that when the data set is updated

- the route ID number is both unique and required (PRIMARY KEY)
- the capacity for business class passengers must either be missing or be greater than the capacity for first class passengers (CHECK).

In the code below, the IC CREATE statement is used to create two general integrity constraints on variables in the data set Capinfo:

- The PRIMARY KEY constraint is placed on the RouteID variable. This constraint ensures that when values of RouteID are updated, they must be unique and nonmissing.

  Note: The same effect could be achieved by applying both the UNIQUE and NOT NULL constraints, but the PRIMARY KEY constraint is used as a shortcut.

- The CHECK constraint uses the WHERE expression to ensure that the only values of CapBusiness that are allowed are those greater than Cap1st or missing.

  Note: If you choose to run this example, you must copy the data set Capinfo from the Sasuser library to the Work library.

```sas
proc datasets nolist;
  modify capinfo;
  ic create PKIDInfo = primary key (routeid)
    message = 'You must supply a Route ID Number';
  ic create Class1 = check (where = (cap1st < capbusiness or capbusiness = .))
    message = 'Cap1st must be less than CapBusiness';
quit;
```

Note: For the UNIQUE and PRIMARY KEY constraints, SAS builds indexes on the columns involved if an appropriate index does not already exist. Any index created by an integrity constraint can be used for other purposes, such as WHERE processing or the KEY= option in a SET statement.

Tip: For more information about creating integrity constraints, see the SAS documentation for the DATASETS procedure.
How Constraints Are Enforced

Once integrity constraints are in place, SAS enforces them whenever you modify values in the data set in place. Techniques for modifying data in place include using

- a DATA step with the MODIFY statement
- interactive data editing windows
- PROC SQL with the INSERT INTO, SET, or UPDATE statements
- PROC APPEND.

Example

The code in the previous example placed a check constraint on `Cap1st` and `CapBusiness` to ensure that values for the capacity in business class were either greater than first class or missing. Suppose that you ran the following program to triple the capacity in first class. This would probably violate the check constraint for some observations.

```sas
data capinfo;
  modify capinfo;
  cap1st=cap1st*3;
run;
```

The observations that failed to pass the integrity constraint are written to the SAS log. As you can see, all these observations would have had values of `Cap1st` greater than those of `CapBusiness`. 
If you used the VIEWTABLE window or another window to make this update, SAS would have displayed the error message defined for the integrity constraint.

Note: Rejected observations can be collected in a special file using the audit trail functionality that you will learn about later in this chapter. △

Copying a Data Set and Preserving Integrity Constraints

The APPEND, COPY, CPORT, CIMPORT, and SORT procedures preserve integrity constraints when their operation results in a copy of the original data file. Integrity constraints are also preserved if you copy a data set using the SAS Explorer window.

Tip: For more information about preserving integrity constraints, see the SAS documentation.

Note: Rejected observations can be collected in a special file using the audit trail functionality that you will learn about later in this chapter. △

Copying a Data Set and Preserving Integrity Constraints

The APPEND, COPY, CPORT, CIMPORT, and SORT procedures preserve integrity constraints when their operation results in a copy of the original data file.

Tip: For more information about preserving integrity constraints, see the SAS documentation.
Documenting Integrity Constraints

To view the descriptor portion of your data, including the integrity constraints that you have placed on a data set, you can use the CONTENTS statement in the DATASETS procedure.

General form, DATASETS procedure with the CONTENTS statement:

\[
\text{PROC DATASETS LIB=libref <NOLIST>;} \\
\text{ CONTENTS DATA=SAS-data-set;} \\
\text{ QUIT;}
\]

where

- \text{libref}  
  
is the library in which the data set is stored.

- \text{NOLIST}  
  
suppresses the directory listing.

- \text{SAS-data-set}  
  
is the name of the data set that you want information about.

Note: The CONTENTS statement in the DATASETS procedure results in the same information as the CONTENTS procedure.

Example

The following code displays information about the Capinfo data set, including the integrity constraints that were added to this data set in the last example. Notice that the NLIST option is used here to suppress the listing of all data sets in the Work library. With this option, only the information for the Capinfo data set is listed.

\[
\text{proc datasets nolist;} \\
\text{ contents data=capinfo;} \\
\text{ quit;} \\
\]

Only the integrity constraints portion of the output is shown below.

<table>
<thead>
<tr>
<th>Integrity # Constraint</th>
<th>Type</th>
<th>Variables</th>
<th>Where Clause</th>
<th>User Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Class1</td>
<td>Check</td>
<td></td>
<td>(Class1&lt;CapBusiness) or</td>
<td>First Class Capacity must less than</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(CapBusiness&lt;=)</td>
<td>Business Capacity</td>
</tr>
<tr>
<td>2 PKIDInfo</td>
<td>Primary Key</td>
<td>RouteID</td>
<td></td>
<td>You must supply a Route ID Number</td>
</tr>
</tbody>
</table>

Removing Integrity Constraints

To remove an integrity constraint from a data set, use the DATASETS procedure with the IC DELETE statement.
General form, DATASETS procedure with the IC DELETE statement:

```
PROC DATASETS LIB=libref <NOLIST>;
   MODIFY SAS-data-set;
   IC DELETE constraint-name;
QUIT;
```

where

- `libref` is the name of the library in which the data set is stored. If you do not specify the `LIB=` option, the procedure uses the `Work` library.
- `NOLIST` suppresses the directory listing.
- `SAS-data-set` is the name of the data set that has the integrity constraint.
- `constraint-name` is the name of the integrity constraint that you want to delete.

Example

The code below removes the integrity constraints on the `Capinfo` data set:

```
proc datasets;
   modify capinfo;
   ic delete pkidinfo;
   ic delete class1;
quit;
```

A message is written to the SAS log when the integrity constraint is deleted.

Table 18.3 SAS Log

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Log Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>modify capinfo;</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>ic delete pkidinfo;</td>
<td><strong>NOTE</strong>: Integrity constraint PKIDInfo deleted.</td>
</tr>
<tr>
<td>55</td>
<td>ic delete class1;</td>
<td><strong>NOTE</strong>: All integrity constraints defined on WORK.CAPINFO.DATA have been deleted.</td>
</tr>
<tr>
<td>56</td>
<td>run;</td>
<td><strong>NOTE</strong>: Integrity constraint PKIDInfo deleted.</td>
</tr>
</tbody>
</table>

Understanding Audit Trails

As you modify a data set, you may want to track the changes that you make by using an audit trail. An audit trail is an optional SAS file that logs modifications to a SAS table. Audit trails are used to track changes that are made to the data set in place. Specifically, audit trails track changes made with

- the VIEWTABLE window
- the data grid
the MODIFY statement in the DATA step
- the UPDATE, INSERT, or DELETE statement in PROC SQL.

For each addition, deletion, and update to the data, the audit trail automatically stores a copy of the variables in the observation that was updated, and information such as who made the modification, what was modified, and when the modification was made. It can also store additional information in user-defined variables.

The following PROC CONTENTS output lists the variables in an audit trail file for a data set that has two variables, A and B. You will learn more about these variables later in this chapter.

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
<th>Format</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>Num</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>Num</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Who</td>
<td>Char</td>
<td>20</td>
<td>16</td>
<td></td>
<td>Name</td>
</tr>
<tr>
<td>4</td>
<td>Why</td>
<td>Char</td>
<td>20</td>
<td>36</td>
<td></td>
<td>Reason</td>
</tr>
<tr>
<td>5</td>
<td><em>ATDATETIME</em></td>
<td>Num</td>
<td>8</td>
<td></td>
<td>56 DATE TIME</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><em>ATMESSAGE</em></td>
<td>Char</td>
<td>8</td>
<td></td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><em>ATOBNO</em></td>
<td>Num</td>
<td>8</td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><em>ATPCODE</em></td>
<td>Char</td>
<td>2</td>
<td></td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><em>ATRCODE</em></td>
<td>Num</td>
<td></td>
<td></td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><em>ATUSERID</em></td>
<td>Char</td>
<td>32</td>
<td></td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

**CAUTION:**

Any procedure or action that replaces the data set (such as the DATA step, CREATE TABLE in PROC SQL, or SORT without the OUT= option) will delete the audit trail. Audit trails should not be deleted with system tools such as Windows Explorer.

A SAS table can have only one audit trail file. The audit trail file
- is a read-only file
- is created by PROC DATASETS
- must be in the same library as the data file associated with it
- has the same name as the data set it is monitoring, but with a member type of AUDIT.

Next let’s take a look at how you initiate an audit trail on a SAS data set.

## Initiating and Reading Audit Trails

You initiate an audit trail using the DATASETS procedure with the AUDIT and INITIATE statements.
General form, DATASETS procedure to initiate an audit trail:

```
PROC DATASETS LIB=libref <NOLIST>;
   AUDIT SAS-data-set <SAS-password>;
   INITIATE;
RUN;
QUIT;
```

where

- `libref` is the name of the library where the data set to be audited resides
- `NOLIST` suppresses the directory listing
- `SAS-data-set` is the name of the SAS data set that you want to audit
- `SAS-password` is the SAS data set password, if one exists
- `INITIATE` begins the audit trail on the data set specified in the AUDIT statement.

**Example**

The following code initiates an audit trail on the data set `Capinfo`.

*Note:* If you choose to run this example, you must copy the data set `Capinfo` from the `Sasuser` library to the `Work` library.

```
proc datasets nolist;
   audit capinfo;
   initiate;
quit;
```

**Table 18.4  SAS Log**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td><code>audit capinfo;</code></td>
</tr>
<tr>
<td>61</td>
<td><code>initiate;</code></td>
</tr>
<tr>
<td>62</td>
<td><code>run;</code></td>
</tr>
<tr>
<td>62</td>
<td><code>NOTE: The data set WORK.CAPINFO.AUDIT has 0 observations and 13 variables.</code></td>
</tr>
</tbody>
</table>

*Note:* The audit trail file uses the SAS password that is assigned to the parent data set; therefore, it is recommended that you alter the password for the parent data set. Use the `ALTER=` data set option to assign an alter-password to a SAS data set or to
access a read-, write-, or alter-protected SAS data set. If another password is used or no password is used, then the audit file is still created, but is not protected.

*Tip:* For more information about audit trails, see the SAS documentation for the DATASETS procedure.

---

## Reading Audit Trail Files

When the audit trail is initiated, it has no observations until the first modification is made to the audited data set. When the audit trail file contains data, you can read it with any component of SAS that reads a data set. To refer to the audit trail file, use the `TYPE=` data set option.

### General form, `TYPE=` data set option to specify an audit file:

```
(TYPE=AUDIT)
```

### Examples

The following PROC CONTENTS code displays the contents of the audit trial file:

```
proc contents data=mylib.sales (type=audit);
run;
```

The following PROC PRINT code lists the data in the audit trail file for the data set `Capinfo`:

```
proc print data=capinfo (type=audit);
run;
```

---

## Controlling Data in the Audit Trail

Now that you have seen how to initiate audit trails and read an audit trail file, let's look more closely at the information the audit trail file contains. The audit trail file can contain three types of variables:

- *data set variables* that store copies of the columns in the audited SAS data set
- *audit trail variables* that automatically store information about data modifications
- *user variables* that store user-entered information.
You can use additional statements in the PROC DATASETS step to control which variables appear in the audit trail. Let’s take a closer look at each of the three types of variables that can be found in an audit trail.

### Data Set Variables
As you might expect, the audit trail file has the same set of variables that are in the audited data set. If the data set contains the variables A and B, the variables A and B are also in the audit trail file.

Next let’s take a look at the audit trail variables that automatically store information about changes that you make to the data.

### Audit Trail Variables
Audit trail variables automatically store information about data modifications. Audit trail variable names begin with AT followed by a specific string, such as DATETIME.

<table>
<thead>
<tr>
<th>Audit trail variable</th>
<th>Information stored</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ATDATETIME</em></td>
<td>date and time of a modification</td>
</tr>
<tr>
<td><em>ATUSERID</em></td>
<td>login user ID associated with a modification</td>
</tr>
<tr>
<td><em>ATOBSNO</em></td>
<td>observation number affected by the modification unless REUSE=YES</td>
</tr>
<tr>
<td><em>ATRETURNCODE</em></td>
<td>event return code</td>
</tr>
<tr>
<td><em>ATMESSAGE</em></td>
<td>SAS log message at the time of the modification</td>
</tr>
<tr>
<td><em>ATOPCODE</em></td>
<td>code describing the type of operation</td>
</tr>
</tbody>
</table>
Values of the _ATOPCODE_ Variable

The _ATOPCODE_ variable contains a code that describes the type of operation that wrote the observation to the audit file. For example, if you modified all observations in an audited data set, the audit file would contain twice as many observations as the original data set. The audit file would contain one observation that matched the original observation with an _ATOPCODE_ value of DR, and one updated observation with an _ATOPCODE_ value of DW.

The table below shows the possible values of the _ATOPCODE_ variable.

<table>
<thead>
<tr>
<th><em>ATOPCODE</em></th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>DA</td>
<td>added data record image</td>
</tr>
<tr>
<td>DD</td>
<td>deleted data record image</td>
</tr>
<tr>
<td>DR</td>
<td>before-update record image</td>
</tr>
<tr>
<td>DW</td>
<td>after-update record image</td>
</tr>
<tr>
<td>EA</td>
<td>observation add failed</td>
</tr>
<tr>
<td>ED</td>
<td>observation delete failed</td>
</tr>
<tr>
<td>EU</td>
<td>observation update failed</td>
</tr>
</tbody>
</table>

You can define what information is stored in the audit file by using the LOG statement when you initiate the audit trail.

Using the LOG Statement to Control the Data in the Audit Trail

When you initiate an audit trail, options in the LOG statement determine the type of entries stored in the audit trail, along with their corresponding _ATOPCODE_ values. The ERROR_IMAGE option controls E operation codes. The BEFORE_IMAGE option controls the DR operation code, and the DATA_IMAGE option controls all other D operation codes. If you omit the LOG statement when you initiate the audit trail, the default behavior is to log all images.

General form, LOG statement:

\[
\text{LOG } \langle \text{audit-settings} \rangle;
\]

where

\[
\langle \text{audit-settings} \rangle
\]

are any of the following:

- \(\text{BEFORE\_IMAGE=\text{YES|NO}}\) controls storage of before-update record images (the ‘DR’ operation).
- \(\text{DATA\_IMAGE=\text{YES|NO}}\) controls storage of after-update record images (for example, other operations starting with ‘D’).
- \(\text{ERROR\_IMAGE=\text{YES|NO}}\) controls storage of unsuccessful update record images (for example, operations starting with ‘E’).
Example

The following code initiates an audit trail on the data set Capinfo but stores only error record images. This means that the audit file will contain only records where an error occurred. The _ATOPCODE_ values can be only EA, ED, and EU.

Note: If you choose to run this example, you must copy the data set Capinfo from the Sasuser library to the Work library.

proc datasets nolist;
  audit capinfo;
  initiate;
  log data_image=NO before_image=NO;
quit;

User Variables

User variables allow the person editing the file to enter information about changes they are making to the data. Although the data values are stored in the audit file, you can update them in the data set like any other variable.

User variables are created by using the USER_VAR statement in the audit trail specification.

General form, USER_VAR statement:

```
USER_VAR variable-name <$><length><LABEL='variable-label'>;
```

where

- variable-name
  - is the name of the user variable you are creating.
- $
  - indicates the variable is a character variable.
- length
  - specifies the length of the variable (the default is 8).
- variable-label
  - specifies a label for the variable enclosed in quotation marks.

Note: You can create more than one user variable in a single USER_VAR statement.

User variables are unique in SAS in that they are stored in one file (the audit file) and opened for update in another file (the data set). When the data set is opened for update, the user variables display, and you can edit them as though they are part of the data set.

Example

Suppose you must monitor the updates for the data set Capinfo. The following code initiates an audit trail for the data set Capinfo and creates two user variables, who and why, to store who made changes to the data set and why the changes were made.
Note: If you choose to run this example, you must copy the data set `Capinfo` from the `Sasuser` library to the `Work` library.

```sas
proc datasets nolist;
    audit capinfo;
    initiate;
    user_var who $20 label = 'Who made the change'
        why $20 label = 'Why the change was made';
quit;
```

Once these user variables are set up, they are retrieved from the audit trail and displayed when the data set is opened for update. You can enter data values for the user variables as you would for any data variable. The data values are saved to the audit trail as each observation is saved. The user variables are not available when the data is opened for browsing or printing. To rename a user variable or modify its attributes, you modify the data set, not the audit file.

### Controlling the Audit Trail

Once you activate an audit trail, you can suspend and resume logging, and terminate (delete) the audit trail by resubmitting a PROC DATASETS step with additional statements. You use the DATASETS procedure to suspend and then resume the audit trail. You also use this procedure to delete or terminate an audit trail.

General form, DATASETS procedure to suspend, resume, or terminate an audit trail:

```sas
PROC DATASETS LIB=libref<NO ListNode>
    AUDIT SAS-data-set <SAS-password>
    SUSPEND | RESUME | TERMINATE;
QUIT;
```

where

- `libref` is the name of the library where the table to be audited resides.
- `NO ListNode` suppresses the directory listing.
- `SAS-data-set` is the name of the SAS data set that you want to audit.
- `SAS-password` is the SAS data file password, if one exists.
- `SUSPEND` suspends event logging to the audit file, but does not delete the audit file.
- `RESUME` resumes event logging to the audit file, if it was suspended.
- `TERMINATE` terminates event logging and deletes the audit file.
**Tip:** Because each update to the data file is also written to the audit file, the audit trail can negatively impact system performance. You may want to consider suspending the audit trail for large, regularly scheduled batch updates.

---

**Example**

The following code terminates the audit trail on the data set *Capinfo*.

*Note:* If you choose to run this example, you must copy the data set *Capinfo* from the *Sasuser* library to the *Work* library.

```plaintext
proc datasets nolist;
   audit capinfo;
   terminate;
quit;
```

A message is written to the log when the audit trail is terminated.

**Table 18.5** SAS Log

<table>
<thead>
<tr>
<th>Line</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>audit capinfo;</td>
<td>Start audit trail on <em>Capinfo</em></td>
</tr>
<tr>
<td>66</td>
<td>terminate;</td>
<td>Terminate audit trail</td>
</tr>
<tr>
<td></td>
<td>NOTE: Deleting WORK.CAPINFO (memtype=AUDIT).</td>
<td>Log message</td>
</tr>
<tr>
<td>67</td>
<td>run;</td>
<td>Run program</td>
</tr>
<tr>
<td>68</td>
<td>quit;</td>
<td>End program</td>
</tr>
</tbody>
</table>

---

**Understanding Generation Data Sets**

You have learned that you can keep an audit trail to track observation updates made to an individual data set in place. However, if you replace the data set, the audit trail is lost. Generation data sets allow you to maintain multiple versions or generations of a SAS data set. A new generation is created each time the file is replaced.

By default, **generation data sets are not in effect**. As the SAS data set *A* is replaced, there are two copies of *A* in the SAS library. When the DATA step completes execution, SAS removes the original copy of the data set *A* from the library.

```plaintext
data a;
   set a;
run;
```
When generation data sets are in effect and the SAS data set A is replaced, there are two copies of A in the SAS library. When the DATA step completes execution, SAS keeps the original copy of the SAS data set A in the library and renames it.

Each generation of a generation data set is stored as part of a generation group. Each generation data set in a generation group has the same root member name, but each has a different version number. The most recent version is called the base version. When generations are in effect, SAS filenames are limited to 28 characters. The last four characters are reserved for the version numbers.

Note: Generation data sets are not supported on VMS.

Initiating Generation Data Sets

To initiate generation data sets and to specify the maximum number of versions to maintain, you use the output data set option GENMAX= when creating or replacing a data set. If the data set already exists, you can use the GENMAX= option with the DATASETS procedure and the MODIFY statement.
General form, DATASETS procedure and MODIFY statement with the GENMAX= option:

```
PROC DATASETS LIB=libref <NOLIST>;
   MODIFY SAS-data-set (GENMAX=n);
QUIT;
```

where

```
libref
   is the library that contains the data you want to modify.
NOLIST
   suppresses the directory listing.
SAS-data-set
   is the name of the SAS data set that you want to modify.
```

```
n
   is the number of historical versions you want to keep, including the base version:
     □ n=0, no historical versions are kept (this is the default).
     □ n>0, the specified number of versions of the file will be kept. The number includes the base version.
```

---

**Example**

The following DATASETS procedure modifies the data set Cargorev and requests that up to four versions be kept (one base version and three historical versions).

*Note:* If you choose to run this example, you must copy the data set Cargorev from the Sasuser library to the Work library.

```
proc datasets nolist;
   modify cargorev (genmax=4);
quit;
```

△

No message is written to the log when you specify the GENMAX= option.

---

**Creating Generation Data Sets**

Remember, new versions of a generation data set are created only when a data set is replaced, not when it is modified in place. To create new generations, use

□ a DATA step with a SET statement
□ a DATA step with a MERGE statement
□ PROC SORT without the OUT= option
□ PROC SQL with a CREATE TABLE statement.

---

**Processing Generation Data Sets**

Once you have a generation group that contains more than one generation data set, you may want to select a particular data set to process. To select a particular generation, you use the GENNUM= data set option.
General form, GENNUM= data set option:

```
GENNUM=n
```

where

- `n` specifies a particular historical version of a data set:
  - `n>0` is an absolute reference to a historical version by its generation number.
  - `n<0` is a relative reference to a historical version.
  - `n=0` is the current version.

Examples

To print the current version of the data, you do not need to use the GENNUM= option. Simply use

```
proc print data=year;
run;
```

To print the youngest historical version, you can either specify the relative or absolute reference on the GENNUM= option, as shown:

```
proc print data=year(gennum=4); /*absolute reference*/
run;
```

or

```
proc print data=year(gennum=-1); /*relative reference*/
run;
```

You can also view information on a specific generation using the GENNUM= option with PROC CONTENTS, as shown:

```
proc contents data=year(gennum=-1); /*relative reference*/
run;
```

Now that you have seen a few examples of using the GENNUM= option, let's take a look at how generation numbers change.

How Generation Numbers Change

When you use the GENNUM= option, you can refer to either the absolute or relative generation number. It's helpful to understand how generation numbers change so that you can identify the generation you want to process.

First, let's take a look at how SAS names generation data sets. The first time a data set with generations in effect is replaced, SAS keeps the replaced data set and appends a four-character version number to its member name, which includes the pound symbol (#) and a three-digit number. That is, for a data set named `A`, the replaced data set becomes `A#001`. When the data set is replaced for the second time, the replaced data set becomes `A#002`; that is, `A#002` is the version that is chronologically closest to the base version. The table below shows the result after three replacements.
Chapter 18

Data Set Name Explanation
---
A base (current) version
A#003 most recent (youngest) historical version
A#002 second most recent historical version
A#001 oldest historical version

The limit for version numbers that SAS can append is #999. After 1000 replacements, SAS rolls over the youngest version number to #000.

Now let’s take a look at how absolute and relative generation numbers (specified on the GENNUM= option) change. Each time SAS creates a new generation, the absolute generation number increases sequentially. As older generations are deleted, their absolute generation numbers are retired.

In contrast, the relative generation number always refers to generations in relation to the base generation. The base or current generation is always 0 and -1 is the youngest historical version.

The following table shows data set names and their absolute and relative GENNUM= numbers.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>SAS Code</th>
<th>Data Set Names</th>
<th>GENNUM=Absolute Reference</th>
<th>GENNUM=Relative Reference</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data Year (genmax=3);</td>
<td>Year</td>
<td>1</td>
<td>0</td>
<td>The data set Year is created, and three generations are requested.</td>
</tr>
<tr>
<td>2</td>
<td>data Year;</td>
<td>Year</td>
<td>2</td>
<td>0</td>
<td>Year is replaced. Year from iteration 1 is renamed Year#001.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year#001</td>
<td>1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>data Year;</td>
<td>Year</td>
<td>3</td>
<td>0</td>
<td>Year is replaced. Year from iteration 2 is renamed Year#002.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year#002</td>
<td>2</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year#001</td>
<td>1</td>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>
### Modifying SAS Data Sets and Tracking Changes

#### How Generation Numbers Change

<table>
<thead>
<tr>
<th>Iteration</th>
<th>SAS Code</th>
<th>Data Set Names</th>
<th>GENNUM=Absolute Reference</th>
<th>GENNUM=Relative Reference</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>data Year;</td>
<td>Year</td>
<td>4</td>
<td>0</td>
<td>Year is replaced. Year from iteration 3 is renamed Year#003. Year#001 from iteration 1, which is the oldest, is deleted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year#003</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year#002</td>
<td>2</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>data Year (genmax=2);</td>
<td>Year</td>
<td>5</td>
<td>0</td>
<td>Year is replaced, and the number of generations is changed to 2. Year from iteration 4 is renamed Year#004. The two oldest versions are deleted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year#004</td>
<td>4</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

You have learned that you use PROC DATASETS to initiate generation data sets on an existing SAS data set. Once you have created generation data sets, you can use PROC DATASETS to perform management tasks such as

- deleting all or some of the generations
- renaming an entire generation group or any member of the group to a new base name.

**General form, PROC DATASETS with the CHANGE and DELETE statements:**

```sas
PROC DATASETS LIB=libref <NOLIST>;
  CHANGE SAS-data-set<(GENNUM=n)>=new-data-set-name;
  DELETE SAS-data-set<(GENNUM=n | HIST | ALL)>;
QUIT;
```

Where

- `libref` is the library that contains the data you want to modify.
- `NOLIST` suppresses the directory listing.
- `SAS-data-set` is the name of the SAS data set you want to change or delete.
- `new-data-set-name` is the new name for the SAS data set in the `CHANGE` statement.
- `n` is the absolute or relative reference to a generation number.
- `HIST` refers to all generations except the base version.
- `ALL` refers to the base version and all generations.
**Examples**

The following code uses the CHANGE statement to rename the data set `SalesData` to `Sales`. If generations have been created, the base name of all generations will be renamed.

```sas
proc datasets library=quarter1 nolist;
  change salesData=sales;
quit;
```

The following code uses the GENNUM= option to rename only the second historical data set:

```sas
proc datasets library=quarter1 nolist;
  change sales(gennum=2)=newsales;
quit;
```

The following code deletes one historical version. This may leave a hole in the generation group.

```sas
proc datasets library=quarter1 nolist;
  delete newsales(gennum=-1);
quit;
```

When you use the GENNUM= option with the DELETE statement, you can use the HIST and ALL keywords. The following code uses the HIST keyword to delete all of the historical versions:

```sas
proc datasets library=quarter1 nolist;
  delete newsales(gennum=HIST);
quit;
```

The following code uses the ALL keyword in the GENNUM= option to delete all of the SAS data sets in a generation group:

```sas
proc datasets library=quarter1 nolist;
  delete newsales(gennum=ALL);
quit;
```

*Tip:* For more information about using the DATASETS procedure to process data, see the SAS documentation.
Summary

This section contains the following:
- a text summary of the material taught in this chapter
- syntax for statements and options
- sample programs
- points to remember.

Text Summary

Using the MODIFY Statement

When you use the MODIFY statement to modify a SAS data set, SAS does not create a second copy of the data as it does when you use the SET, MERGE, or UPDATE statements. The descriptor portion of the SAS data set stays the same and the updated observation is written to the data set in the location of the original observation.

Modifying All Observations in a SAS Data Set

You can use the MODIFY statement with an assignment statement to modify all the observations for a variable in a SAS data set.

Modifying Observations Using a Transaction Data Set

To modify a master data set using a transaction data set, you use the MODIFY statement with a BY statement to specify the variable you want to use to match. When you use the MODIFY/BY statements, SAS uses a dynamic WHERE clause to locate observations in the master data set. You can specify how missing values in the transaction data set are handled by using the UPDATEMODE= option in the MODIFY statement.

Modifying Observations Located by an Index

You can use the MODIFY statement with the KEY= option to name a simple or composite index for the SAS data set that is being modified. The KEY= argument retrieves observations from the SAS data file based on index values that are supplied by like-named variables in a transaction data set. If you have contiguous duplications in the transaction data set such that there is no match in the master data set, you can use the UNIQUE option to cause a KEY= search to always begin at the top of the index file for each duplicate transaction.

Controlling the Update Process

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. If no statement is present, SAS writes the current observation to its original place in the SAS data set. This occurs as the last action in the step as though a REPLACE statement were the last statement in the step. However, you can override this default behavior by explicitly adding the OUTPUT, REPLACE, or REMOVE statement.

You can use the automatic variable _IORC_ with the %SYSRC autocall macro to test for specific I/O error conditions that are created when you use the BY statement or the KEY= option in the MODIFY statement. The automatic variable _IORC_ contains a
Placing Integrity Constraints on a Data Set

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a variable in a SAS data file. SAS enforces integrity constraints when values associated with a variable are added, updated, or deleted. You can place integrity constraints on an existing data set using the IC CREATE statement in the DATASETS procedure.

Documenting and Removing Integrity Constraints

You can view information about the integrity constraints on a data set using the CONTENTS statement in the DATASETS procedure. If you want to remove integrity constraints from a file, you use the IC DELETE statement.

Initiating and Terminating Audit Trails

An audit trail is an optional SAS file that logs modifications to a SAS table. You initiate an audit trail using the DATASETS procedure with the AUDIT and INITIATE statements. You also suspend, resume, and terminate audit trails using the DATASETS procedure. Once there is data in the audit trail file, you can read it with the TYPE= data set option.

Controlling Data in the Audit Trail

The audit trail file can contain three types of variables:

- data set variables that store copies of the columns in the audited SAS data file
- audit trail variables that automatically store information about data modifications
- user variables that store user-entered information.

You can use the LOG statement to control which types of records are written to an audit trail file.

Initiating Generation Data Sets

Each generation of a generation data set is stored as part of a generation group. A new generation is created each time the file is replaced. Each generation in a generation group has the same root member name, but each has a different version number. You initiate generation data sets by using the GENMAX= option to specify the number of generation data sets to keep.

Processing Generation Data Sets

To select a particular generation to process, you use the GENNUM= option. GENNUM= is an input/update data set option that identifies which generation to open. The GENNUM can be a relative or absolute reference to a generation within a generation group. You can rename or delete generations using the CHANGE and DELETE statements in a PROC DATASETS step.
Syntax

Modifying All Observations in a SAS Data Set

DATA SAS-data-set;
    MODIFY SAS-data-set;
    existing-variable = expression;
RUN;

Modifying a Master Data Set Using the BY Statement

MODIFY master-data-set transaction-data-set
    <UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK>;
    BY key-variable;

Modifying a Master Data Set Using a Transaction Data Set and an Index

MODIFY master-data-set KEY=index </UNIQUE>;

Controlling the Update Process

OUTPUT;
REPLACE;
REMOVE;

Using PROC DATASETS to Create Integrity Constraints, Generation Data Sets, and Audit Trails

PROC DATASETS <LIB=libref> <NOLIST> ;
    IC CREATE <constraint-name=>constraint-type
        <MESSAGE='Error Message'>;
    IC DELETE constraint-name;
    MODIFY SAS-data-set (GENMAX=n);
    AUDIT SAS-data-file <password>;
    INITIATE;
    <LOG<audit-settings>>;
    <USER_VAR variable-name <$><length><LABEL='variable-label'>>
    SUSPEND | RESUME | TERMINATE;
    CONTENTS data=SAS-data-set;
QUIT;

Using _IORC_ with %SYSRC

IF _IORC_=%SYSRC(<mnemonic>) THEN...

Specifying an Audit Trail File

(TYPE=AUDIT)
Using PROC DATASETS to Rename or Delete Generation Data Sets

```sas
PROC DATASETS LIB=SAS-library <NOLIST>;
   CHANGE SAS-data-set<(GENNUM=n)>=new-data-set-name;
   DELETE SAS-data-set<(GENNUM=n | HIST | ALL)>;
QUIT;
```

---

Sample Programs

Modifying a Data Set Using the MODIFY Statement with a BY Statement and with the KEY= Option

```sas
data capacity;
   modify capacity sasuser.newrtnum;
   by flightid;
run;

data cargo99;
   set sasuser.newcgnum (rename =
   (capcargo = newCapCargo
cargowgt = newCargowgt
cargorev = newCargoRev));
   modify cargo99 key=flghtdte;
   capcargo = newcapcargo;
   cargowgt = newcargowgt;
   cargorev = newcargorev;
run;
```

Placing Integrity Constraints on Data

```sas
proc datasets nolist;
   modify capinfo;
   ic create PKIDInfo=primary key(routeid)
      message='You must supply a Route ID Number';
   ic create Class1=check(where=(cap1st<capbusiness
      or capbusiness=.)
      message='Cap1st must be less than CapBusiness';
quit;
```

Initiating an Audit Trail

```sas
proc datasets nolist;
   audit capinfo;
   initiate;
quit;
```

Initiating Generation Data Sets

```sas
proc datasets nolist;
   modify cargorev (genmax=4);
quit;
```
Points to Remember

- The MODIFY statement in a DATA step is used to make updates to a SAS data set in place. The descriptor portion of the SAS data set cannot be changed.
- Integrity constraints are only enforced when modifications are made to the data. If the data set is replaced, integrity constraints are lost.
- Audit trail files track changes made to data sets in place with
  - the MODIFY statement in the DATA step
  - the UPDATE, INSERT, or DELETE statement in PROC SQL.
- Generation data sets are used to track changes that are made when a data set is replaced by
  - using the SET, MERGE, or UPDATE statements in the DATA step
  - sorting data in place with PROC SORT
  - using the CREATE TABLE statement in PROC SQL.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which type of integrity constraint would you place on the variable StoreID to ensure that there are no missing values and that there are no duplicate values?
   - a) UNIQUE
   - b) CHECK
   - c) PRIMARY KEY
   - d) NOT NULL

2. Which code creates an audit trail on the SAS data set Reports.Quarter1?
   - a) proc datasets nolist;
     audit quarter1;
     initiate;
     quit;
   - b) proc datasets lib=reports nolist;
     audit initiate reports.quarter1;
     quit;
   - c) proc datasets lib=reports nolist;
     initiate audit quarter1;
     quit;
   - d) proc datasets lib=reports nolist;
     audit quarter1;
     initiate;
     quit;
3 Which DATA step uses the transaction data set `Records.Overnight` to update the master data set `Records.Snowfall` by `accumAmt`?

   a data records.snowfall;
      modify records.snowfall records.overnight
         key=accumAmt;
      run;

   b data records.snowfall;
      modify records.overnight records.snowfall;
         by accumAmt;
      run;

   c data records.snowfall;
      modify records.snowfall records.overnight;
         by accumAmt;
      run;

   d data records.snowfall;
      modify records.snowfall records.overnight;
      update accumAmt;
      run;

4 The automatic variable `_IORC_` is created when you use the MODIFY statement with a BY statement or the KEY= option. How can you use the value of `_IORC_`?

   a to determine whether the index specified on the KEY= option is a valid index
   b to determine the number of observations that were updated in the master data set
   c to determine the status of the I/O operation
   d to determine the number of observations that could not be updated in the master data set

5 Which PROC DATASETS step creates an integrity constraint named `val_age` on the data set `Survey` to ensure that values of the variable `age` are greater than or equal to 18?

   a proc datasets nolist;
      modify age;
      ic create val_age=check(where=(age>=18));
      quit;

   b proc datasets nolist;
      modify Survey;
      ic create val_age=check(age>=18);
      quit;

   c proc datasets nolist;
      modify survey;
      integrity constraint
         val_age=check(where=(age>=18));
      quit;

   d proc datasets nolist;
      modify survey;
      ic create val_age=check(where=(age>=18));
      quit;
6 Which statement about using the MODIFY statement in a DATA step is true?
   a MODIFY creates a second copy of the data while variables in the data are being matched with a WHERE clause and then deletes the second copy.
   b You cannot modify the descriptor portion of the data set using the MODIFY statement.
   c You can use the MODIFY statement to change the name of a variable.
   d If the system terminates abnormally while a DATA step that is using the WHERE statement is processing, SAS automatically saves a copy of the unaltered data set.

7 Which of the following statements about audit trails is true?
   a They create historical versions of data so that a copy of the data set is saved each time the data is replaced.
   b They record information about changes to observations in a data set each time the data set is replaced.
   c They record information about changes to observations in a data set each time the data is modified in place.
   d The audit trail file has the same name as the SAS data file it is monitoring, but has #AUDIT at the end of the data set name.

8 Which code initiates generation data sets on the existing SAS data set Sasuser:Amounts and specifies that five historical versions are saved in addition to the base version?
   a proc datasets lib=sasuser nolist;
      modify Amounts (genmax=6);
      quit;
   b proc datasets lib=sasuser nolist;
      modify Amounts (genmax=5);
      quit;
   c proc datasets lib=sasuser nolist;
      modify Amounts (gennum=6);
      quit;
   d proc datasets lib=sasuser nolist;
      modify Amounts (gennum=5);
      quit;

9 Which statement about using the KEY= option in the MODIFY statement is true?
   a SAS locates the variables to update using the index specified in the KEY= option and then automatically overlays nonmissing transaction values as it does when you use the MODIFY/BY statements.
   b When you use the KEY= option, you must explicitly state the update that you want to make. SAS does not automatically overlay nonmissing transaction values.
   c The KEY= option is used to specify a variable to match for updating observations.
   d The index named in the KEY= option must be a simple index.
10 Which code deletes all generations of the data set *Sasuser.Amounts* including the base data set?

a) proc datasets lib=sasuser nolist;
   delete amounts (gennum=ALL);
   quit;

b) proc datasets lib=sasuser nolist;
   delete amounts (gennum=HIST);
   quit;

c) proc datasets lib=sasuser nolist;
   delete amounts (gennum=0);
   quit;

d) proc datasets lib=sasuser nolist;
   delete amounts;
   quit;
Optimizing SAS Programs

Chapter 19. Introduction to Efficient SAS Programming  679
Chapter 20. Controlling Memory Usage  689
Chapter 21. Controlling Data Storage Space  707
Chapter 22. Utilizing Best Practices  741
Chapter 23. Selecting Efficient Sorting Strategies  785
Chapter 24. Querying Data Efficiently  833
Overview

Introduction

As an experienced programmer, you want your SAS programs to obtain the desired results while minimizing the use of resources such as CPU time, real time, memory, and I/O.

It is particularly important to optimize your SAS programs if you write or maintain production programs and work with large data sets. However, before you can select the most efficient programming technique to perform a particular task, you must carefully consider the technical environment and the resource constraints at your site. There is no single set of programming techniques that is most efficient in all situations. Instead, trade-offs in resource usage are associated with each technique.
In this chapter you will learn about analyzing the requirements for efficiency at your site and about running benchmarks to select the most efficient SAS programming techniques.

Note: This chapter has no quiz.

Objectives

In this chapter, you learn to

- identify the resources that are used by a SAS program
- identify the main factors to consider when you assess the efficiency needs at your site
- identify common efficiency trade-offs
- select the appropriate SAS system option(s) to track and report the resource usage statistics that you want in your operating environment
- interpret resource usage statistics that are displayed in the SAS log in the z/OS, UNIX, and Windows operating environments
- identify the guidelines that you should follow in order to benchmark effectively.

Overview of Computing Resources

Running a SAS program requires the programmer and the computer to perform a variety of tasks. The programmer must write, submit, and maintain the program. The computer must load the required SAS software components and the program into memory, compile the program, locate the data that the program requires, and execute the program.

The following resources are used to run a SAS program:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>programmer time</td>
<td>the amount of time required for the programmer to write and maintain the program. Programmer time is difficult to quantify, but it can be decreased through well-documented, logical programming practices and the use of reusable SAS code modules.</td>
</tr>
<tr>
<td>CPU time</td>
<td>the amount of time the central processing unit (CPU) uses to perform requested tasks such as calculations, reading and writing data, conditional logic, and iterative logic.</td>
</tr>
<tr>
<td>real time</td>
<td>the clock time (elapsed time) it takes to execute a job or step. Real time is heavily dependent on the capacity of the system and on the load (the number of users that are sharing the system’s resources). Because you cannot always control the capacity and the load demands on your system, real time is sometimes a less useful measure of program efficiency than CPU time. However, excessive use of real time often motivates programmers to improve a program’s efficiency. Some procedures in SAS 9.1 give you the option of using threaded processing to reduce real time. Threaded processing can cause an increase in CPU time, so it is recommended that you track both CPU time and real time.</td>
</tr>
</tbody>
</table>
Resource Description

memory the size of the work area in volatile memory that is required for holding executable program modules, data, and buffers.

data storage space the amount of space on a disk or tape that is required for storing data. Data storage space is measured in a variety of units, some of which are used only in certain operating environments, as described below:
  □ All operating environments use bytes, kilobytes, megabytes, gigabytes, and terabytes.
  □ z/OS also uses blocks, tracks, and cylinders.

I/O a measurement of the read and write operations that are performed as data and programs are copied from a storage device to memory (input) or from memory to a storage or display device (output).

Assessing Efficiency Needs at Your Site

The first step in making an effective decision about how to optimize your SAS programs is to assess your site’s technical environment, your program(s), and your data.

Assessing Your Technical Environment

To determine which resources are scarce or costly at your site, work with your IT department to analyze the following characteristics of your technical environment:

<table>
<thead>
<tr>
<th>Category</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardware</td>
<td>□ amount of available memory</td>
</tr>
<tr>
<td></td>
<td>□ number of CPUs</td>
</tr>
<tr>
<td></td>
<td>□ number and type of peripheral devices</td>
</tr>
<tr>
<td></td>
<td>□ communications hardware</td>
</tr>
<tr>
<td></td>
<td>□ network bandwidth</td>
</tr>
<tr>
<td></td>
<td>□ storage capacity</td>
</tr>
<tr>
<td></td>
<td>□ I/O bandwidth</td>
</tr>
<tr>
<td></td>
<td>□ capacity to upgrade</td>
</tr>
<tr>
<td>operating environment</td>
<td>□ resource allocation</td>
</tr>
<tr>
<td></td>
<td>□ scheduling algorithms</td>
</tr>
<tr>
<td></td>
<td>□ I/O methods</td>
</tr>
</tbody>
</table>
In most cases, one or two resources are the most limited or most expensive for your programs. You can usually decrease the amount of critical resources that are used if you are willing to sacrifice some efficiency of the resources that are less critical at your site.

### Assessing Your Programs

Developing an efficient program requires time and thought. To determine whether the additional amount of resources saved is worth the time and effort spent to achieve the savings, consider the following characteristics of each of your programs:

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Guidelines for Optimizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of the program</td>
<td>As the program increases in size, the potential for savings increases. Focus on improving the efficiency of large programs.</td>
</tr>
<tr>
<td>number of times the program will run</td>
<td>The difference in resources used by an inefficient program and an efficient program that is run once or a few times is relatively small, whereas the cumulative difference for a program that is run frequently is large. Focus on improving the efficiency of programs that are run many times.</td>
</tr>
</tbody>
</table>

### Assessing Your Data

The effectiveness of any efficiency technique depends greatly on the data with which you use it. When you know the characteristics of your data, you can select the techniques that take advantage of those characteristics. Consider the following characteristics of your data:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Guidelines for Optimizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume of data</td>
<td>As the volume of data increases, the potential for savings also increases. Focus on improving the efficiency of programs that use large data sets or many data sets.</td>
</tr>
<tr>
<td>type of data</td>
<td>Specific efficiency techniques might work better with some types of data (for example, data that has missing values) than with others.</td>
</tr>
</tbody>
</table>
Understanding Efficiency Trade-offs

When you are trying to optimize SAS programs, it is important to understand that there are trade-offs. Decreasing the use of one resource frequently increases the use of another. The following table shows examples of some common efficiency trade-offs.

<table>
<thead>
<tr>
<th>Decreasing usage of this resource ...</th>
<th>Might increase usage of this resource ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>disk space</td>
<td>CPU time</td>
</tr>
<tr>
<td>I/O (by reading or writing more data at one time)</td>
<td>memory</td>
</tr>
<tr>
<td>real time (by enabling threading in SAS 9.1)</td>
<td>CPU time</td>
</tr>
<tr>
<td>any computer resource (by increasing program complexity)</td>
<td>programmer time</td>
</tr>
</tbody>
</table>

As these trade-offs illustrate, there is no single best way to optimize a SAS program; it depends on your situation. However, this chapter and the following chapters provide information that can help you determine which programming techniques are most efficient in your environment:

- Chapter 20, “Controlling Memory Usage,” on page 689
- Chapter 21, “Controlling Data Storage Space,” on page 707
- Chapter 22, “Utilizing Best Practices,” on page 741
- Chapter 23, “Selecting Efficient Sorting Strategies,” on page 785
- Chapter 24, “Querying Data Efficiently,” on page 833.

Using SAS System Options to Track Resources

You can specify one or more of the SAS system options STIMER, MEMRPT, FULLSTIMER, and STATS to track and report on resource utilization. The availability, usage, and functionality of these options vary by operating environment, as described below:
### Option Network Options to Track Resources

<table>
<thead>
<tr>
<th>Option</th>
<th>z/OS</th>
<th>UNIX and Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>STIMER</td>
<td>Specifies that the CPU time is to be tracked throughout the SAS session. Can be set at invocation only. Is the default setting.</td>
<td>Specifies that CPU time and real-time statistics are to be tracked and written to the SAS log throughout the SAS session. Can be set either at invocation or by using an OPTIONS statement. Is the default setting.</td>
</tr>
<tr>
<td>MEMRPT</td>
<td>Specifies that memory usage statistics are to be tracked throughout the SAS session. Can be set either at invocation or by using an OPTIONS statement. Is the default setting.</td>
<td>Not available as a separate option; this functionality is part of the FULLSTIMER option.</td>
</tr>
<tr>
<td>FULLSTIMER</td>
<td>Specifies that all available resource usage statistics are to be tracked and written to the SAS log throughout the SAS session. Can be set either at invocation or by using an OPTIONS statement. In the z/OS operating environment, FULLSTIMER is an alias for the FULLSTATS option. This option is ignored unless STIMER or MEMRPT is in effect.</td>
<td>Specifies that all available resource usage statistics are to be tracked and written to the SAS log throughout the SAS session. Can be set either at invocation or by using an OPTIONS statement. In Windows operating environments, some statistics will not be calculated accurately unless FULLSTIMER is specified at invocation.</td>
</tr>
<tr>
<td>STATS</td>
<td>Tells SAS to write statistics that are tracked by any combination of the preceding options to the SAS log. Can be set either at invocation or by using an OPTIONS statement. Is the default setting.</td>
<td>Not available as a separate option.</td>
</tr>
</tbody>
</table>

SAS system options are initialized with default settings when SAS is invoked. However, the default settings for some SAS system options vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

You can turn off any of these system options by using the options below:

- NOSTIMER
- NOMEMRPT
- NOFULLSTIMER
- NOSTATS.

*Note:* In the z/OS operating environment, NOFULLSTIMER is an alias for the NOFULLSTATS option. △
Viewing Resource Statistics

To see examples of the resource usage statistics that are written to the SAS log in each operating environment, select the links below:

- z/OS
- UNIX
- Windows.

Note: For more information about using these system options to track resource usage in your operating environment, see the SAS documentation for your operating environment. Guidelines for interpreting the statistics that are generated by the FULLSTIMER SAS option are also available at support.sas.com.

Note: You can also use SAS Application Response Measurement (ARM) macros to monitor the performance of your applications. ARM macros are not covered in this course. To learn more about ARM macros, see the SAS documentation and detailed information about ARM macros at support.sas.com.

Using Benchmarks to Compare Techniques

To decide which SAS programming technique is most efficient for a particular task, you can benchmark (measure and compare) the resource usage for each technique that you are comparing.

Guidelines for Benchmarking

Your benchmarking is most likely to yield useful results if you follow these guidelines:

- Before you test the programming techniques, turn on the SAS system options that report resource usage.
  
  As explained earlier, to track and report on resource utilization, you can use some or all of the system options STIMER, MEMRPT, FULLSTIMER, and STATS. The availability, usage, and functionality of these options vary by operating environment. You can also specify MSGLEVEL=I to display additional notes in the SAS log.

  Note: For more information about the SAS system option MSGLEVEL=, see Chapter 13, “Creating Samples and Indexes,” on page 451 or Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221.

- Execute the code for each programming technique in a separate SAS session.
  
  If the program that you are benchmarking is not large, you can optimize it by changing individual programming techniques, one at a time, and running the entire program after each change to measure the effect on resource usage. However, a more complex program might be easier to optimize by identifying the steps that use the most resources and extracting those steps into separate programs. You can measure the effects of different programming techniques by repeatedly changing, running, and measuring the separate programs. When isolating parts of your program, be careful to measure their resource usage under the conditions in which they are used in the complete program.

  Note: The first time that program code (including the DATA step, functions and formats, and SAS procedures) is referenced, the operating system might have
to load the code into memory or assign virtual address space to it. The first time 
data is read, it is often loaded into a cache from which it can be retrieved more 
quickly the next time it is read. The resource usage that is required for performing 
these actions is called overhead.

- **In each programming technique that you are testing, include only the SAS code 
  that is essential for performing the task.**

  If you include too many elements in the code for each technique, you will not 
  know what caused the results.

- **If your system is doing other work at the same time you are running your 
  benchmarking tests, be sure to run the code for each programming technique 
  several times.**

  Running the code several times reduces any variability in resource consumption 
  that is associated with other work that the system is doing. How you handle 
multiple measurements depends on the resource, as indicated below:

  - Use the minimum real time and CPU time measurements, because these 
    represent most closely the amount of time your programming technique 
    actually requires. The larger time values (especially in the case of real time) 
    are the result of interference from other work that the computer was doing 
    while your program ran.

  - The amount of memory shouldn’t vary from trial to trial. If memory does 
    vary, it’s possible that your program sometimes shares a resource with 
    another program. In this situation, you must determine whether the higher 
    or lower memory consumption is more likely to be the case when your 
    program is used in production.

  - I/O can be an especially elusive resource to measure. With modern file 
    systems and storage systems, the effect of your program on the I/O activity of 
    the computer sometimes has to be observed by operating system tools, file 
    system tools, or storage system tools because it cannot be captured by your 
    SAS session. Data is often aggressively cached by modern file systems and 
    storage systems, and file caches are greatly affected by other activity in the 
    file system. Be as realistic as you can about measuring I/O—it is possible to 
    achieve good performance on a system that is not doing other work, but 
    performance is likely to worsen when the application is deployed in a more 
    realistic environment.

  - **Run your benchmarking tests under the conditions in which your final program 
    will run.**

    Results might vary under different conditions, so it is important to control the 
    conditions under which your benchmarks are tested. For example, if batch 
    execution and large data sets are used in your environment, you should 
    incorporate these conditions into your benchmarking environment.

  - **After testing is finished, consider turning off the options that report resource usage.**

    The options that report resource usage consume resources. If it is a higher 
    priority in your environment to minimize resource usage than to periodically check 
    an application’s resource usage, then it is most efficient to turn off these options.

    **Note:** To turn off the FULLSTIMER option, use the following statement:

    ```
    options nofullstimer;
    ```
Summary

Overview of Computing Resources

Resources that are required for running a SAS program include the following: programmer time, CPU time, real time, memory, data storage space, and I/O.

Assessing Efficiency Needs at Your Site

To make an effective decision about how to optimize your SAS programs, work with your IT department to assess the following factors at your site: the technical environment, your individual SAS programs, and the data.

Understanding Efficiency Trade-offs

It is important to understand the trade-offs that are involved in optimizing your SAS programs. Decreasing the use of one resource frequently increases the use of another. There is no single best way to optimize a SAS program; it depends on your situation.

Using SAS System Options to Track Resources

You can specify one or more of the SAS system options STIMER, MEMRPT, FULLSTIMER, and STATS to track and report on resource utilization. (In the z/OS environment, FULLSTIMER is an alias for FULLSTATS.) The availability, usage, and functionality of these options varies by operating environment.

Using Benchmarks to Compare Techniques

To determine which SAS programming technique is most efficient for a particular task, you can benchmark (measure and compare) the resource usage of each technique.
Overview 689

Introduction 689
Objectives 690
Prerequisites 690

Controlling Page Size and the Number of Buffers 690

Measuring I/O 690
Page Size 691
Reporting Page Size 692
Using the BUFSIZE= Option 693
Using the BUFNO= Option 694
Comparative Example: Using the BUFSIZE= Option and the BUFNO= Option 696

Programming Techniques 696
General Recommendations 697

Using the SASFILE Statement 698
Guidelines for Using the SASFILE Statement 699
Comparative Example: Using the SASFILE Statement 699

Programming Techniques 700
General Recommendations 702

Additional Features 703
Using the IBUFSIZE= System Option 703

Summary 704

CONTROLLING PAGE SIZE AND THE NUMBER OF BUFFERS 704
Using the SASFILE Statement 704
Additional Features 704

Quiz 705

Overview

Introduction

As you have learned, there is no single set of programming techniques that is most efficient or appropriate in all situations. However, if reducing execution time is an important consideration in your computing environment, one way of achieving that goal is to reduce the number of times SAS has to read from or write to the storage medium.

In this chapter you learn to use options and a statement to control the size and number of data buffers, which in turn can affect your programs’ execution times by reducing the number of I/O operations that SAS must perform.
Note: This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SAS SPD Engine to improve performance, see the SAS documentation.

Objectives

In this chapter, you learn to
- control the amount of data that is loaded into memory with each I/O transfer
- reduce I/O by holding a SAS data file in memory through multiple steps of a program.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

Part 1: SQL Processing with SAS
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
- Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125
- Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159
- Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221
- Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243
- Chapter 8, “Managing Processing Using PROC SQL,” on page 261

Part 3: Advanced SAS Programming Techniques
- Chapter 13, “Creating Samples and Indexes,” on page 451
- Chapter 14, “Combining Data Vertically,” on page 481
- Chapter 15, “Combining Data Horizontally,” on page 513
- Chapter 16, “Using Lookup Tables to Match Data,” on page 559
- Chapter 17, “Formatting Data,” on page 603
- Chapter 18, “Modifying SAS Data Sets and Tracking Changes,” on page 633

Part 4: Optimizing SAS Programs

Controlling Page Size and the Number of Buffers

Measuring I/O

Improvement in I/O can come at the cost of increased memory consumption. In order to understand the relationship between I/O and memory, it is helpful to know when
Controlling Memory Usage

Data is copied to a buffer and where I/O is measured. When you create a SAS data set using a DATA step,

1. SAS copies the data from the input data set to a buffer in memory
2. one observation at a time is loaded into the program data vector
3. each observation is written to an output buffer when processing is complete
4. the contents of the output buffer are written to the disk when the buffer is full.

The process for reading external files is similar. However, each record is first read into the input buffer before the data is parsed and read into the program data vector.

In both cases, I/O is measured when the input data is copied to the buffer in memory and when it is read from the output buffer to the output data set.

**Page Size**

Think of a buffer as a container in memory that is big enough for only one page of data. A *page*

- is the unit of data transfer between the storage device and memory
includes the number of bytes used by the descriptor portion, the data values, and any overhead.

- is fixed in size when the data set is created, either to a default value or to a user-specified value.

The amount of data that can be transferred to one buffer in a single I/O operation is referred to as page size. Page size is analogous to buffer size for SAS data sets.

A larger page size can reduce execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

**Reporting Page Size**

You can use the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure to report the page size and the number of pages.

```
proc contents data=company.order_fact;
run;
```

<table>
<thead>
<tr>
<th>Engine/Host Dependent Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Set Page Size: 8192 bytes</td>
</tr>
<tr>
<td>Number of Data Set Pages: 9423</td>
</tr>
<tr>
<td>First Data Page: 1</td>
</tr>
<tr>
<td>Max Obs per Page: 101</td>
</tr>
<tr>
<td>Obs in First Data Page: 70</td>
</tr>
<tr>
<td>Number of Data Set Repairs: 0</td>
</tr>
<tr>
<td>File Name: c:\order_fact.sas7dat</td>
</tr>
<tr>
<td>Release Created: 9.310180</td>
</tr>
<tr>
<td>Host Created: XP PRO</td>
</tr>
</tbody>
</table>

The total number of bytes that a data file occupies equals the page size multiplied by the number of pages. For example, the page size for Company.Order_fact is 8192 and the number of pages is 9423. Therefore, the data file occupies 77,193,216 bytes.

*Note:* Note that the information that is available from PROC CONTENTS depends on the operating environment.

*Note:* Page size is analogous to buffer size for SAS data sets.
Note: In uncompressed data files, there is a 16-byte overhead at the beginning of each page and a 1-bit per observation overhead (rounded up to the nearest byte), used to denote an observation’s status as deleted or not deleted, at the end of each page.

You can learn about the structure of uncompressed and compressed data files in Chapter 21, “Controlling Data Storage Space,” on page 707.

Using the BUFSIZE= Option

To select a default page size, SAS uses an algorithm that is based on observation length, engine, and operating environment. The default page size is optimal for most SAS activities, especially on computers that are supporting multiple SAS jobs concurrently. However, in some cases, choosing a page/buffer size that is larger than the default can speed up execution time by reducing the number of times that SAS must read from or write to the storage medium.

You can use the BUFSIZE= system option or data set option to control the page size of an output SAS data set. BUFSIZE= specifies not only the page size (in bytes), but also the size of each buffer that is used for reading or writing the SAS data set. The new buffer size is a permanent attribute of the data set; after it is specified, it is used whenever the data set is processed.

General form, BUFSIZE= option:

```
BUFSIZE= MIN | MAX | n;
```

where

- **MIN** sets the page size to the smallest possible number in your operating environment.
- **MAX** sets the page size to the maximum possible number in your operating environment.
- **n** specifies the page size in bytes. For example, a value of 8 specifies a page size of 8 bytes, and a value of 4K specifies a page size of 4096 bytes. The default is 0, which causes SAS to use the optimal page size for the operating environment.

**CAUTION:**

MIN might cause unexpected results and should be avoided. Use BUFSIZE=0 to reset the buffer page size to the default value in your operating environment.

Note: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

Only certain page/buffer size values are valid for each operating environment. If you request an invalid value for your operating environment, SAS automatically rounds up to the next valid page/buffer size. BUFSIZE=0 is interpreted as a request for the default page/buffer size.

In the following program, the BUFSIZE= system option specifies a page size of 30720 bytes.

```sql
options bufsize=30720;
filename orders 'c:\orders.dat';
data company.orders_fact;
```
infile orders;
    <more SAS code>
run;

Before you change the default page size, it is important to consider the access pattern for the data as well as the I/O transfer rate of the underlying hardware. In some cases, increasing the page size might degrade performance, particularly when the data is processed using direct (random) access.

Note: The default value for BUFSIZE= is determined by your operating environment and is set to optimize sequential access. To improve performance for direct access, you should change the value for BUFSIZE=. For the default setting and possible settings for direct access, see the BUFSIZE= system option in the SAS documentation for your operating environment.

Note: You can override the BUFSIZE= system option by using the BUFSIZE= data set option.

CAUTION: If you use the COPY procedure to copy a data set to a library that is accessed via a different engine, the original page/buffer size is not necessarily retained.

---

Using the BUFNO= Option

You can use the BUFNO= system or data set option to control the number of buffers that are available for reading or writing a SAS data set. By increasing the number of buffers, you can control how many pages of data are loaded into memory with each I/O transfer.

Note: Increasing the number of buffers might not affect performance under the Windows and UNIX operating environments, especially when you work with large data sets. By default, the Windows and UNIX operating environments read one buffer at a time. Under the SAS 9 Windows environment, you can override this default by turning on the SGIO system option when you invoke SAS. For details on the SGIO system option, see the SAS documentation for the Windows operating environment.

The following techniques might help to minimize I/O consumption:

- When you work with a small data set, allocate as many buffers as there are pages in the data set so that the entire data set can be loaded into memory. This technique is most effective if you read the same observations several times during processing.
- Under the z/OS operating environment, increase the number of buffers allocated, rather than the size of each buffer, as the size of the data set increases.
General form, BUFNO= option:

```
BUFNO= MIN | MAX | n;
```

where

- **MIN**
  sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default.

- **MAX**
  sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is $2^{31}-1$, or approximately 2 billion.

- **n**
  specifies the number of buffers to be allocated.

**CAUTION:**

The recommended maximum for this option is 10.

Note: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

In the following program, the BUFNO= system option specifies that 4 buffers are available.

```sas
options bufno=4;
filename orders 'c:\orders.dat';
data company.orders_fact;	infile orders;
<more SAS code>
run;
proc print data=company.orders_fact;
run;
```

The buffer number is not a permanent attribute of the data set and is valid only for the current step or SAS session.

**Figure 20.1** Current SAS Session

Note: You can override the BUFNO= system option by using the BUFNO= data set option.

Note: In SAS 9 and later, the BUFNO= option has no effect on thread-enabled procedures under the z/OS operating environment.

The product of BUFNO= and BUFSIZE=, rather than the specific value of either option, determines how much data can be transferred in one I/O operation. Increasing
the value of either option increases the amount of data that can be transferred in one I/O operation.

<table>
<thead>
<tr>
<th>BUFSIZE</th>
<th>BUFNO</th>
<th>Bytes Transferred In One I/O Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6144</td>
<td>2</td>
<td>12,288</td>
</tr>
<tr>
<td>6144</td>
<td>10</td>
<td>61,440</td>
</tr>
<tr>
<td>30,720</td>
<td>2</td>
<td>61,440</td>
</tr>
<tr>
<td>30,720</td>
<td>10</td>
<td>307,200</td>
</tr>
</tbody>
</table>

The number of buffers and the buffer size have a minimal effect on CPU usage.

**Comparative Example: Using the BUFSIZE= Option and the BUFNO= Option**

Suppose you want to compare the resource usage when a data set is read using different buffer sizes and a varying number of buffers. The following sample programs use the following settings for the BUFNO= option and the BUFSIZE= option.

1. BUFSIZE=6144, BUFNO=2
2. BUFSIZE=6144, BUFNO=5
3. BUFSIZE=6144, BUFNO=10
4. BUFSIZE=12288, BUFNO=2
5. BUFSIZE=12288, BUFNO=5
6. BUFSIZE=12288, BUFNO=10

You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for controlling page size and the number of buffers.

*Note:* 6144 bytes is the default page size under the z/OS operating environment.

**Programming Techniques**

```plaintext
1. BUFSIZE=6144, BUFNO=2

This program reads the data set Retail.Order_fact and creates the data set Work.Orders. The BUFSIZE= option specifies that Work.Orders is created with a buffer size of 6144 bytes. The BUFNO= option specifies that 2 pages of data are loaded into memory with each I/O transfer.

```data work.orders (bufsize=6144 bufno=2);
  set retail.order_fact;
run;```
Comparative Example: Using the BUFSIZE= Option and the BUFNO= Option

1. **BUFSIZE=6144, BUFNO=5**
   - This program reads the data set `Retail.Order_fact` and creates the data set `Work.Orders`. The BUFSIZE= option specifies that `Work.Orders` is created with a buffer size of 6144 bytes. The BUFNO= option specifies that 5 pages of data are loaded into memory with each I/O transfer.
   ```
   data work.orders (bufsize=6144 bufno=5);
     set retail.order_fact;
   run;
   ```

2. **BUFSIZE=6144, BUFNO=10**
   - This program reads the data set `Retail.Order_fact` and creates the data set `Work.Orders`. The BUFSIZE= option specifies that `Work.Orders` is created with a buffer size of 6144 bytes. The BUFNO= option specifies that 10 pages of data are loaded into memory with each I/O transfer.
   ```
   data work.orders (bufsize=6144 bufno=10);
     set retail.order_fact;
   run;
   ```

3. **BUFSIZE=12288, BUFNO=2**
   - This program reads the data set `Retail.Order_fact` and creates the data set `Work.Orders`. The BUFSIZE= option specifies that `Work.Orders` is created with a buffer size of 12288 bytes. The BUFNO= option specifies that 2 pages of data are loaded into memory with each I/O transfer.
   ```
   data work.orders (bufsize=12288 bufno=2);
     set retail.order_fact;
   run;
   ```

4. **BUFSIZE=12288, BUFNO=5**
   - This program reads the data set `Retail.Order_fact` and creates the data set `Work.Orders`. The BUFSIZE= option specifies that `Work.Orders` is created with a buffer size of 12288 bytes. The BUFNO= option specifies that 5 pages of data are loaded into memory with each I/O transfer.
   ```
   data work.orders (bufsize=12288 bufno=5);
     set retail.order_fact;
   run;
   ```

5. **BUFSIZE=12288, BUFNO=10**
   - This program reads the data set `Retail.Order_fact` and creates the data set `Work.Orders`. The BUFSIZE= option specifies that `Work.Orders` is created with a buffer size of 12288 bytes. The BUFNO= option specifies that 10 pages of data are loaded into memory with each I/O transfer.
   ```
   data work.orders (bufsize=12288 bufno=10);
     set retail.order_fact;
   run;
   ```

**General Recommendations**

- To reduce I/O operations on a small data set, allocate as many buffers as there are pages in the data set so that the entire data set can be loaded into memory. This
technique is most effective if you read the same observations several times during processing.

- Under the z/OS operating environment, as the size of the data set increases, increase the number of buffers allocated, rather than the size of each buffer, to minimize I/O consumption.

## Using the SASFILE Statement

Another way of improving performance is to use the SASFILE statement to hold a SAS data file in memory so that the data is available to multiple program steps. Keeping the data file open reduces open/close operations, including the allocation and freeing of memory for buffers.

### General form, SASFILE statement:

```
SASFILE SAS-data-file <(password-option(s))> OPEN | LOAD | CLOSE;
```

where

- **SAS-data-file**
  - is a valid SAS data file (a SAS data set with the member type DATA).

- **password-option(s)**
  - specifies one or more password options.

- **OPEN**
  - opens the file and allocates the buffers, but defers reading the data into memory until a procedure or statement is executed.

- **LOAD**
  - opens the file, allocates the buffers, and reads the data into memory.

- **CLOSE**
  - closes the file and frees the buffers.

The SASFILE statement opens a SAS data file and allocates enough buffers to hold the entire file in memory. Once the data file is read, the data is held in memory, and it is available to subsequent DATA and PROC steps or applications until either

- a SASFILE CLOSE statement frees the buffers and closes the file
- the program ends, which automatically frees the buffers and closes the file.

In the following program, the SASFILE statement opens the SAS data file `Company.Sales`, allocates the buffers, and reads the data into memory.

```sas
sasfile company.sales load;
proc print data=company.sales;
   var Customer_Age_Group;
run;
proc tabulate data=company.sales;
   class Customer_Age_Group;
   var Customer_BirthDate;
   table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
sasfile company.sales close;
```
Controlling Memory Usage

Note: The SASFILE statement can also be used to reduce CPU time and I/O in SAS programs that repeatedly read one or more SAS data views. Use a DATA step to create a SAS data file in the Work library that contains the view’s result set. Then use the SASFILE statement to load that data file into memory.

Note: Though a file that is opened with the SASFILE statement can be used for subsequent input or update processing, it cannot be used for subsequent utility or output processing. For example, you cannot replace the file or rename its variables.

Guidelines for Using the SASFILE Statement

When the SASFILE statement executes, SAS allocates the number of buffers based on the number of pages for the data file and index file. If the file in memory increases in size during processing because of changes or additions to the data, the number of buffers also increases.

It is important to note that I/O processing is reduced only if there is sufficient real memory. If there is not sufficient real memory, the operating environment might

- use virtual memory
- use the default number of buffers.

If SAS uses virtual memory, there might be a degradation in performance.

If you need to repeatedly process part of a SAS data file and the entire file won’t fit into memory, use a DATA step with the SASFILE statement to create a subset of the file that does fit into memory, and then process that subset repeatedly. This saves CPU time in the processing steps because those steps will read a smaller file, in addition to the benefit of the file being resident in memory.

Note: When using a SASFILE statement, monitor the paging activity (the I/O activity that is done by the virtual memory management subsystem of your operating environment) while your program runs. If the paging activity increases substantially, consider keeping less data in memory and using techniques described elsewhere in this course to reduce memory requirements.

Comparative Example: Using the SASFILE Statement

Suppose you want to create multiple reports from SAS data files that vary in size. Using small, medium, and large data files, you can compare the resource usage when the PRINT, TABULATE, MEANS, and FREQ procedures are used with and without the SASFILE statement to create reports.

<table>
<thead>
<tr>
<th>Name of Data File</th>
<th>Number of Rows</th>
<th>Page Size</th>
<th>Number of Pages</th>
<th>Number of Byes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retail.Small</td>
<td>45,876</td>
<td>24,576</td>
<td>540</td>
<td>13,279,232</td>
</tr>
<tr>
<td>Retail.Medium</td>
<td>458,765</td>
<td>24,576</td>
<td>5,398</td>
<td>132,669,440</td>
</tr>
<tr>
<td>Retail.Large</td>
<td>4,587,654</td>
<td>24,576</td>
<td>53,973</td>
<td>1,326,448,640</td>
</tr>
</tbody>
</table>

1 Small Data File without the SASFILE Statement
2 Medium Data File without the SASFILE Statement
3 Large Data File without the SASFILE Statement
4 Small Data File with the SASFILE Statement
5 Medium Data File with the SASFILE Statement

6 Large Data File with the SASFILE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using the SASFILE statement.

**Programming Techniques**

1. **Small Data File without the SASFILE Statement**

   This program creates reports using the PRINT, TABULATE, MEANS, and FREQ procedures. The SAS data file `Retail.Small` is opened and closed with each procedure.

   ```sas
   proc print data=retail.small;
     where cs=100;
     var Customer_Age_Group;
   run;
   proc tabulate data=retail.small;
     class Customer_Age_Group;
     var Customer_BirthDate;
     table Customer_Age_Group,Customer_BirthDate*(mean median);
   run;
   proc means data=retail.small;
     var Customer_Age;
     class Customer_Group;
     output out=summary sum=;
   run;
   proc freq data=retail.small;
     tables Customer_Country;
   run;
   ```

2. **Medium Data File without the SASFILE Statement**

   This program creates reports using the PRINT, TABULATE, MEANS, and FREQ procedures. The SAS data file `Retail.Medium` is opened and closed with each procedure.

   ```sas
   proc print data=retail.medium;
     where cm=100;
     var Customer_Age_Group;
   run;
   proc tabulate data=retail.medium;
     class Customer_Age_Group;
     var Customer_BirthDate;
     table Customer_Age_Group,Customer_BirthDate*(mean median);
   run;
   proc means data=retail.medium;
     var Customer_Age;
     class Customer_Group;
     output out=summary sum=;
   run;
   proc freq data=retail.medium;
     tables Customer_Country;
   run;
   ```
Large Data File without the SASFILE Statement

This program creates reports using the PRINT, TABULATE, MEANS, and FREQ procedures. The SAS data file Retail.Large is opened and closed with each procedure.

```sas
proc print data=retail.large;
   where cl=100;
   var Customer_Age_Group;
run;
proc tabulate data=retail.large;
   class Customer_Age_Group;
   var Customer_BirthDate;
   table Customer_Age_Group,Customer_BirthDate*(mean median); run;
proc means data=retail.large;
   var Customer_Age;
   class Customer_Group;
   output out=summary sum=; run;
proc freq data=retail.large;
   tables Customer_Country; run;
```

Small Data File with the SASFILE Statement

In this program, the SASFILE LOAD statement opens the SAS data file Retail.Small and loads the entire file into memory. The data is then available to the PRINT, TABULATE, MEANS, and FREQ procedures. The SASFILE CLOSE statement closes Retail.Small and frees the buffers.

```sas
sasfile retail.small load;
proc print data=retail.small;
   where cs=100;
   var Customer_Age_Group;
run;
proc tabulate data=retail.small;
   class Customer_Age_Group;
   var Customer_BirthDate;
   table Customer_Age_Group,Customer_BirthDate*(mean median); run;
proc means data=retail.small;
   var Customer_Age;
   class Customer_Group;
   output out=summary sum=; run;
proc freq data=retail.small;
   tables Customer_Country; run;
sasfile retail.small close;
```
Medium Data File with the SASFILE Statement

In this program, the SASFILE LOAD statement opens the SAS data file *Retail.Medium* and loads the entire file into memory. The data is then available to the PRINT, TABULATE, MEANS, and FREQ procedures. The SASFILE CLOSE statement closes *Retail.Medium* and frees the buffers.

```
sasfile retail.medium load;
proc print data=retail.medium;
   where cm=100;
   var Customer_Age_Group;
run;
proc tabulate data=retail.medium;
   class Customer_Age_Group;
   var Customer_BirthDate;
   table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.medium;
   var Customer_Age;
   class Customer_Group;
   output out=summary sum=;
run;
proc freq data=retail.medium;
   tables Customer_Country;
run;
sasfile retail.medium close;
```

Large Data File with the SASFILE Statement

In this program, the SASFILE LOAD statement opens the SAS data file *Retail.Large* and loads the entire file into memory. The data is then available to the PRINT, TABULATE, MEANS, and FREQ procedures. The SASFILE CLOSE statement closes *Retail.Large* and frees the buffers.

```
sasfile retail.large load;
proc print data=retail.large;
   where cl=100;
   var Customer_Age_Group;
run;
proc tabulate data=retail.large;
   class Customer_Age_Group;
   var Customer_BirthDate;
   table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.large;
   var Customer_Age;
   class Customer_Group;
   output out=summary sum=;
run;
proc freq data=retail.large;
   tables Customer_Country;
run;
sasfile retail.large close;
```

General Recommendations

- If you need to repeatedly process a SAS data file that will fit entirely in memory, use the SASFILE statement to reduce I/O and some CPU usage.
- If you use the SASFILE statement and the SAS data file will not fit entirely in memory, the code will execute, but there might be a degradation in performance.
If you need to repeatedly process part of a SAS data file and the entire file won’t fit into memory, use a DATA step with the SASFILE statement to create a subset of the file that does fit into memory, and then process that subset repeatedly. This saves CPU time in the processing steps because those steps will read a smaller file, in addition to the benefit of the file being resident in memory.

Additional Features

Using the IBUFSIZE= System Option

Beginning with SAS 9, you can use the IBUFSIZE= system option to specify the page size for an index file. Typically, you do not need to specify an index page size. However, you might need to use the IBUFSIZE= option if

- there are many levels in the index
- the length of an index value is very large.

The main resource that is saved when reducing levels in the index is I/O. If your application is experiencing a lot of I/O in the index file, increasing the page size might help. However, you must re-create the index file after increasing the page size. The number of pages that are required for the index varies with the page size, the length of the index value, and the values themselves.

General form, IBUFSIZE= system option:

```
IBUFSIZE= MIN | MAX | n;
```

where

- **MIN**
  sets the page size for an index file to −32767. The IBUFSIZE= option is defined as a signed integer so that negative values can be supplied for internal testing purposes. This might cause unexpected results.

- **MAX**
  sets the page size for an index file to the maximum possible number. For IBUFSIZE=, the maximum value is 32,767 bytes.

- **n**
  specifies the page size in bytes.

**Note:** The MIN setting should be avoided.

When an index is used to process a request, such as for WHERE processing, SAS searches the index file in order to rapidly locate the requested record(s). The page size affects the number of levels in the index. The more pages there are, the more levels in the index. The more levels, the longer the index search takes. Increasing the page size allows more index values to be stored on each page, thus reducing the number of pages (and the number of levels).

Use IBUFSIZE=0 to reset the index page size to the default value in your operating environment.

**Note:** For details on using the IBUFSIZE= system option, see the SAS documentation.
Chapter 20

Summary

Controlling Page Size and the Number of Buffers

When you read a SAS data set or an external file, I/O is measured when the input data is copied to the buffer in memory and when it is read from the output buffer to the output data set.

A page is the unit of data transfer between the storage device and memory. When you create a SAS data set, SAS takes the data and copies it to a buffer. Each buffer can hold one page of data.

The amount of data that can be transferred to one buffer in a single I/O operation is referred to as the page size. Increasing the page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. You can use the CONTENTS procedure to report the page size and the number of pages.

You can use the BUFSIZE= system option or data set option to control the page size of an output SAS data set. The new buffer size is permanent; after it is specified, it is used whenever the data set is processed.

You can use the BUFNO= system or data set option to control how many buffers are available for reading or writing a SAS data set. By increasing the number of buffers, you can control how many pages of data are loaded into memory with each I/O transfer.

The product of BUFNO= and BUFSIZE=, rather than the specific value of either option, determines how much data can be transferred in one I/O operation. Increasing either option increases the amount of data that can be transferred in one I/O operation. However, the improvement in I/O comes at the cost of increased memory consumption.

Review the related comparative example:

- “Comparative Example: Using the BUFSIZE= Option and the BUFNO= Option” on page 696.

Using the SASFILE Statement

Another way of improving performance is to use the SASFILE statement to hold a SAS data file in memory so that the data is available to multiple program steps. Keeping the data set open reduces open/close operations, including the allocation and freeing of memory for buffers.

When the SASFILE statement executes, SAS allocates the number of buffers based on the number of pages for the data file and index file. If the file in memory increases in size during processing because of changes or additions to the data, the number of buffers also increases.

It is important to note that I/O processing is reduced only if there is sufficient real memory. If SAS uses virtual memory, there can be a degradation in performance.

Review the related comparative example:

- “Comparative Example: Using the SASFILE Statement” on page 699.

Additional Features

The IBUFSIZE= system option specifies the page size for an index file. Typically, you do not need to specify an index page size. However, you might need to use the IBUFSIZE= option if

- there are many levels in the index
- the length of an index value is very large.

The main resource that is saved when reducing levels in the index is I/O. If your application is experiencing a lot of I/O in the index file, increasing the page size might
help. However, you must re-create the index file after increasing the page size. The number of pages that are required for the index varies with the page size, the length of the index value, and the values themselves.

**Quiz**

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which of the following statements is true regarding the BUFNO= option?
   a. The BUFNO= option specifies the size of each buffer that is used for reading or writing a SAS data set.
   b. The BUFNO= option can improve execution time by limiting the number of input/output operations that are required.
   c. Using the BUFNO= option results in permanent changes to the data set.
   d. Using the BUFNO= option to increase the number of buffers results in decreased memory consumption.

2. Which of the following statements is not true regarding a page?
   a. A page is the unit of data transfer between the engine and the operating environment.
   b. A page includes the number of bytes used by the descriptor portion, the data values, and the overhead.
   c. The size of a page is analogous to buffer size.
   d. The size of a page can be changed at any time.

3. The total number of bytes occupied by a data set equals...?
   a. the page size multiplied by the number of pages.
   b. the page size multiplied by the number of observations.
   c. the sum of the page size and the number of pages.
   d. the number of pages multiplied by the number of variables.

4. Which statement opens the file *Work.Quarter1*, allocates enough buffers to hold the entire file in memory, and reads the data into memory?
   a. `sasfile work.quarter1 open;`
   b. `sasfile work.quarter1 load;`
   c. `sasfile work.quarter1 bufno=max;`
   d. `sasfile work.quarter1 bufsize=max;`

5. Which of the following statements is true regarding a file that is opened with the SASFILE statement?
   a. The file is available to subsequent DATA and PROC steps or applications until a SASFILE CLOSE statement is executed or until the program ends.
   b. The file is available to subsequent DATA and PROC steps or applications until a SASFILE END statement is executed.
   c. The file is available for subsequent utility or output processing until the program ends.
   d. If the file increases in size during processing, the number of buffers remains the same.
CHAPTER 21

Controlling Data Storage Space

Overview 708
Introduction 708
Objectives 709
Prerequisites 709
Reducing Data Storage Space for Character Variables 709
   How SAS Assigns Lengths to Character Variables 710
   Reducing the Length of Character Data with the LENGTH Statement 710
   Other Techniques 710
Reducing Data Storage Space for Numeric Variables 711
   How SAS Stores Numeric Variables 711
   Assigning Lengths to Numeric Variables 712
   Example 713
   Maintaining Precision in Reduced-Length Numeric Variables 713
   Using PROC COMPARE 714
   Example 715
   Comparative Example: Creating a SAS Data Set That Contains Reduced-Length Numeric Variables 716
      Programming Techniques 717
      General Recommendations 718
Compressing Data Files 719
   Review of Uncompressed Data File Structure 719
   Compressed Data File Structure 719
   Deciding Whether to Compress a Data File 720
   The COMPRESS= System Option and the COMPRESS= Data Set Option 721
      Example 723
   Accessing Observations Directly in a Compressed Data Set 723
      The POINTOBS= Data Set Option 724
      Example 724
   The REUSE= System Option and the REUSE= Data Set Option 724
      Example 726
   How SAS Compresses Data 726
   Comparative Example: Creating and Reading Compressed Data Files 727
      Programming Techniques 727
      General Recommendations 730
Using SAS DATA Step Views to Conserve Data Storage Space 730
   DATA Step Views 731
      Example 732
   The DESCRIBE Statement 732
   Creating and Referencing a SAS DATA Step View 733
   Referencing a Data View Multiple Times in One Program 734
      Example 734
Overview

Introduction

In many computing environments, data storage space is a limited resource. Therefore, it might be more important for you to conserve data storage space than to conserve other resources.

When you store your data in a SAS data file, you use the sum of the data storage space that is required for the following:

- the descriptor portion
- the observations
- any storage overhead
- any associated indexes.

In this chapter you learn to use a variety of techniques for minimizing the amount of space that your SAS data files occupy.

Note: This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SAS SPD Engine to improve performance, see the SAS documentation.

△
Objectives

In this chapter, you learn to

- describe how SAS stores character variables
- determine how to reduce the length of character variables and how to expand the values
- describe how SAS stores numeric variables
- determine how to safely reduce the space that is required for storing numeric variables in SAS data sets
- define the structure of a compressed SAS data file
- create a compressed SAS data file
- examine the advantages and disadvantages of compression
- describe the difference between a SAS data file and a SAS data view
- examine the advantages and disadvantages of DATA step views.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

Part 1: SQL Processing with SAS
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
- Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125
- Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159
- Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221
- Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243
- Chapter 8, “Managing Processing Using PROC SQL,” on page 261.

Part 3: Advanced SAS Programming Techniques
- Chapter 13, “Creating Samples and Indexes,” on page 451
- Chapter 14, “Combining Data Vertically,” on page 481
- Chapter 15, “Combining Data Horizontally,” on page 513
- Chapter 16, “Using Lookup Tables to Match Data,” on page 559
- Chapter 17, “Formatting Data,” on page 603

Part 4: Optimizing SAS Programs
- Chapter 19, “Introduction to Efficient SAS Programming,” on page 679
- Chapter 20, “Controlling Memory Usage,” on page 689.

Reducing Data Storage Space for Character Variables

One way to reduce the amount of data storage space that you need is to reduce the length of character data, thereby eliminating wasted space. Before discussing how to reduce the length of a character variable, let’s look at how SAS assigns lengths to character variables.
How SAS Assigns Lengths to Character Variables

SAS character variables store data as 1 character per byte. A SAS character variable can be from 1 to 32,767 bytes in length.

The first reference to a variable in the DATA step defines it in the program data vector and in the descriptor portion of the data set. Unless otherwise defined, the first value that is specified for a SAS character variable determines the variable's length. For example, if the length of a character variable called Name has not been defined and if the first value that is specified for it is Smith, the length of Name is set to 5. Then, if the next value specified for Name is Johnson, the value is stored as Johns in the data set. Similarly, if the first value specified for Town is Williamsburg, the length of Town is set to 12. If the next value for Town is specified as Cary, the length is still 12, and the value is padded with blanks to fill the extra space.

Keep in mind that the first reference to a variable might not be an assignment statement. If SAS cannot determine a length for a character variable when the variable is created in the program data vector, SAS assigns a default length of 8 bytes to the variable.

Reducing the Length of Character Data with the LENGTH Statement

You can use a LENGTH statement to reduce the length of character variables. It is useful to reduce the length of a character variable with a LENGTH statement when you have a large data set that contains many character variables.

General form, LENGTH statement for character variables:

```
LENGTH variable(s) $ length;
```

where

- `variable(s)` specifies the name of one or more SAS variables, separated by spaces.
- `length` is an integer from 1 to 32,767 that specifies the length of the variable(s).

Note: Make sure the LENGTH statement appears before any other reference to the variable in the DATA step. If the variable has been created by another statement, then a later use of the LENGTH statement will not change its size.

Other Techniques

There are other techniques that you can use to reduce the length of your character data—especially if your data set has repeated values.

For example, suppose you have a data set that records employee names along with the employees’ departments. Instead of recording the complete department name in the data set, you could assign a code to each department and record the code in your data set instead. You could record the complete department name along with the corresponding code in a separate data set, where you would have to record each full-length value only once. Then you could use a table lookup operation to convert the code to the department name for reporting purposes. This is called normalizing the data.
Controlling Data Storage Space

The tables below represent two data sets. *Employees* records the employee names and department codes for all employees. *DeptCodes* records the department name and department code, and is shown in its entirety. If the *Employees* data set contains several hundred observations, then using department codes instead of the complete department names can save a substantial amount of data storage space.

<table>
<thead>
<tr>
<th>Employees (partial listing)</th>
<th>DeptCodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LastName</strong></td>
<td><strong>FirstName</strong></td>
</tr>
<tr>
<td>Mills</td>
<td>Dorothy E.</td>
</tr>
<tr>
<td>Bower</td>
<td>Eileen A.</td>
</tr>
<tr>
<td>Reading</td>
<td>Tony R.</td>
</tr>
<tr>
<td>Judd</td>
<td>Carol A.</td>
</tr>
<tr>
<td>Wonsild</td>
<td>Hanna</td>
</tr>
<tr>
<td>Anderson</td>
<td>Christopher</td>
</tr>
<tr>
<td>Massengill</td>
<td>Annette W.</td>
</tr>
<tr>
<td>Badine</td>
<td>David</td>
</tr>
</tbody>
</table>

*Note:* You can learn about table lookup operations in Chapter 15, “Combining Data Horizontally,” on page 513.

Reducing Data Storage Space for Numeric Variables

Another way to eliminate wasted space and thereby to reduce the amount of data storage space that you need is to reduce the length of numeric variables. In addition to conserving data storage space, reduced-length numeric variables use less I/O, both when data is written and when it is read. For a file that is read frequently, this savings can be significant. However, in order to safely reduce the length of numeric variables, you need to understand how SAS stores numeric data.

How SAS Stores Numeric Variables

To store numbers of large magnitude and to perform computations that require many digits of precision to the right of the decimal point, SAS stores all numeric values using double-precision floating-point representation. SAS stores the value of a numeric variable as multiple digits per byte. A SAS numeric variable can be from 2 to 8 bytes or 3 to 8 bytes in length, depending on your operating environment. The default length for a numeric variable is 8 bytes.

The figures below show how SAS stores a numeric value in 8 bytes. For mainframe environments, the first bit stores the sign, the next seven bits store the exponent of the value, and the remaining 56 bits store the mantissa.
For non-mainframe environments, the first bit stores the sign, the next eleven bits store the exponent of the value, and the remaining 52 bits store the mantissa.

Note: The minimum length for a numeric variable is 2 bytes in mainframe environments and 3 bytes in non-mainframe environments.

Now that you have seen how SAS stores numeric variables, let’s look at how you can assign a length to your numeric variables that is less than the default length of 8 bytes.

**Assigning Lengths to Numeric Variables**

You can use a `LENGTH statement` to assign a length from 2 to 8 bytes to numeric variables. Remember, the minimum length of numeric variables depends on the operating environment. Also, keep in mind that the LENGTH statement affects the length of a numeric variable only in the output data set. Numeric variables always have a length of 8 bytes in the program data vector and during processing.

General form, LENGTH statement for numeric variables:

```
LENGTH variable(s) length <DEFAULT=n>;
```

where

- `variable(s)` specifies the name of one or more numeric SAS variables, separated by spaces.
- `length` is an integer that specifies the length of the variable(s).
- `DEFAULT=n` this optional argument changes the default number of bytes that SAS uses to store the values of any newly created numeric variables. If you use the `DEFAULT= n` argument, you do not need to list any `variable(s)`.

Note: Values between 2 and 8 or between 3 and 8 depending on your operating environment are valid for `n` or `length`.

`DEFAULT=` applies only to numeric variables that are added to the program data vector after the LENGTH statement is compiled. You would list specific variables in the LENGTH statement along with the `DEFAULT=` argument only if you wanted those variables to have a length other than the value for `DEFAULT=`. If you list individual variables in the LENGTH statement, you must list an integer length for each of them.

**CAUTION:**

You should assign reduced lengths to numeric variables only if those variables have integer values. Fractional numbers lose precision if truncated. You will learn more about the loss of precision with reduced-length numeric variables on the next section of this chapter.
Example

The following program assigns a length of 4 to the new variable `Sale_Percent` in the data set `ReducedSales`. The LENGTH statement in this DATA step does not apply to the variables that are read in from the `Sales` data set; those variables will maintain whatever length they had in `Sales` when they are read into `ReducedSales`.

```sas
data reducedsales;
  length default=4;
  set sales;
  Sale_Percent=15;
run;
```

Maintaining Precision in Reduced-Length Numeric Variables

There is a limit to the values that you can precisely store in a reduced-length numeric variable. You have learned that reducing the number of bytes that are used for storing a numeric variable does not affect how the numbers are stored in the program data vector. Instead, specifying a value of less than 8 in the LENGTH statement causes the number to be truncated to the specified length when the value is written to the SAS data set.

You should never use the LENGTH statement to reduce the length of your numeric variables if the values are not integers. Fractional numbers lose precision if truncated. Even if the values are integers, you should keep in mind that reducing the length of a numeric variable limits the integer values that can accurately be stored as a value.

The following table lists the possible storage length for integer values on UNIX or Windows operating environments.

Table 21.1  UNIX/Windows

<table>
<thead>
<tr>
<th>Length (bytes)</th>
<th>Largest Integer Represented Exactly</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8,192</td>
</tr>
<tr>
<td>4</td>
<td>2,097,152</td>
</tr>
<tr>
<td>5</td>
<td>536,870,912</td>
</tr>
<tr>
<td>6</td>
<td>137,438,953,472</td>
</tr>
<tr>
<td>7</td>
<td>35,184,372,088,832</td>
</tr>
<tr>
<td>8</td>
<td>9,007,199,254,740,992</td>
</tr>
</tbody>
</table>

The following table lists the possible storage length for integer values on the z/OS operating environment.

Table 21.2  z/OS

<table>
<thead>
<tr>
<th>Length (bytes)</th>
<th>Largest Integer Represented Exactly</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>3</td>
<td>65,536</td>
</tr>
<tr>
<td>4</td>
<td>16,777,216</td>
</tr>
<tr>
<td>5</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>6</td>
<td>1,099,511,627,776</td>
</tr>
</tbody>
</table>
When you store an integer that is equal to or less than the number listed above as the largest integer that can be represented exactly in a reduced-length variable, SAS truncates bytes that contain only zeros. If the integer that is stored in a reduced-length variable is larger than the recommended limit, SAS truncates bytes that contain numbers other than zero, and the integer value is changed. Similarly, you should not reduce the stored size of non-integer data because it can result in a loss of precision due to the truncation of nonzero bytes.

If you decide to reduce the length of your numeric variables, you might want to verify that you have not lost any precision in your values. Let's look at one way to do this.

Using PROC COMPARE

You can use PROC COMPARE to gauge the precision of the values that are stored in a shortened numeric variable by comparing the original variable with the shortened variable. The COMPARE procedure compares the contents of two SAS data sets, selected variables in different data sets, or variables within the same data set.

<table>
<thead>
<tr>
<th>Length (bytes)</th>
<th>Largest Integer Represented Exactly</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2,814,749,467,106,560</td>
</tr>
<tr>
<td>8</td>
<td>72,057,594,037,927,936</td>
</tr>
</tbody>
</table>

PROC COMPARE is a good technique to use for gauging the loss of precision in shortened numeric variables because it shows you whether there are differences in the stored numeric values even if these differences do not show up once the numeric variables have been formatted. PROC COMPARE looks at the two data sets and compares their

- data set attributes
- variables
- variable attributes for matching variables
- observations
- values in matching variables.

Output from the COMPARE procedure includes

- a data set summary
- a variables summary
- a listing of common variables that have different attributes
- an observation summary
- a values comparison summary
Example

The data set *Company.Discount* contains data about sale dates and discounts for certain retail products. There are 35 observations in *Company.Discount*, which is described below.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product_ID</td>
<td>num</td>
<td>8</td>
<td>product ID number</td>
</tr>
<tr>
<td>Start_Date</td>
<td>num</td>
<td>4</td>
<td>start date of sale</td>
</tr>
<tr>
<td>End_Date</td>
<td>num</td>
<td>5</td>
<td>end date of sale</td>
</tr>
<tr>
<td>Unit_Sales_Price</td>
<td>num</td>
<td>8</td>
<td>discounted sales price per unit</td>
</tr>
<tr>
<td>Discount</td>
<td>num</td>
<td>8</td>
<td>discount as percent of normal sales price</td>
</tr>
</tbody>
</table>

Suppose you shorten the length of the numeric variable *Discount*. The DATA step below creates a new data set named *Company.Discount_Short*, whose only difference from *Company.Discount* is that the length of the variable *Discount* is 4 instead of 8.

```sas
data company.discount_short;
  length Discount 4;
  set Company.Discount;
run;
```

You can use PROC COMPARE to evaluate whether shortening the length of *Discount* affects the precision of its values by comparing *Company.Discount* to *Company.Discount_Short*.

```sas
proc compare base=company.discount
  compare=company.discount_short;
run;
```

If you were to print these two data sets (*Company.Discount* and *Company.Discount_Short*), the values might appear to be identical. However, there are differences in the values as they are stored that are not apparent in the formatted output.

In the partial output below, you can see that shortening the length of *Discount* results in a loss of precision in its values; the values for *Discount* in *Company.Discount_Short* differ by a maximum of 1.9073E-07. The value comparison results show that although the values for *Discount* in the first five observations appear as 70% in both data sets, the precise (unformatted) values differ by −1.907E-7.
Comparative Example: Creating a SAS Data Set That Contains Reduced-Length Numeric Variables

Suppose you want to create a SAS data set in which to store retail data about a group of orders. Suppose that the data you want to include in your data set is all numeric data and that it is currently stored in a raw data file. You can create the data set using

1 Default-Length Numeric Variables
2 Reduced-Length Numeric Variables

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating
Controlling Data Storage Space  △  Creating a SAS Data Set That Contains Reduced-Length Numeric Variables

environment, and the resources that are available at your site. You can also view general recommendations for creating reduced-length numeric variables.

Note: Throughout this book, the keyword _NULL_ is often used as the data set name in sample programs. Using _NULL_ causes SAS to execute the DATA step as if it were creating a new data set. However, no observations or variables are written to an output data set. Using _NULL_ when benchmarking enables you to determine what resources are used to read a SAS data set. △

Programming Techniques

Default-Length Numeric Variables

This program reads the external data file that is referenced by the fileref flat1 and creates a new data set called Retail.Longnums that contains 12 numeric variables. Each of the variables in Retail.Longnums has the default storage length of 8 bytes. The second DATA step in this program reads the numeric variables from Retail.Longnums.

```sas
data retail.longnums;
    infile flat1;
    input Customer_ID 12.
    Employee_ID 12.
    Street_ID 12.
    Order_Date date9.
    Delivery_Date date9.
    Order_ID 12.
    Order_Type comma16.
    Product_ID 12.
    Quantity 4.
    Total_Retail_Price dollar13.2
    CostPrice_Per_Unit dollar13.2
    Discount 5.2 ;
run;

data _null_;  
    set retail.longnums;
run;
```
Reduced-Length Numeric Variables

This program reads the external data file that is referenced by the fileref flat1 and creates a new SAS data set called Retail.Shortnums that contains 12 numeric variables. A LENGTH statement is used to reduce the storage length of most of the numeric variables in Retail.Shortnums, as follows:

- **Total_Retail_Price** and **CostPrice_Per_Unit** have a storage length of 8 bytes.
- **Product_ID** has a storage length of 7 bytes.
- **Street_ID** and **Order_ID** have a storage length of 6 bytes.
- **Employee_ID** has a storage length of 5 bytes.
- **Customer_ID, Order_Date, Delivery_Date** and **Discount** have a storage length of 4 bytes.
- **Order_Type** and **Quantity** have a storage length of 3 bytes.

The second DATA step reads the reduced-length numeric variables from Retail.Shortnums.

data retail.shortnums;
  infile flat1;
  length Quantity 3;
  input Customer_ID 12.
  Employee_ID 12.
  Street_ID 12.
  Order_Date date9.
  Delivery_Date date9.
  Order_ID 12.
  Order_Type comma16.
  Product_ID 12.
  Quantity 4.
  Total_Retail_Price dollar13.2
  CostPrice_Per_Unit dollar13.2
  Discount 5.2;
run;

data _null_;  
  set retail.shortnums;  
run;

Note: Remember that when you reduce the storage length of numeric variables, you risk losing precision in their values. You can use PROC COMPARE to verify the precision of shortened numeric variables.

proc compare data=retail.longnums;
  compare=retail.shortnums;
run;

General Recommendations

Create reduced-length numeric variables for integer values when you need to conserve data storage space.
Compressing Data Files

Compressing your data files is another method that you can use to conserve data storage space. Compressing a data file is a process that reduces the number of bytes that are required in order to represent each observation in a data file.

Reading from or writing to a compressed file during data processing requires fewer I/O operations because there are fewer data set pages in a compressed data file. However, in order to read a compressed file, each observation must be uncompressed. This requires more CPU resources than reading an uncompressed file. Also, in some cases, compressing a file might actually increase its size rather than decreasing it.

Before examining how to compress a data file, let’s review the structure of an uncompressed data file as compared to the structure of a compressed data file.

Review of Uncompressed Data File Structure

By default, a SAS data file is not compressed. In uncompressed data files,

- each data value of a particular variable occupies the same number of bytes as any other data value of that variable.
- each observation occupies the same number of bytes as any other observation.
- character values are padded with blanks.
- numeric values are padded with binary zeros.
- there is a 16-byte overhead at the beginning of each page.
- there is a 1-bit per observation overhead (rounded up to the nearest byte) at the end of each page; this bit denotes an observation’s status as deleted or not deleted.
- new observations are added at the end of the file. If a new observation won’t fit on the current last page of the file, a whole new data set page is added.
- the descriptor portion of the data file is stored at the end of the first page of the file.

The figure below depicts the structure of an uncompressed data file.

<table>
<thead>
<tr>
<th>Page 1</th>
<th>16 byte OH</th>
<th>Obs 1</th>
<th>Obs 2</th>
<th>Obs 3</th>
<th>Unused Space</th>
<th>1 bit per Obs OH</th>
<th>Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 2</td>
<td>16 byte OH</td>
<td>Obs 4</td>
<td>Obs 5</td>
<td>Obs 6</td>
<td>Obs 7</td>
<td>Unused Space</td>
<td>1 bit per Obs OH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page n</td>
<td>16 byte OH</td>
<td>Obs γ</td>
<td>Obs ζ</td>
<td>Unused Space</td>
<td>1 bit per Obs OH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In comparison, let’s look at the characteristics of a compressed data file.

Compressed Data File Structure

Compressed data files
Deciding Whether to Compress a Data File

Not all data files are good candidates for compression. Remember that in order for SAS to read a compressed file, each observation must be uncompressed. This requires more CPU resources than reading an uncompressed file. However, compression can be beneficial when the data file has one or more of the following properties:

- It is large.
- It contains many long character values.
- It contains many values that have repeated characters or binary zeros.
- It contains many missing values.
- It contains repeated values in variables that are physically stored next to one another.

In character data, the most frequently encountered repeated value is the blank. Long text fields, such as comments and addresses, often contain repeated blanks. Likewise, binary zeros are used to pad numeric values that can be stored in fewer bytes than are
available in a particular numeric variable. This happens most often when you assign a small or medium-sized integer to an 8-byte numeric variable.

*Note:* If saving disk space is crucial, consider storing missing data as a small integer, such as 0 or 9, rather than as a SAS missing value. Small integers can be compressed more than SAS missing values can.

A data file is *not* a good candidate for compression if it has
- few repeated characters
- small physical size
- few missing values
- short text strings.

Next, let's look at how to compress a data file.

### The COMPRESS= System Option and the COMPRESS= Data Set Option

To compress a data file, you use either the COMPRESS= data set option or the COMPRESS= system option. You use the **COMPRESS= system option** to compress all data files that you create during a SAS session. Similarly, you use the **COMPRESS= data set option** to compress an individual data file.

**General form, COMPRESS= system option:**

```
OPTIONS COMPRESS= NO | YES | CHAR | BINARY;
```

where

- **NO**
  - is the default setting, which does not compress the data set.

- **CHAR** or **YES**
  - uses the Run Length Encoding (RLE) compression algorithm, which compresses repeating consecutive bytes such as trailing blanks or repeated zeros.

- **BINARY**
  - uses Ross Data Compression (RDC), which combines run-length encoding and sliding-window compression.

**CAUTION:**

If you set the COMPRESS= system option to a value other than **NO**, SAS compresses every data set that is created during the current SAS session, including temporary data sets in the *Work* library. Although this might conserve data storage space, it will also use greater amounts of other resources.
The COMPRESS= System Option and the COMPRESS= Data Set Option

Chapter 21

General form, COMPRESS= data set option:

```
DATA SAS-data-set (COMPRESS= NO | YES | CHAR | BINARY);
```

where

- **SAS-data-set** specifies the data set that you want to compress.
- **NO** is the default setting, which does not compress the data set.
- **CHAR** or **YES** uses the Run Length Encoding (RLE) compression algorithm, which compresses repeating consecutive bytes such as trailing blanks or repeated zeros.
- **BINARY** uses Ross Data Compression (RDC), which combines run-length encoding and sliding-window compression.

**Note:** The COMPRESS= data set option overrides the COMPRESS= system option.

The **YES** or **CHAR** setting for the COMPRESS= option uses the RLE compression algorithm. RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations. Therefore, RLE is most often useful for character data that contains repeated blanks. The **YES** or **CHAR** setting is also good for compressing numeric data in which most of the values are zero.

The **BINARY** setting for the COMPRESS= option uses RDC, which combines run-length encoding and sliding-window compression. This method is highly effective for compressing medium to large blocks of binary data (numeric variables).

A file that has been compressed using the **BINARY** setting of the COMPRESS= option takes significantly more CPU time to uncompress than a file that was compressed with the **YES** or **CHAR** setting. **BINARY** is more efficient with observations that are several hundred bytes or more in length. **BINARY** can also be very effective with character data that contains patterns rather than simple repetitions.

When you create a compressed data file, SAS compares the size of the compressed file to the size of the uncompressed file of the same page size and record count. Then SAS writes a note to the log indicating the percentage of reduction that is obtained by compressing the file.

When you use either of the COMPRESS= options, SAS calculates the size of the overhead that is introduced by compression as well as the maximum size of an observation in the data set that you are attempting to compress. If the maximum size of the observation is smaller than the overhead that is introduced by compression, SAS disables compression, creates an uncompressed data set, and issues a warning message stating that the file was not compressed. This feature is available beginning in SAS 8.2.

Once a file is compressed, the setting is a permanent attribute of the file. In order to change the setting to uncompressed, you must re-create the file.

**CAUTION:**

Compression of observations is not supported by all SAS engines. See the SAS documentation for the COMPRESS= data set option for more information.
Example

The data set `Company.Customer` contains demographic information about a retail company’s customers. The data set includes character variables such as `Customer_Name`, `Customer_FirstName`, `Customer_LastName`, and `Customer_Address`. These character variables have the potential to contain many repeated blanks in their values. The following program will create a compressed data set named `Company.Customers_Compressed` from `Company.Customer` even if the COMPRESS= system option is set to NO.

```sas
data company.customer_compressed (compress=char);
  set company.customer;
run;
```

SAS writes a note to the SAS log about the compression of the new data set, as shown below.

<table>
<thead>
<tr>
<th>Table 21.3</th>
<th>SAS Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTE:</td>
<td>There were 89954 observations read from the data set COMPANY.CUSTOMER.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The data set COMPANY.CUSTOMER_COMPRESSED has 89954 observations and 11</td>
</tr>
<tr>
<td></td>
<td>variables.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>Compressing data set COMPANY.CUSTOMER_COMPRESSED decreased size by</td>
</tr>
<tr>
<td></td>
<td>32.81 percent. Compressed is 991 pages; un-compressed would require</td>
</tr>
<tr>
<td></td>
<td>1475 pages.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>DATA statement used (Total process time):</td>
</tr>
<tr>
<td></td>
<td>real time 3.90 seconds</td>
</tr>
<tr>
<td></td>
<td>cpu time 0.96 seconds</td>
</tr>
</tbody>
</table>

Now that you have seen how to create a compressed data set, let’s look at working with compressed data sets. In general, you use a compressed data set in your programs in the same way that you would use an uncompressed data set. However, there are two options that relate specifically to compressed data sets.

Accessing Observations Directly in a Compressed Data Set

By default, the DATA step processes observations in a SAS data set sequentially. However, sometimes you might want to access observations directly rather than sequentially because doing so can conserve resources such as CPU time, I/O, and real time. You can use the POINT= option in the MODIFY or SET statement to access observations directly rather than sequentially. You can review information about the POINT= option in Chapter 13, “Creating Samples and Indexes,” on page 451. You can also use the FSEDIT procedure to access observations directly.

Allowing direct access to observations in a compressed data set increases the CPU time that is required for creating or updating the data set. You can set an option that does not allow direct access for compressed data sets. If it is not important for you to be able to point directly to an observation by number within a compressed data set, it is a good idea to disallow direct access in order to improve the efficiency of creating and updating the data set. Let’s look at how to disallow direct access to observations in a compressed data set.
The **POINTOBS=** Data Set Option

When you are working with compressed data sets, you use the **POINTOBS=** data set option to control whether observations can be processed with direct access (by observation number) rather than with sequential access only. This option is available beginning in SAS 8.

**General form, POINTOBS= data set option:**

```
DATA SAS-data-set (POINTOBS= YES | NO);
```

where

- **SAS-data-set** specifies the data set that you want to compress.
- **YES** is the default setting, which allows random access to the data set.
- **NO** does not allow random access to the data set.

**Note:** In order to use the **POINTOBS=** data set option, the **COMPRESS=** option must have a value of **YES, CHAR,** or **BINARY** for the **SAS-data set** that is specified.

Allowing random access to a data set does not affect the efficiency of retrieving information from a data set, but it does increase the CPU usage by approximately 10% when you create or update a compressed data set. That is, allowing random access reduces the efficiency of writing to a compressed data set but does not affect the efficiency of reading from a compressed data set. Therefore, if you do not need to access data by observation number, then by specifying **POINTOBS=NO**, you can improve performance by approximately 10% when creating a compressed data set and when updating or adding observations to it.

**Example**

The following program creates a data set named `Company.Customer_Compressed` from the `Company.Customer` data set and ensures that random access to the compressed data set is not allowed.

```
data company.customer_compressed (compress=yes pointobs=no);
  set company.customer;
run;
```

Now let’s look at an option that enables you to further reduce the data storage space that is required for your compressed data sets.

**The **REUSE=** System Option and the **REUSE=** Data Set Option**

In a compressed data set, SAS appends new observations to the end of the data set by default. If you delete an observation within the data set, empty disk space remains in its place. However, it is possible to track and reuse free space within the data set when you delete or update observations. By reusing space within a data set, you can conserve data storage space.
The REUSE= system option and the REUSE= data set option specify whether or not SAS reuses space when observations are added to a compressed data set. If you set the REUSE= data set option to YES in a DATA statement, SAS tracks and reuses space in the compressed data set that is created in that DATA step. If you set the REUSE= system option to YES, SAS tracks and reuses free space in all compressed data sets that are created for the remainder of the current SAS session.

General form, REUSE= system option:

\begin{verbatim}
OPTIONS REUSE= NO | YES;
\end{verbatim}

where

NO
  is the default setting, which specifies that SAS does not track unused space in the compressed data set.

YES
  specifies that SAS tracks free space and reuses it whenever observations are added to an existing compressed data set.

General form, REUSE= data set option:

\begin{verbatim}
DATA SAS-data-set (COMPRESS=YES REUSE=NO | YES);
\end{verbatim}

where

SAS-data-set
  specifies the data set that you want to compress.

NO
  is the default setting, which specifies that SAS does not track unused space in the compressed data set.

YES
  specifies that SAS tracks free space and reuses it whenever observations are added to an existing compressed data set.

Note: The REUSE= data set option overrides the REUSE= system option.

If the REUSE= option is set to YES, observations that are added to the SAS data set are inserted wherever enough free space exists, instead of at the end of the SAS data set. Specifying NO for the REUSE= option results in less efficient usage of space if you delete or update many observations in a SAS data set because there will be unused space within the data set. With the REUSE= option set to NO, the APPEND procedure, the FEDIT procedure, and other procedures that add observations to the SAS data set add observations to the end of the data set, as they do for uncompressed data sets.

You cannot change the REUSE= attribute of a compressed data set after it is created. This means that space is tracked and reused in the compressed SAS data set according to the value of the REUSE= option that was specified when the SAS data set was created, not when you add and delete observations. Also, you should be aware that even with the REUSE= option set to YES, the APPEND procedure will add observations to the end of the data set.
CAUTION:
Specifying YES as the value for the REUSE= option causes the POINTOBS= option to have a value of NO even if you specify YES as the value for POINTOBS=. The insertion of a new observation into unused space (rather than at the end of the data set) and the use of direct access are not compatible.

---

Example

The following program creates a compressed data set named *Company.Customer_Compressed* from the *Company.Customer* data set. Because the REUSE= option is set to YES, SAS will track and reuse any empty space within the compressed data set.

```sas
data company.customer_compressed (compress=yes reuse=yes);
  set company.customer;
run;
```

How SAS Compresses Data

Let’s take a closer look at how SAS compresses data. A fictional data set named *Roster* is described in the table below.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>FirstName</td>
<td>Character</td>
<td>15</td>
</tr>
</tbody>
</table>

In uncompressed form, each observation in *Roster* uses a total of 35 bytes to store these two variables: 20 bytes for the first variable, *LastName*, and 15 bytes for the second variable, *FirstName*. The image below illustrates the storage of the first observation in the uncompressed version of *Roster*.

```
LastName: ADAMS
FirstName: BILL
```

Suppose that you use the CHAR setting for the COMPRESS= option to compress *Roster*. In compressed form, the repeated blanks are removed from each value. The first observation from *Roster* uses a total of only 13 bytes: 7 for the first variable, *LastName*, and 6 for the second variable, *FirstName*. The image below illustrates the storage of the first observation in the compressed version of *Roster*.

```
LastName: A@DAMS
FirstName: B@ILL
```

The @ indicates the number of uncompressed characters that follow. The # indicates the number of blanks repeated at this point in the observation. Only a SAS engine can access these bytes. You cannot print or manipulate them.
Comparative Example: Creating and Reading Compressed Data Files

Suppose you want to create two SAS data sets from data that is stored in two raw data files. The raw data file that is referenced by the fileref flat1 contains numeric data about customer orders for a retail company; you want to create a SAS data set named Retail.Orders from this raw data file. The raw data file that is referenced by the fileref flat2 contains character data about customers for a retail company; you want to create a SAS data set named Retail.Customers from this raw data file.

In both cases, you can use the DATA step to create either an uncompressed data file or a compressed data file. Furthermore, you can use either binary or character compression in either case. That is, you can use the following techniques:

1. Numeric Data, No Compression
2. Numeric Data, BINARY Compression
3. Numeric Data, CHAR Compression
4. Character Data, No Compression
5. Character Data, BINARY Compression
6. Character Data, CHAR Compression.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for creating compressed data files.

Programming Techniques

1. Numeric Data, No Compression

The following program creates the SAS data set Retail.Orders, which contains numeric data and is uncompressed. The second DATA step reads the uncompressed data file.

```sas
data retail.orders(compress=no);
  infile flat1;
  input Customer_ID 12.
    Employee_ID 12.
    Street_ID 12.
    Order_Date date9.
    Delivery_Date date9.
    Order_ID 12.
    Order_Type comma16.
    Product_ID 12.
    Quantity 4.
    Total_Retail_Price dollar13.2
    CostPrice_Per_Unit dollar13.2
    Discount 5.2;
run;
```

```sas
data _null_; set retail.orders; run;
```
Numeric Data, BINARY Compression

The following program creates the SAS data set `Retail.Orders_binary`, which contains numeric data and uses BINARY compression. The second DATA step reads the compressed data file.

```sas
data retail.orders_binary(compress=binary);
  infile flat1;
  input Customer_ID 12.
  Employee_ID 12.
  Street_ID 12.
  Order_Date date9.
  Delivery_Date date9.
  Order_ID 12.
  Order_Type comma16.
  Product_ID 12.
  Quantity 4.
  Total_Retail_Price dollar13.2
  CostPrice_Per_Unit dollar13.2
  Discount 5.2;
run;

data _null_;  
  set retail.orders_binary;  
run;
```

Numeric Data, CHAR Compression

The following program creates the SAS data set `Retail.Orders_char`, which contains numeric data and uses CHAR compression. The second DATA step reads the compressed data file.

```sas
data retail.orders_char(compress=char);
  infile flat1;
  input Customer_ID 12.
  Employee_ID 12.
  Street_ID 12.
  Order_Date date9.
  Delivery_Date date9.
  Order_ID 12.
  Order_Type comma16.
  Product_ID 12.
  Quantity 4.
  Total_Retail_Price dollar13.2
  CostPrice_Per_Unit dollar13.2
  Discount 5.2;
run;

data _null_;  
  set retail.orders_char;  
run;
```
Comparative Example: Creating and Reading Compressed Data Files

Character Data, No Compression

The following program creates the SAS data set `Retail.Customers`, which contains character data and is uncompressed. The second DATA step reads the uncompressed data file.

```sas
data retail.customers(compress=no);
    infile flat2;
    input Customer_Country $40.
        Customer_Gender $1.
        Customer_Name $40.
        Customer_FirstName $20.
        Customer_LastName $30.
        Customer_Age_Group $12.
        Customer_Type $40.
        Customer_Group $40.
        Customer_Address $45.
        Street_Number $8. ;
run;

data _null_;  
    set retail.customers;  
run;
```

Character Data, BINARY Compression

The following program creates the SAS data set `Retail.Customers_binary`, which contains character data and uses `BINARY` compression. The second DATA step reads the compressed data file.

```sas
data retail.customers_binary(compress=binary);
    infile flat2;
    input Customer_Country $40.
        Customer_Gender $1.
        Customer_Name $40.
        Customer_FirstName $20.
        Customer_LastName $30.
        Customer_Age_Group $12.
        Customer_Type $40.
        Customer_Group $40.
        Customer_Address $45.
        Street_Number $8. ;
run;

data _null_;  
    set retail.customers_binary;  
run;
```
The following program creates the SAS data set `Retail.Customers_char`, which contains character data and uses `CHAR` compression. The second DATA step reads the compressed data file.

```sas
data retail.customers_char(compress=char);
  infile flat2;
  input Customer_Country $40.
  Customer_Gender $1.
  Customer_Name $40.
  Customer_FirstName $20.
  Customer_LastName $30.
  Customer_Age_Group $12.
  Customer_Type $40.
  Customer_Group $40.
  Customer_Address $45.
  Street_Number $8.;
run;

data _null_; set retail.customers_char; run;
```

**General Recommendations**

- Save data storage space by compressing data, but remember that compressed data causes an increase in CPU usage because the data must be uncompressed for processing. Compressing data always uses more CPU resources than not compressing data.

- Use binary compression only if the observation length is several hundred bytes or more.

**Using SAS DATA Step Views to Conserve Data Storage Space**

Another way to save disk space is to leave your data in its original location and use a SAS data view to access it. Before looking at working with data views, let’s look at what a SAS data view is and how it compares to a SAS data file.

A SAS data file and a SAS data view are both types of SAS data sets. The first type, a SAS data file, contains both descriptor information about the data and the data values. The second type, a SAS data view, contains only descriptor information about the data and instructions on how to retrieve data values that are stored elsewhere.
The main difference between SAS data files and SAS data views is where the data values are stored. A SAS data file contains the data values, and a SAS data view does not contain the values. Therefore, data views can be particularly useful if you are working with data values that change often.

Suppose you have a flat file that you read into a SAS data file. If the values in the flat file change, you need to update the data file in order to reflect those changes so that you access the correct values when you reference the data file. However, suppose you use a data view to access the values in your flat file instead of reading those values into a data file. You do not need to update the data view when the values in your flat file change, because each time you reference the view it will execute and access the most recent values in your flat file.

In most cases, you can use a SAS data view as if it were a SAS data file, although there are a few things to keep in mind when you are working with data views.

Note: There are multiple types of SAS data views. This chapter discusses only DATA step views. To learn more about PROC SQL views, see Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243. For more information about SAS data views and SAS data files, see the SAS documentation.

Now let’s look at DATA step views.

DATA Step Views

A DATA step view contains a partially compiled DATA step program that can read data from a variety of sources, including

- raw data files
- SAS data files
- PROC SQL views
- SAS/ACCESS views
- DB2, ORACLE, or other DBMS data.

A DATA step view can be created only in a DATA step. A DATA step view cannot contain global statements, host-specific data set options, or most host-specific FILE and INFILE statements. Also, a DATA step view cannot be indexed or compressed.

You can use DATA step views to

- always access the most current data in changing files
avoid storing a copy of a large data file
combine data from multiple sources.

The compiled code does not take up much room for storage, so you can create DATA step views to conserve disk space. On the other hand, use of DATA step views can increase CPU usage because SAS must execute the stored DATA step program each time you use the view.

To create a DATA step view, specify the VIEW= option after the final data set name in the DATA statement.

General form, DATA step to create a DATA step view:

```
DATA SAS-data-view <SAS-data-file-1 ... SAS data-file-n> /
    VIEW=SAS-data-view;
    <SAS statements>
RUN;
```

where

- **SAS-data-view**
  names the data view to be created.

- **SAS-data-file-1 ... SAS-data-file-n**
  is an optional list that names any data files to be created.

- **SAS statements**
  includes other DATA step syntax to create the data view and any data files that are listed in the DATA statement.

The VIEW= option tells SAS to compile, but not to execute, the source program and to store the compiled code in the input DATA step view that is named in the option.

*Note:* If you specify additional data files in the DATA statement, SAS creates these data files when the view is processed in a subsequent DATA or PROC step. Therefore, you need to reference the data view before you attempt to reference the data file in later steps. △

**Example**

The following program creates a DATA step view named *Company.Newdata* that reads from the file referenced by the fileref in the INFILE statement.

```
data company.newdata / view=company.newdata;
    infile <fileref>;
    <DATA step statements>
run;
```

**The DESCRIBE Statement**

Beginning in SAS 8, DATA step views retain source statements. You can retrieve these statements by using the DESCRIBE statement. The following example uses the DESCRIBE statement in a DATA step to write a copy of the source code for the data view *Company.Newdata* to the SAS log:
data view=company.newdata;
  describe;
run;

Now let’s look at using DATA step views.

### Creating and Referencing a SAS DATA Step View

In order to use DATA step views successfully, you need to understand what happens when you create and reference one.

When you create a DATA step view,
- the DATA step is partially compiled
- the intermediate code is stored in the specified SAS data library with a member type of VIEW.

You reference a DATA step view in the same way that you reference a data file. When you reference the view in a subsequent DATA or PROC step,
- the compiler resolves the intermediate code and generates executable code for the host environment
- the generated code is executed as the DATA or PROC step requests observations.

You can use a DATA step view as you would use any other SAS data set, with the exception that you cannot write to the view except under very specific circumstances. Also, you should keep in mind that a SAS data view reads from its source files each time it is used, so if the data that it is accessing changes, the view will change also. Likewise, if the structure of the data that a view accesses changes, you will probably need to alter the view in order to account for this change.

**Note:** In SAS 9.1, the OBSBUF= data set option enables you to specify how many observations to read at one time from the source data for the DATA step view. The default size of the view buffer is 32K, which means that the number of observations that can be read into the view buffer at one time depends on the observation length. If
the observation length is larger than 32K, then only one observation can be read into the buffer at a time.

Remember that although data views conserve data storage space, processing them can require more resources than processing a data file. Let’s look at a few situations where using a data view can adversely affect processing efficiency.

---

**Referencing a Data View Multiple Times in One Program**

SAS executes a view each time it is referenced, even within one program. Therefore, if data is used many times in one program, it is more efficient to create and reference a temporary SAS data file than to create and reference a view.

**Example**

Instead of referencing a data view in each step in the program, you could add a DATA step to the beginning of the program to create a temporary data file and read the data view into it. Then you could reference the temporary data set in each of the subsequent steps. By referencing the temporary data file rather than the data view in each of the PROC steps, you enable SAS to execute the data view only once instead of three times.

There are other reasons why extracting data to a temporary data file is a good idea. Suppose you submit this code and it takes a long time to run. If someone changes the flat file while your code is running, you will have inconsistent results unless you have created a SAS data file before submitting the PROC PRINT, PROC FREQ, and PROC MEANS steps, and you use the data file in your program.

**Making Multiple Passes Through Data in a Data View**

Expect a degradation in performance when you use a SAS data view with a procedure that requires multiple passes through the data. When multiple passes are requested, the view must build a cache, which is referred to as a spill file, that contains all generated observations. Then SAS reads the data in the spill file on each of the multiple passes through the data in order to ensure that subsequent passes read the same data that was read by previous passes.

For example, the UNIFORM option of the PRINT statement makes all the columns consistent from page to page by determining the longest value for a particular variable. In order to do this, SAS must make two passes through the data: one pass to find the longest value in the data, and one pass to print the data. If you use the UNIFORM option to print a data view, SAS creates a spill file as it generates observations from the view. Then SAS makes two passes through the observations that are in the spill file.
Creating Data Views on Unstable Data

Avoid creating views on files whose structures often change. The view syntax describes the structure of the raw data; therefore, you need to make changes to the view each time the file changes.

For example, suppose you create a view that combines the data file Company.Roster with the data file Company.Demog. Roster contains the variables LastName and FirstName, and Company.Demog contains the variables LastName, Address, and Age, as shown below.

<table>
<thead>
<tr>
<th>Company.Roster</th>
<th>Company.Demog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>Type</td>
</tr>
<tr>
<td>LastName</td>
<td>Character</td>
</tr>
<tr>
<td>FirstName</td>
<td>Character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>Address</td>
<td>Character</td>
<td>45</td>
</tr>
<tr>
<td>Age</td>
<td>Numeric</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that both Company.Roster and Company.Demog are sorted by LastName. You could use a MERGE statement to combine these two data files into a view named Company.Roster_View, as shown below.

```plaintext
data company.roster_view/view=company.roster_view;
  merge company.roster company.demog;
  by lastname;
run;
```

Now suppose Company.Roster changes so that LastName is named Surname. Your data view must also be updated.

<table>
<thead>
<tr>
<th>Company.Roster</th>
<th>Company.Demog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>Type</td>
</tr>
<tr>
<td>Surname</td>
<td>Character</td>
</tr>
<tr>
<td>FirstName</td>
<td>Character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>Character</td>
<td>20</td>
</tr>
<tr>
<td>Address</td>
<td>Character</td>
<td>45</td>
</tr>
<tr>
<td>Age</td>
<td>Numeric</td>
<td>3</td>
</tr>
</tbody>
</table>

```plaintext
data roster_view/view=roster_view;
  merge company.roster company.demog(rename=(LastName=Surname));
  by lastname;
run;
```

If Company.Roster changed again so that Surname and FirstName were combined into one variable called FullName, the code for your data view would need additional changes. Although this is a simple example, you can see that a data view that is based on unstable data will require additional maintenance work.
Comparative Example: Creating and Reading a SAS Data View

Suppose you have two SAS data sets, Retail.Custview and Retail.Custdata, that have been created from the same raw data file. Retail.Custview is a DATA step view, and Retail.Custdata is a data file. You can use these two data sets to compare the disk space that is required for each as well as the resources that are used to read from each:

1. Data View
2. Data File

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using SAS DATA step views.

Programming Techniques

1. Data View

This program reads data from a raw data file and creates a SAS DATA step view named Retail.Custview, then reads from the new DATA step view. The first DATA step creates the data view Retail.Custview. The second DATA step reads from the DATA step view.

```sas
data retail.custview / view = retail.custview;
infile flat1;
  input @1 Customer_ID 12.
  @13 Country $2.
  @15 Gender $1.
  @16 Personal_ID $15.
  @31 Customer_Name $40.
  @71 Customer_FirstName $20.
  @91 Customer_LastName $30.
  @121 Birth_Date date9.
  @130 Customer_Address $45.
  @175 Street_ID 12.
  @199 Street_Number $8.
  @207 Customer_Type_ID 8.;
run;

data _null_; set retail.custview;
run;
```
## Data View

This program reads data from a raw data file and creates a SAS data file named `Retail.Custdata`, then reads from the new SAS data file. The first DATA step creates the data file `Retail.Custdata`. The second DATA step reads from the data file.

```sas
data retail.custdata;
  infile flat1;
  input @1 Customer_ID 12.
    @13 Country $2.
    @15 Gender $1.
    @16 Personal_ID $15.
    @31 Customer_Name $40.
    @71 Customer_FirstName $20.
    @91 Customer_LastName $30.
    @121 Birth_Date date9.
    @130 Customer_Address $45.
    @175 Street_ID 12.
    @199 Street_Number $8.
    @207 Customer_Type_ID 8.;
run;

data _null_;
  set retail.custdata;
run;
```

## General Recommendations

- Create a SAS DATA step view to avoid storing a raw data file and a copy of that data in a SAS data file.
- Use a SAS DATA step view if the content, but not the structure, of the flat file is dynamic.
Summary

Reducing Data Storage Space for Character Variables

SAS stores character data as one character per byte. The default length for a character variable is 8 bytes. You can use the LENGTH statement to reduce the length of a character variable. You can also use other coding techniques to reduce the space that is needed for storing your character data.

Reducing Data Storage Space for Numeric Variables

SAS stores numeric data in floating-point representation. The default length for a numeric variable is 8 bytes. You can use a LENGTH statement to reduce the length of a numeric variable. Reading reduced-length numeric variables requires less I/O but more CPU resources than reading full-length numeric variables. You should store only integer values in reduced-length numeric variables, and you should limit the values according to the length that you use. You can use PROC COMPARE to see the precision loss, if any, in the values of reduced-length numeric variables.

Review the related comparative example:
- “Comparative Example: Creating a SAS Data Set That Contains Reduced-Length Numeric Variables” on page 716.

Compressing Data Files

By default, a SAS data file is uncompressed. You can compress your data files in order to conserve disk space, although some files are not good candidates. COMPRESS= data set option or system option to compress a data file. You use the POINTOBS= data set option to enable SAS to access observations in compressed files directly rather than sequentially. You use the REUSE= data set option or system option to specify that SAS should reuse space in a compressed file when observations are deleted or updated.

Review the related comparative example:
- “Comparative Example: Creating and Reading Compressed Data Files” on page 727.

Using SAS DATA Step Views to Conserve Data Storage Space

You can leave your data in its original storage location and use SAS data views to access the data in order to reduce the amount of space needed for storing data on disk. A DATA step view is a specific type of data view that is created in a DATA step with the VIEW= option. You use the DESCRIBE statement to write the source code for a data view to the SAS log. Some of the advantages of using DATA step views rather than data files are that they always access the most recent data in dynamic files and that they require less disk space. However, there can be an effect on performance when you use a DATA step view.

Review the related comparative example:
- “Comparative Example: Creating and Reading a SAS Data View” on page 736.
Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Which of the following statements about uncompressed SAS data files is true?
   a The descriptor portion is stored on whatever page has enough room for it.
   b New observations are always added in the first sufficient available space.
   c Deleted observation space is tracked.
   d New observations are always added at the end of the data set.

2 Which of the following statements about compressed SAS data files is true?
   a The descriptor portion is stored on whatever data set page has enough room for it.
   b Deleted observation space can be reused.
   c Compressed SAS data files have a smaller overhead than uncompressed SAS data files.
   d In a compressed SAS data set, each observation must be the same size.

3 Which of the following programs correctly creates reduced-length numeric variables?
   a data temp;
       infile file1;
       input x 4.
       y 3.
       z 2.;
       run;
   b data temp;
       format x 4.
       y 3.
       z 2.;
       infile file1;
       input x 4.
       y 3.
       z 2.;
       run;
   c data temp;
       length x 4
       y 3
       z 2;
       infile file1;
       input x 4.
       y 3.
       z 2.;
       run;
   d data temp;
       informat x 4.
       y 3.
       z 2.;
       infile file1;
       input x 4.
       y 3.
       z 2.;
4 Which of the following statements about SAS data views is true?
   a. SAS data views use less disk space but more CPU resources than SAS data files.
   b. SAS data views can be created only in permanent SAS data libraries.
   c. SAS data views use less CPU resources but more disk space than SAS data files.
   d. SAS data views can be created only in temporary SAS data libraries.

5 Which of the following programs should you use to detect any loss of precision between the default-length numeric variables in Company.Regular and the reduced-length numeric variables in the data set Company.Reduced?
   a. proc contents data=company.regular;
      compare data=company.reduced;
   run;
   b. proc compare base=company.regular
      compare=company.reduced;
   run;
   c. proc print data=company.regular;
   run;
      proc print data=company.reduced;
   run;
   d. proc datasets library=company;
      contents data=regular compare=reduced;
   run;
CHAPTER 22
Utilizing Best Practices

Overview 742
   Introduction 742
   Objectives 743
   Prerequisites 743

Executing Only Necessary Statements 743
   Positioning of the Subsetting IF Statement 744
      Comparative Example: Creating a Subset of Data 744
         Programming Techniques 745
         General Recommendations 745
   Using Conditional Logic Efficiently 746
      Comparative Example: Creating Variables Conditionally Using DO Groups 747
         Programming Techniques 748
         General Recommendations 749
      Comparative Example: Creating Variables Conditionally When Calling Functions 749
         Programming Techniques 751
         General Recommendations 752

Using DO Groups Efficiently 753
   Comparative Example: Creating Data in DO Groups 754
      Programming Techniques 755
      General Recommendations 758

Eliminating Unnecessary Passes through the Data 758
   Using a Single DATA or PROC Step to Enhance Efficiency 758
   Using a Single DATA Step to Create Multiple Output Data Sets 759
   Using the SORT Procedure with a WHERE Statement to Create Sorted Subsets 759
   Using the DATASETS Procedure to Modify Variable Attributes 759
      Comparative Example: Creating Multiple Subsets of a SAS Data Set 760
         Programming Techniques 760
         General Recommendations 761
      Comparative Example: Creating a Sorted Subset of a SAS Data Set 761
         Programming Techniques 762
         General Recommendations 762
      Comparative Example: Changing the Variable Attributes of a SAS Data Set 762
         Programming Techniques 763
         General Recommendations 763

Reading and Writing Only Essential Data 763
   Selecting Observations Using Subsetting IF versus WHERE Statement 764
      Comparative Example: Creating a Subset of a SAS Data Set 764
         Programming Techniques 765
         General Recommendations 765
   Other Differences between the IF and WHERE Statements 765
   Using the WHERE Statement with the OBS= and FIRSTOBS= Options 766
Overview

Introduction

This chapter demonstrates different ways of utilizing best SAS programming practices to optimize performance. As you compare the techniques described in this chapter, remember that differences in the use of resources are affected by which operating environment you work in and by the characteristics of your data.

This chapter is organized by topics that emphasize the following basic principles:

- Execute only necessary statements.
- Eliminate unnecessary passes of the data.
- Read and write only the data that you require.
- Store data in SAS data sets.
- Avoid unnecessary procedure invocation.

Each topic includes comparative examples that can improve the efficiency of your programs. Write programs to generate your own benchmarks, and adopt the programming techniques that produce the most savings for you.

Note: This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SAS SPD Engine to improve performance, see the SAS documentation.
Objectives

In this chapter, you learn to efficiently
- subset observations
- create new variables
- process and output data conditionally
- create multiple output data sets and sorted subsets
- modify variable attributes
- select observations from SAS data sets and external files
- subset variables
- read data from SAS data sets
- invoke SAS procedures.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

Part 1: SQL Processing with SAS
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
- Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125
- Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159
- Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221
- Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243
- Chapter 8, “Managing Processing Using PROC SQL,” on page 261.

Part 3: Advanced SAS Programming Techniques
- Chapter 13, “Creating Samples and Indexes,” on page 451
- Chapter 14, “Combining Data Vertically,” on page 481
- Chapter 15, “Combining Data Horizontally,” on page 513
- Chapter 16, “Using Lookup Tables to Match Data,” on page 559
- Chapter 17, “Formatting Data,” on page 603

Part 4: Optimizing SAS Programs
- Chapter 19, “Introduction to Efficient SAS Programming,” on page 679
- Chapter 20, “Controlling Memory Usage,” on page 689
- Chapter 21, “Controlling Data Storage Space,” on page 707.

Executing Only Necessary Statements

Best practices dictate that you should write programs that cause SAS to execute only necessary statements. When you execute the minimum number of statements in the most efficient order, you minimize the hardware resources that SAS uses. The resources that are affected include disk usage, memory usage, and CPU usage.

Techniques for executing only the statements you need include
- subsetting your data as soon as is logically possible
Positioning of the Subsetting IF Statement

To subset your data based on a newly derived or computed variable, you must use the subsetting IF statement in a DATA step. As you output data to a SAS data set, you can subset it by processing only those observations that meet a specified condition.

The subsetting IF statement causes the DATA step to continue processing only those raw data records or observations that meet the condition of the expression specified in the IF statement. The resulting SAS data set or data sets contain a subset of the original external file or SAS data set.

Position the subsetting IF statement in the program so that it checks the subsetting condition as soon as it is logically possible, and so that unnecessary statements do not execute. When a statement is false, no further statements are processed for that observation.

Also, remember to subset data before performing calculations and to minimize the use of function calls or arithmetic operators. Unnecessary processing of unwanted observations results in higher expenditure of hardware resources.

Comparative Example: Creating a Subset of Data

Suppose you want to create a subset of data, calculate six new variables, and conditionally output data by reading from the SAS data set `Retail.Order_fact`. The data set should contain new variables for:

- the month of the order
- the elapsed time between the order date and the delivery date
- the profit, based on the retail price, discount, and unit price
- total profit
- total discount
- total wait time.

The subset of data that includes only orders for the month of December is approximately 9.66% of the data.

You can accomplish this task by using a subsetting IF statement. Placement of this statement in the DATA step can affect the efficiency of the DATA step in terms of CPU time and real time. Notice the comparison between these two approaches:

1. A Subsetting IF Statement at the Bottom
2. A Subsetting IF Statement near the Top.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment,
and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

**Programming Techniques**

1. **A Subsetting IF Statement at the Bottom**
   This program calculates six new variables before the subsetting IF statement selects only observations whose values for Month are 12.

   ```
   data profit;
   retain TotalProfit TotalDiscount TotalWait Count 0;
   set retail.order_fact;
   MonthOfOrder=month(order_date);
   WaitTime=sum(delivery_date,-order_date);
   if discount gt . then
     CalcProfit=sum((total_retail_price*discount),-costprice_per_unit)*quantity;
   else CalcProfit=sum(total_retail_price,-costprice_per_unit)*quantity;
   TotalProfit=sum(totalprofit,calcprofit);
   TotalDiscount=sum(totaldiscount,discount);
   TotalWait=sum(totalwait,waittime);
   Count+1;
   if monthoforder=12;
   run;
   ```

2. **A Subsetting IF Statement near the Top**
   In this program, the subsetting IF statement is positioned immediately after the value for MonthofOrder has been calculated. If the value is not 12, then no further statements are processed for that observation. In this program, calculations are performed on a smaller number of observations, which results in greater program efficiency.

   ```
   data profit;
   retain TotalProfit TotalDiscount TotalWait Count 0;
   set retail.order_fact;
   MonthOfOrder=month(order_date);
   if monthoforder=12;
   WaitTime=sum(delivery_date,-order_date);
   if discount gt . then
     CalcProfit=sum((total_retail_price*discount),-costprice_per_unit)*quantity;
   else CalcProfit=sum(total_retail_price,-costprice_per_unit)*quantity;
   TotalProfit=sum(totalprofit,calcprofit);
   TotalDiscount=sum(totaldiscount,discount);
   TotalWait=sum(totalwait,waittime);
   Count+1;
   run;
   ```

**General Recommendations**

- Position the subsetting IF statement in a DATA step as soon as is logically possible in order to save the most resources.
In the last comparative example you saw how a subsetting IF statement can be positioned in the DATA step so that no further statements are processed for that observation. Next, let’s look at how different programming techniques can be used to

- create variables conditionally using DO groups
- create variables conditionally when calling functions.

Before viewing the sample code for these two comparative examples, let’s review guidelines for using these techniques.

**Using Conditional Logic Efficiently**

You can use conditional logic to change how SAS processes selected observations. Two techniques—IF-THEN/ELSE statements and SELECT statements—can be used interchangeably and perform comparably. Based on the characteristics of your data and depending on your environment, one of these techniques might give you better performance. Choose a technique that conserves your programming time and makes the program easiest to read.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF-THEN/ELSE statement</td>
<td>executes a SAS statement for observations that meet specific conditions.</td>
</tr>
<tr>
<td>SELECT statement</td>
<td>executes one of several statements or groups of statements.</td>
</tr>
</tbody>
</table>

*Note:* The number of conditions (values) tested and the type of variable tested affect CPU resources. For numeric variables, SELECT statements should always be slightly more efficient (use less CPU time) than IF-THEN/ELSE statements. The performance gap between IF-THEN/ELSE and SELECT statements gradually widens as the number of conditions increases. For character variables, IF-THEN/ELSE statements are always more efficient than SELECT statements. The performance gap widens quickly between the two techniques as the number of conditions increases.

Use IF-THEN/ELSE statements when

- the data values are character values
- the data values are not uniformly distributed
- there are few conditions to check.

For best practices, follow these guidelines for writing efficient IF/THEN logic:

- For mutually exclusive conditions, use the ELSE IF statement rather than an IF statement for all conditions except the first.
- Check the most frequently occurring condition first, and continue checking conditions in descending order of frequency.
- When you execute multiple statements based on a condition, put the statements in a DO group.

Use SELECT statements when

- you have a long series of mutually exclusive numeric conditions
- data values are uniformly distributed.

Before writing conditional logic, determine the distribution of your data values. You can use the
Utilizing Best Practices △ Comparative Example: Creating Variables Conditionally Using DO Groups

- FREQ procedure to examine the distribution of the data values
- GCHART or GPLOT procedure to display the distribution graphically
- UNIVARIATE procedure to examine distribution statistics and to display the information graphically.

Comparative Example: Creating Variables Conditionally Using DO Groups

Suppose you want to calculate an adjusted profit based on the values of the variable **Order_Type** in the data set *Retail.Order_fact*. For retail sales, which are represented by the value 1, the adjusted profit should be calculated as 105% of profit. For catalog sales, which are represented by the value 2, the adjusted profit should be calculated as 103% of profit. For internet sales, which are represented by the value 3, the adjusted profit should be equal to profit.

The following table shows that the values for the variable **Order_Type** are not uniformly distributed.

<table>
<thead>
<tr>
<th>Order Type</th>
<th>Frequency</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3379850</td>
<td>75.23</td>
</tr>
<tr>
<td>2</td>
<td>635645</td>
<td>13.36</td>
</tr>
<tr>
<td>3</td>
<td>542830</td>
<td>11.41</td>
</tr>
</tbody>
</table>

The following table shows that the values for the variable **Discount** also are not uniformly distributed.

<table>
<thead>
<tr>
<th>Discount</th>
<th>Frequency</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>4712585</td>
<td>99.04</td>
</tr>
<tr>
<td>30%</td>
<td>19740</td>
<td>0.41</td>
</tr>
<tr>
<td>40%</td>
<td>15170</td>
<td>0.32</td>
</tr>
<tr>
<td>50%</td>
<td>9805</td>
<td>0.21</td>
</tr>
<tr>
<td>60%</td>
<td>1045</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Techniques for creating new variables conditionally include

1. IF-THEN/ELSE statements
2. SELECT statements.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.
Programming Techniques

1. IF-THEN/ELSE Statements

This program uses IF-THEN/ELSE statements with DO groups to conditionally execute multiple statements that calculate an adjusted profit. Conditions are checked in descending order of frequency.

```sas
data retail.order_info_1;
  set retail.order_fact;
  if order_type=1 then
    do; /* Retail Sale */
      Float=delivery_date-order_date;
      RevenueQuarter=qtr(order_date);
      AveragePrice=total_retail_price/quantity;
      if discount=. then NetPrice=total_retail_price;
      else NetPrice=total_retail_price-discount;
      Profit=netPrice-(quantity*costprice_per_unit)*1.05;
    end;
  else if order_type=2 then
    do; /* Catalog Sale */
      Float=delivery_date-order_date;
      RevenueQuarter=qtr(order_date);
      AveragePrice=total_retail_price/quantity;
      if discount=. then NetPrice=total_retail_price;
      else NetPrice=total_retail_price-discount;
      Profit=netprice-(quantity*costprice_per_unit)*1.03;
    end;
  else
    do; /* Internet Sale */
      Float=delivery_date-order_Date;
      RevenueQuarter=qtr(order_date);
      AveragePrice=total_retail_price/quantity;
      if discount=. then NetPrice=total_retail_price;
      else NetPrice=total_retail_price-discount;
      Profit=netprice-(quantity*costprice_per_unit);
    end;
run;
```
SELECT Statements

This program uses SELECT/WHEN statements with DO groups to conditionally execute multiple statements that calculate an adjusted profit. Conditions are checked in descending order of frequency.

data retail.order_info_2;
  set retail.order_fact;
  select(order_type);
  when (1)
    do; /* Retail Sale */
      Float=delivery_date-order_date;
      RevenueQuarter=qtr(order_date);
      AveragePrice=total_retail_price/quantity;
      if discount=. then NetPrice=total_retail_price;
      else NetPrice=total_retail_price-discount;
      Profit=netprice-(quantity*costprice_per_unit)*1.05;
      end;
    when (2)
    do; /* Catalog Sale */
      Float=delivery_date-order_date;
      RevenueQuarter=qtr(order_date);
      AveragePrice=total_retail_price/quantity;
      if discount=. then NetPrice=total_retail_price;
      else NetPrice=total_retail_price-discount;
      Profit=netprice-(quantity*costprice_per_unit)*1.03;
    end;
  otherwise
    do; /* Internet Sale */
      Float=delivery_date-order_date;
      RevenueQuarter=qtr(order_date);
      AveragePrice=total_retail_price/quantity;
      if discount=. then NetPrice=total_retail_price;
      else NetPrice=total_retail_price-discount;
      Profit=netprice-(quantity*costprice_per_unit);
    end;
end;
run;

General Recommendations

☐ Check the most frequently occurring condition first, and continue checking conditions in descending order of frequency, regardless of whether you use IF-THEN/ELSE or SELECT statements.

☐ When you execute multiple statements based on a condition, put the statements in a DO group.

Comparative Example: Creating Variables Conditionally When Calling Functions

Suppose you want to create a report that includes a new variable that is based on the value of an existing variable in the SAS data set Retail.Order_fact. Values for the new Month variable are extracted from the existing variable Order_Date by using the MONTH function.

The following table shows that the values for Month are fairly evenly distributed.
Techniques for creating new variables conditionally include
1 Parallel IF Statements
2 ELSE IF Statements, Many Function References
3 ELSE IF Statements, One Function Reference
4 SELECT Group.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.
Programming Techniques

\[1\] Parallel IF Statements
This program calls the MONTH function 12 times. With these non-exclusive cases, each IF statement executes for each observation that is read from Retail.Order.Fact. This is the least efficient approach.

```sas
data retail.orders;
  set retail.order_fact;
  if month(order_date)=1 then Month='Jan';
  if month(order_date)=2 then Month='Feb';
  if month(order_date)=3 then Month='Mar';
  if month(order_date)=6 then Month='Jun';
  if month(order_date)=7 then Month='Jul';
  if month(order_date)=8 then Month='Aug';
  if month(order_date)=9 then Month='Sep';
  if month(order_date)=10 then Month='Oct';
  if month(order_date)=11 then Month='Nov';
  if month(order_date)=12 then Month='Dec';
run;
```

\[2\] ELSE IF Statements, Many Function References
This program uses ELSE IF statements that call the function MONTH. Once the true condition is found, subsequent ELSE IF statements are not executed. This is more efficient than using parallel IF statements, but the MONTH function is executed many times.

```sas
data retail.orders;
  set retail.order_fact;
  if month(order_date)=1 then Month='Jan';
  else if month(order_date)=2 then Month='Feb';
  else if month(order_date)=3 then Month='Mar';
  else if month(order_date)=4 then Month='Apr';
  else if month(order_date)=5 then Month='May';
  else if month(order_date)=6 then Month='Jun';
  else if month(order_date)=7 then Month='Jul';
  else if month(order_date)=10 then Month='Oct';
  else if month(order_date)=11 then Month='Nov';
  else if month(order_date)=12 then Month='Dec';
run;
```
ELSE IF Statements, One Function Reference

This program uses the MONTH function to find the value of `Order_Date`, but only once. The MONTH function is called immediately after reading the data set and before any IF-THEN/ELSE statements execute. This is efficient.

```sas
data retail.orders(drop=mon);
set retail.order_fact;
mon=month(order_date);
if mon=1 then Month='Jan';
else if mon=2 then Month='Feb';
else if mon=3 then Month='Mar';
else if mon=4 then Month='Apr';
else if mon=5 then Month='May';
else if mon=6 then Month='Jun';
else if mon=7 then Month='Jul';
else if mon=8 then Month='Aug';
else if mon=9 then Month='Sep';
else if mon=10 then Month='Oct';
else if mon=11 then Month='Nov';
else if mon=12 then Month='Dec';
run;
```

SELECT Group

In this program, the SELECT statement calls the MONTH function only once, before WHEN statements execute and assign values for `Month`. This is efficient.

```sas
data retail.orders;
set retail.order_fact;
select(month(order_date));
when (1) Month='Jan';
when (2) Month='Feb';
when (3) Month='Mar';
when (4) Month='Apr';
when (5) Month='May';
when (6) Month='Jun';
when (7) Month='Jul';
when (8) Month='Aug';
when (11) Month='Nov';
when (12) Month='Dec';
otherwise;
end;
run;
```

General Recommendations

- Avoid using parallel IF statements, which use the most resources and are the least efficient way to conditionally execute statements.
- Use IF-THEN/ELSE statements and SELECT blocks to be more efficient.
- To significantly reduce the amount of resources used, write programs that call a function only once instead of repetitively using the same function in many statements. SAS functions are convenient, but they can be expensive in terms of CPU resources.

You’ve seen different ways of efficiently creating variables conditionally, which reinforce the principle of executing only necessary statements. The next comparative
example in this topic addresses techniques that can be used to create data in DO groups. Before viewing this example, you might want to review the following information about which statements are needed in DO groups.

**Using DO Groups Efficiently**

You can conditionally execute only necessary statements by placing them in DO groups that are associated with IF-THEN/ELSE statements or with SELECT/WHEN statements. Groups of statements execute only when a particular condition is true. Remember to use the following criteria when choosing which technique is more efficient:

<table>
<thead>
<tr>
<th>IF-THEN/ELSE Statements</th>
<th>SELECT/WHEN Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>The number of conditions</td>
<td>few</td>
</tr>
<tr>
<td>The distribution of a variable’s values</td>
<td>not uniform</td>
</tr>
<tr>
<td>The type of variable</td>
<td>character</td>
</tr>
</tbody>
</table>

When using a DO group with IF-THEN/ELSE statements, add DO after the THEN clause, and add an END statement after all of the statements that you want executed as a group.

```
data orders;
  set company.orders;
  if order_type = 1 then
    do;
    <multiple executable statements here>
    end;
  else if order_type = 2 then
    do;
    <multiple executable statements here>
    end;
  else if order_type = 3 then
    do;
    <multiple executable statements here>
    end;
  run;

*Note:* Use an IF-THEN DO group when you create multiple variables based on a condition.

When using a DO group with SELECT/WHEN statements, add DO after the WHEN condition, and add an END statement after all of the statements that you want executed as a group. Use an OTHERWISE statement to specify the statements that you want executed if no WHEN condition is met.

```
data orders;
  set company.orders;
  select (order_type);
    when (1)
      do;
      <multiple executable statements here>
    end;
  run;
```
when (2)
do;
  <multiple executable statements here>
end;
when (3)
do;
  <multiple executable statements here>
end;
otherwise;
end;
run;

Remember that IF-THEN/ELSE and SELECT/WHEN logic require that there be no
intervening statements between the IF and the ELSE conditions or between the
SELECT and the WHEN conditions.

Comparative Example: Creating Data in DO Groups

Suppose you want to identify which customer groups are Club Members, Club Gold
Members, or Internet/Catalog members, based on data from the data set
Retail.Customer_hybrid. You also want to identify the nature of customer activity as
“inactive”, “low activity”, “medium activity”, or “high activity”.
The following table shows the distribution of values for Customer_Type_ID.

<table>
<thead>
<tr>
<th>Customer_Type_ID</th>
<th>Frequency</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>568446</td>
<td>12.39</td>
</tr>
<tr>
<td>1020</td>
<td>579156</td>
<td>12.62</td>
</tr>
<tr>
<td>1030</td>
<td>571608</td>
<td>12.46</td>
</tr>
<tr>
<td>1040</td>
<td>574209</td>
<td>12.52</td>
</tr>
<tr>
<td>2010</td>
<td>566457</td>
<td>12.35</td>
</tr>
<tr>
<td>2020</td>
<td>571914</td>
<td>12.47</td>
</tr>
<tr>
<td>2030</td>
<td>579054</td>
<td>12.62</td>
</tr>
<tr>
<td>3010</td>
<td>576810</td>
<td>12.57</td>
</tr>
</tbody>
</table>

Techniques for creating new variables based on the values of specific variables include
1  SELECT, IF/SELECT Statements
2  Nested SELECT Statements
3  Serial IF Statements
4  IF-THEN/ELSE IF Statements with a Link.

The following sample programs show each of these techniques. You can use these
samples as models for creating benchmark programs in your own environment. Your
results may vary depending on the structure of your data, your operating environment,
and the resources that are available at your site. You can also view general
recommendations for executing only necessary statements.
### Programming Techniques

**SELECT, IF/SELECT Statements**

This program creates a permanent SAS data set named `Retail.Customers` by reading the `Retail.Customer_hybrid` data set. SELECT/WHEN logic and SELECT/WHEN statements in an IF/THEN DO group populate the variables `Customer_Group` and `Customer_Activity`. If the value of the first two digits of `Customer_Type_ID` is 10, 20, or 30, then `Customer_Group` is populated. If the value of the first two digits of `Customer_Type_ID` is 10 or 20, then `Customer_Activity` is populated by reading the last two digits of `Customer_Type_ID`.

```sas
data retail.customers;
  length Customer_Group $ 26 Customer_Activity $ 15;
  set retail.customer_hybrid;
  select(substr(put(customer_type_ID,4.),1,2));
    when ('10') customer_group='Orion Club members';
    when ('20') customer_group='Orion Club Gold members';
    when ('30') customer_group='Internet/Catalog Customers';
  otherwise;
  end;
  if substr(put(customer_type_ID,4.),1,2) in ('10', '20') then do;
    select(substr(put(customer_type_ID,4.),3,2));
      when ('10') customer_activity='inactive';
      when ('20') customer_activity='low activity';
      when ('30') customer_activity='medium activity';
      when ('40') customer_activity='high activity';
    otherwise;
  end;
  end;
run;
```
Nested SELECT Statements

This program creates a permanent SAS data set named Retail.Customers by reading the Retail.Customer_hybrid data set. Nested SELECT statements are used to populate the variables Customer_Group and Customer_Activity.

data retail.customers;
  length Customer_Group $ 26 Customer_Activity $ 15;
  set retail.customer_hybrid;
  select(substr(put(customer_type_ID,4.),1,2));
    when ('10')
      do;
        customer_group='Orion Club members';
        select(substr(put(customer_type_ID,4.),3,2));
          when ('10') customer_activity='inactive';
          when ('20') customer_activity='low activity';
          when ('30') customer_activity='medium activity';
          when ('40') customer_activity='high activity';
          otherwise;
        end;
      end;
    when ('20')
      do;
        customer_group='Orion Club Gold members';
        select(substr(put(customer_type_ID,4.),3,2));
          when ('10') customer_activity='inactive';
          when ('20') customer_activity='low activity';
          when ('30') customer_activity='medium activity';
          when ('40') customer_activity='high activity';
          otherwise;
        end;
      end;
    when ('30')
      do;
        customer_group='Internet/Catalog Customers';
        otherwise;
      end;
  otherwise;
end;
run;
Serial IF Statements

This program creates a permanent SAS data set named *Retail.Customers* by reading the *Retail.Customer_hybrid* data set. Serial IF statements are used to populate the variables *Customer_Group* and *Customer_Activity*.

```sas
data retail.customers;
  length Customer_Group $ 26 Customer_Activity $ 15;
  set retail.customer_hybrid;
  if substr(put(customer_type_ID,4.),1,2)='10' then
    customer_group='Orion Club members';
  if substr(put(customer_type_ID,4.),1,2)='20' then
    customer_group='Orion Club Gold members';
  if substr(put(customer_type_ID,4.),1,2)='30' then
    customer_group='Internet/Catalog Customers';
  if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
    substr(put(customer_type_ID,4.),3,2)='10' then
    customer_activity='inactive';
  if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
    substr(put(customer_type_ID,4.),3,2)='20' then
    customer_activity='low activity';
  if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
    substr(put(customer_type_ID,4.),3,2)='30' then
    customer_activity='medium activity';
  if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
    substr(put(customer_type_ID,4.),3,2)='40' then
    customer_activity='high activity';
run;
```
IF-THEN/ELSE IF Statements with a Link

This program creates a permanent SAS data set named `Retail.Customers` by reading the `Retail.Customer_hybrid` data set. IF-THEN/ELSE IF statements are used with a link to populate the variables `Customer_Group` and `Customer_Activity`.

```sas
data retail.customers;
  length Customer_Group $ 26 Customer_Activity $ 15;
  set retail.customer_hybrid;
  if substr(put(customer_type_ID,4.),1,2)='10' then do;
    customer_group='Orion Club members';
    link activity;
  end;
  else if substr(put(customer_type_ID,4.),1,2)='20' then do;
    customer_group='Orion Club Gold members';
    link activity;
  end;
  else if substr(put(customer_type_ID,4.),1,2)='30' then do;
    customer_group='Internet/Catalog Customers';
    return;
  activity:
    if substr(put(customer_type_ID,4.),3,2)='10' then
      customer_activity='inactive';
    else if substr(put(customer_type_ID,4.),3,2)='20' then
      customer_activity='low activity';
    else if substr(put(customer_type_ID,4.),3,2)='30' then
      customer_activity='medium activity';
    else if substr(put(customer_type_ID,4.),3,2)='40' then
      customer_activity='high activity';
    return;
  run;
```

General Recommendations

- Avoid serial IF statements because they use extra resources.

Eliminating Unnecessary Passes through the Data

Best practices dictate that you should eliminate unnecessary passes through the data. To minimize I/O operations and CPU time, avoid reading or writing data more than necessary. Accomplish this by taking advantage of one-step processing, which can lead to efficiencies.

Using a Single DATA or PROC Step to Enhance Efficiency

Whenever possible, use a single DATA or PROC step to enhance efficiency. Techniques that minimize passes through the data include

- using a single DATA step to create multiple output data sets
- using the SORT procedure with a WHERE statement to create sorted subsets
- using the DATASETS procedure to modify variable attributes.

Before viewing comparative examples that address these techniques, it might be helpful to review the following information about these practices.
Using a Single DATA Step to Create Multiple Output Data Sets

It is good programming practice to take advantage of the DATA step’s ability to create multiple output data sets at the same time. This is more efficient than using a series of individual DATA steps. Using a single DATA step saves resources because input data is read only once.

The following program demonstrates how to use a single DATA step to read data once and create three subsets of data.

```sas
data Sales_managers Account_managers Finance_managers;
  set company.organization;
  if job_title='Sales Manager' then
    output Sales_managers;
  else if job_title='Account Manager' then
    output Account_managers;
  else if job_title='Finance Manager' then
    output Finance_managers;
run;
```

Using the SORT Procedure with a WHERE Statement to Create Sorted Subsets

It is good programming practice to take advantage of the SORT procedure’s ability to sort and subset in the same PROC step. This is more efficient than using two separate steps to accomplish this—a DATA step to subset followed by a procedure step that sorts.

The following program demonstrates how you can sort data and select only those observations that meet the conditions of the WHERE statement—in one step. Processing a data set only once saves I/O and CPU resources.

```sas
proc sort data=company.organization
  out=company.managers;
  by job_title;
  where job_title in('Sales Manager',
    'Account Manager',
    'Finance Manager');
run;
```

Using the DATASETS Procedure to Modify Variable Attributes

Use PROC DATASETS instead of a DATA step to modify data attributes. The DATASETS procedure uses fewer resources than the DATA step because it processes only the descriptor portion of the data set, not the data portion. PROC DATASETS retains the sort flag, as well as indexes.

```sas
proc datasets lib=company;
  modify organization;
  rename Job_title=Title;
quit;
```

*Note:* You cannot use the DATASETS procedure to modify the type, length, or position of variables because these attributes directly affect the data portion of the data set. To perform these operations, use the DATA step. ▲
Comparative Example: Creating Multiple Subsets of a SAS Data Set

Suppose you want to create five subsets of data from the data set Retail.Customer. You need a subset for each of five countries. Techniques for creating multiple subsets include writing

1. Multiple DATA Steps

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

Programming Techniques

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>retail.UnitedStates</td>
<td>Includes multiple DATA steps and subsequently reads data five times from the same Retail.Customer data set. Individual subsetting IF statements appear in five separate DATA steps.</td>
</tr>
<tr>
<td>retail.France</td>
<td></td>
</tr>
<tr>
<td>retail.Italy</td>
<td></td>
</tr>
<tr>
<td>retail.Germany</td>
<td></td>
</tr>
<tr>
<td>retail.Spain</td>
<td></td>
</tr>
</tbody>
</table>

```sas
data retail.UnitedStates;
  set retail.customer;
  if country='US';
run;

data retail.France;
  set retail.customer;
  if country='FR';
run;

data retail.Italy;
  set retail.customer;
  if country='IT';
run;

data retail.Germany;
  set retail.customer;
  if country='DE';
run;

data retail.Spain;
  set retail.customer;
  if country='ES';
run;
```
General Recommendations

□ When creating multiple subsets from a SAS data set, use a single DATA step with IF-THEN/ELSE IF logic to output to appropriate data sets.

Comparative Example: Creating a Sorted Subset of a SAS Data Set

Suppose you want to create a sorted subset of a SAS data set named Retail.Customer. You want only data for customers in the United States, France, Italy, Germany, and Spain.

Techniques for creating sorted subsets of SAS data sets include

1 A DATA Step and PROC SORT
2 PROC SORT with a WHERE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.
Programming Techniques

1. A DATA Step and PROC SORT
   This program has two steps. The first step creates a SAS data set by subsetting observations based on the value of the variable `Country`. The second step sorts the data according to the values for each country. Passing through all the data once and the subset twice can increase I/O and CPU operations.

   ```sas
   data retail.CountrySubset;
     set retail.customer;
     where country in('US','FR','IT','DE','ES');
   run;
   
   proc sort data=retail.CountrySubset;
     by country;
   run;
   ```

2. PROC SORT with a WHERE Statement
   In one step, this program sorts data and selects only those observations that meet the conditions of the WHERE statement. Processing only one data set once saves CPU and I/O resources.

   ```sas
   proc sort data=retail.customer out=retail.CountrySubset;
     by country;
     where country in('US','FR','IT','DE','ES');
   run;
   ```

General Recommendations

- When you need to process a subset of data with a procedure, use a WHERE statement in the procedure instead of creating a subset of data and reading that data with the procedure.
- Write one program step that both sorts and subsets. This approach can take less programmer time and debugging time than writing separate program steps that subset and sort.

Comparative Example: Changing the Variable Attributes of a SAS Data Set

Suppose you want to change the variable attributes in `Retail.NewCustomer` to make them consistent with those in the `Retail.Customer` data set. The data set `Retail.NewCustomer` contains 89954 observations and 12 variables. The following table shows the variable names and formats in each SAS data set.
Techniques for changing the variable attributes of a SAS data set include

1. A DATA Step
2. PROC DATASETS.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

**Programming Techniques**

1. **A DATA Step**
   
   This program uses a DATA step with a RENAME statement and a FORMAT statement to modify attributes for the variables `Country_ID` and `Birth_Date`.
   ```sas
   data retail.newcustomer;
   set retail.newcustomer;
   rename Country_ID=country;
   format birth_date date9.;
   run;
   ```

2. **PROC DATASETS**
   
   This program uses PROC DATASETS to modify the names and formats of the variables `Country_ID` and `Birth_Date`.
   ```sas
   proc datasets lib=retail;
   modify newcustomer;
   rename Country_ID=country;
   format birth_date date9.;
   quit;
   ```

**General Recommendations**

- To save significant resources, use the DATASETS procedure instead of a DATA step to change the attributes of a SAS data set.

**Reading and Writing Only Essential Data**

Best practices dictate that you should write programs that read and write only essential data. If you process fewer observations and variables, you conserve resources. This topic covers many different techniques that can improve performance when you...
Selecting Observations Using Subsetting IF versus WHERE Statement

You can use WHERE statements or subsetting IF statements to subset data. Although both statements test a condition to determine whether SAS should process an observation, the WHERE statement is more efficient.

The following graphic illustrates differences in how these statements process data.

I/O operations are measured as data moves between the disk that contains input SAS data and the buffer in memory, and when data moves from the output buffer to the disk that contains output data sets. Input data is not affected by the WHERE statement or subsetting IF statement. However, output data is affected by both.

CPU time is measured when data must be processed in the program data vector. CPU time can be saved if fewer observations are processed.

A WHERE statement and an IF statement make different use of the program data vector. The WHERE statement examines what is in the input page buffer and selects observations before they are loaded in the program data vector, which results in a savings in CPU operations. The subsetting IF statement loads all observations sequentially into the program data vector. If the statement finds a match and the statement is true, then the data is processed and is written to the output page buffer.

WHERE statements work on variables that are already in SAS data sets. IF statements can work on any variable in the program data vector, including new or old variables.

Comparative Example: Creating a Subset of a SAS Data Set

Suppose you want to create a subset of the data set Retail.Customer. You want to include data for only the United Kingdom. The subset contains approximately 5.56% of the Retail.Customer data.

Techniques for subsetting observations include

- select observations from SAS data sets
- select observations from external files
- keep or drop variables when creating or reporting on SAS data sets.

Let’s begin by looking at how two useful statements differ in how they select observations to subset.
1. Subsetting IF Statement

2. WHERE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

**Programming Techniques**

|   | Subsetting IF Statement
|---|---|
| 1 | This program uses the IF statement to select observations if the value for Country is GB.  
|   | data retail.UnitedKingdom;  
|   | set retail.customer;  
|   | if country='GB';  
|   | run;  

|   | WHERE Statement
|---|---|
| 2 | This program uses the WHERE statement to select observations when the value for Country is GB. This is more efficient than using a subsetting IF statement.  
|   | data retail.UnitedKingdom;  
|   | set retail.customer;  
|   | where country='GB';  
|   | run;  

**General Recommendations**

- To save CPU resources, use a WHERE statement instead of a subsetting IF statement to subset a SAS data set.

**Other Differences between the IF and WHERE Statements**

Review the following table to note other differences between the IF and WHERE statements.

<table>
<thead>
<tr>
<th></th>
<th>The IF Statement</th>
<th>The WHERE Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selecting Data</td>
<td>can select records from external files, observations from SAS data sets, observations created with an INPUT statement, or observations based on the value of a computed or derived variable.</td>
<td>can only select observations from SAS data sets.</td>
</tr>
<tr>
<td>Conditional Execution</td>
<td>can be executed conditionally as part of an IF statement.</td>
<td>cannot be executed conditionally as part of an IF statement.</td>
</tr>
</tbody>
</table>
Using the WHERE Statement with the OBS= and FIRSTOBS= Options

Another way to read and write only essential data is to process a segment of subsetted data. You accomplish this specialized task by using a WHERE expression in conjunction with the OBS= and FIRSTOBS= data set options. This programming feature is available in SAS 8.1 and later.

In the following example, the WHERE expression selects observations before the OBS= and FIRSTOBS= options are applied. The values specified for OBS= and FIRSTOBS= are the logical observation numbers in the subset, not the physical observation numbers in the data set.

```
proc print data=company.organization(firstobs=5 obs=8);
var employee_id employee_gender salary;
where salary>40000;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee_ID</th>
<th>Employee_Gender</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>120201</td>
<td>Male</td>
<td>$43,290</td>
</tr>
<tr>
<td>157</td>
<td>120257</td>
<td>Female</td>
<td>$156,245</td>
</tr>
<tr>
<td>158</td>
<td>120268</td>
<td>Male</td>
<td>$85,305</td>
</tr>
<tr>
<td>159</td>
<td>120259</td>
<td>Male</td>
<td>$432,800</td>
</tr>
</tbody>
</table>

FIRSTOBS = 5 is the fifth observation in the subset, whereas it was observation 101 in the data set Company.Staff.

OBS = 8 is the eighth observation in the subset, whereas it was observation 159 in the data set Company.Staff.

Now that you’ve seen techniques for efficiently subsetting observations that are read from SAS data sets, let’s look at techniques for subsetting records that are read from external files.

Before viewing a comparative example that illustrates these techniques, it might be useful to review which resources are affected as SAS reads and processes data that is read from external files.
Selecting Observations When Reading Data from External Files

Positioning a subsetting IF statement in a DATA step so that it reads only the variables that are needed to select the subset—before reading all the data—can reduce the overhead required for processing data.

The following graphic illustrates how data is read from an external file, loaded into the input buffer, and read into the program data vector.

Remember that I/O operations are measured as data moves between disks and buffers—for both input and output data. All records are loaded into the input buffer before moving to the program data vector for processing, so I/O is not affected by the placement of a subsetting IF statement in the DATA step.

You can reduce the CPU resources that are required for processing data by limiting what is read into the program data vector. Position a subsetting IF statement after an INPUT statement that reads only the data that is required in order to check for specific conditions. Subsequent statements do not execute and do not process variable values for unwanted observations.

Note: Converting raw character fields to SAS character variables requires less CPU time than converting raw numeric fields to the real binary encoding of SAS numeric variables.

Comparative Example: Creating a Subset of Data by Reading Data from an External File

Suppose you want to create a SAS data set by reading a subset of data from an external file that is referenced by the fileref Customerdata. You want the subset to contain only customers in the United Kingdom.

The subset is approximately 5.56% of the countries in the external file, which contains 89,954 records and 12 fields.

Techniques for doing this include

1. Reading All Variables and Subsetting
2. Reading Selected Variables and Subsetting.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your
results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

**Programming Techniques**

1. **Reading All Variables and Subsetting**

   In this program, the INPUT statement reads the values for all variables before the subsetting IF statement checks for the value of `Country`. Then, if the value for `Country` is GB, the observation is written to the output data set `Retail.UnitedKingdom`.

   ```
   data retail.UnitedKingdom;
   infile customerdata;
   input @1 Customer_ID 12.
   @13 Country $2.
   @15 Gender $1.
   @16 Personal_ID $15.
   @31 Customer_Name $40.
   @71 Customer_FirstName $20.
   @91 Customer_LastName $30.
   @121 Birth_Date date9.
   @130 Customer_Address $45.
   @175 Street_ID12.
   @199 Street_Number $8.
   @207 Customer_Type_ID 8.;
   if country='GB';
   run;
   ```

2. **Reading Selected Variables and Subsetting**

   In this program, the first INPUT statement reads only the value for `Country` and holds the record in the input buffer using the single trailing @ sign. Then the program uses a subsetting IF statement to check for the value of `Country`. If the value for `Country` is not GB, values for other variables are not read in or written to the output data set `Retail.UnitedKingdom`. If the value for `Country` is GB, values for other variables are input and written to the output data set `Retail.UnitedKingdom`.

   ```
   data retail.UnitedKingdom;
   infile customerdata;
   input @13 Country $2. @;
   if country='GB';
   input @1 Customer_ID12.
   @15 Gender $1.
   @16 Personal_ID $15.
   @31 Customer_Name $40.
   @71 Customer_FirstName $20.
   @91 Customer_LastName $30.
   @121 Birth_Date date9.
   @130 Customer_Address $45.
   @175 Street_ID12.
   @199 Street_Number $8.
   @207 Customer_Type_ID 8.;
   run;
   ```

**General Recommendations**
Position a subsetting IF statement in a DATA step so that only variables that are necessary to select the record are read before subsetting. This can result in significant savings in CPU time. There is no difference in I/O or memory usage between the two techniques.

When selecting rows of data from an external file, read the field(s) on which the selection is being made before reading all the fields into the program data vector.

Use the single trailing @ sign to hold the input buffer so that you can continue to read the record when the variable(s) satisfy the IF condition.

In addition to subsetting observations, you can subset variables by using statements or options that efficiently read and write only essential data.

Before viewing two comparative examples that illustrate how to best limit which variables are read and processed, let's review how these useful statements and options work.

### Subsetting Variables with the KEEP= and DROP= Statements and Options

To subset variables, you can use

- the DROP and KEEP statements
- the DROP= and KEEP= data set options.

**Input Data Set**

<table>
<thead>
<tr>
<th>Discount</th>
<th>Total_Retail</th>
<th>Price</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Output Data Set**

<table>
<thead>
<tr>
<th>Price</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** You cannot use the DROP and KEEP statements and the DROP= and KEEP= output data set options in the same step. △

Use of the KEEP= data set option and the DROP= data set option can affect resource usage, depending on whether they are used in a SET or MERGE statement or in a DATA statement.

The following figure shows how options in these statements process data.
When used in the SET or MERGE statement, the KEEP= and DROP= data set options affect which variables are read into the program data vector. Reading only the variables that need to be processed in the DATA step can improve efficiency.

When used in the DATA statement, these same options put drop flags on variables to be excluded and affect which variables are written to the output data set. The DROP and KEEP statements work just like the KEEP= or DROP= options in the DATA statement.

The following table describes differences in how the KEEP statement and the KEEP= data set option write variables to SAS data sets.

<table>
<thead>
<tr>
<th>KEEP Statement</th>
<th>KEEP= Output Data Set Option</th>
<th>KEEP= Input Data Set Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causes a DATA step to write only the variables listed in the KEEP statement to one or more data sets.</td>
<td>Causes a DATA step to write only the variables listed in the KEEP= variable list to the output data set.</td>
<td>In the absence of a KEEP=/DROP= output data set option, causes the DATA step to write only the variables listed in the KEEP= variable list.</td>
</tr>
<tr>
<td>In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.</td>
<td>In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.</td>
<td>Enables processing of only the variables listed in the KEEP= variable list.</td>
</tr>
<tr>
<td>Applies to all data sets that are created within the same DATA step.</td>
<td>Can write different variables to different data sets.</td>
<td>In the absence of a KEEP=/DROP= output data set option or KEEP/DROP statement, enables processing of only the variables listed in the KEEP= variable list.</td>
</tr>
</tbody>
</table>

Available only in the DATA step. | Available in the DATA step or in a PROC step. | Available in the DATA step or in a PROC step. |
The following table describes differences in how the DROP statement and the DROP= data set option write variables to SAS data sets.

<table>
<thead>
<tr>
<th>DROP Statement</th>
<th>DROP= Output Data Set Option</th>
<th>DROP= Input Data Set Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causes a DATA step to write only the variables not listed in the DROP statement to one or more data sets.</td>
<td>Causes a DATA step to write only the variables not listed in the DROP= variable list to the output data set.</td>
<td>In the absence of a KEEP=/DROP= output data set option, enables processing of only the variables not listed in the DROP= variable list.</td>
</tr>
<tr>
<td>In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.</td>
<td>In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.</td>
<td>Enables processing of only the variables not listed in the DROP= variable list.</td>
</tr>
<tr>
<td>Applies to all data sets created within the same DATA step.</td>
<td>Can write different variables to different data sets.</td>
<td>In the absence of a KEEP=/DROP= output data set option or KEEP/DROP statement, enables processing of only the variables not listed in the DROP= variable list.</td>
</tr>
</tbody>
</table>

Comparative Example: Creating a Report That Contains Average and Median Statistics

Suppose you want to create a report that contains the average and median values for the variable Profit, based on data that is read from the data set Retail.Order_fact. Depending on the number of variables eliminated, it might be more efficient to use the KEEP= option in a SET statement to limit which variables are read.

Techniques for reading and writing variables to a data set include

1. Without the KEEP= Statement
2. KEEP= in the DATA Statement
3. KEEP= in the DATA and SET Statements
4. KEEP= in the SET and MEANS Statements.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

Programming Techniques
Comparative Example: Creating a Report That Contains Average and Median Statistics

Chapter 22

1. Without the KEEP= Statement

This program reads all variables from the data set Retail.Order_fact and does not restrict which variables are written to the output data set Retail.Profit. PROC MEANS reads all the variables from the data set.

```plaintext
data retail.profit;
  set retail.order_fact;
  if discount=. then
    Profit=(total_retail_price-costPrice_Per_Unit)*quantity;
  else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
title 'Order Information';
class employee_id;
var profit;
run;
```

2. KEEP= in the DATA Statement

This program uses the KEEP= data set option in the DATA statement to write two variables to the output data set Retail.Profit. PROC MEANS reads only two variables from the data set.

```plaintext
data retail.profit(keep=employee_id profit);
  set retail.order_fact;
  if discount=. then
    Profit=(total_retail_price-costPrice_Per_Unit)*quantity;
  else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
title 'Order Information';
class employee_id;
var profit;
run;
```

3. KEEP= in the DATA and SET Statements

This program uses the KEEP= option in the SET statement to read six variables from Retail.Order_fact, and it uses the KEEP= data set option in the DATA statement to write two variables to the output data set Retail.Profits. PROC MEANS reads only two variables from the data set.

```plaintext
data retail.profits(keep=employee_id profit);
  set retail.order_fact(keep=employee_id total_retail_price discount costprice_per_unit quantity);
  if discount=. then
    Profit=(total_retail_price-costPrice_Per_Unit)*quantity;
  else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
title 'Order Information';
class employee_id;
var profit;
run;
```
**KEEP= in the SET and MEANS Statements**

This program uses the KEEP= option in the SET statement to read selected variables from *Retail.Order_fact*, and it uses the KEEP= data set option in the MEANS statement to process only the variables that are needed for the statistical report. You might do this if you need additional variables in *Retail.Profits* for further processing, but only two variables for processing by PROC MEANS.

```sas
data retail.profit;
  set retail.order_fact(keep=employee_id total_retail_price discount costprice_per_unit quantity);
  if discount=. then
    Profit=(total_retail_price-costprice_per_unit)*quantity;
  else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit(keep=employee_id profit) mean median maxdec=2;
  title 'Order Information';
  class employee_id;
  var profit;
run;
```

**General Recommendations**

- To reduce both CPU time and I/O operations, avoid reading and writing variables that are not needed.

**Comparative Example: Creating a SAS Data Set That Contains Only Certain Variables**

Suppose you want to read data from an external file that is referenced by the fileref *Rawdata* and to create a SAS data set that contains only the variables *Customer_ID*, *Country*, *Gender*, and *Customer_Name*.

Techniques for accomplishing this task include

1. Reading All Fields
2. Reading Selected Fields.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

**Programming Techniques**
Reading All Fields

In this program, the KEEP= data set option writes only the variables that are needed to the output data set, whereas the INPUT statement reads all fields from the external file.

```sas
data retail_customers(keep=Customer_ID Country Gender Customer_Name);
infile rawdata;
input @1 Customer_ID 12.
@13 Country $2.
@15 Gender $1.
@16 Personal_ID $15.
@31 Customer_Name $40.
@71 Customer_FirstName $20.
@91 Customer_LastName $30.
@121 Birth_Date date9.
@130 Customer_Address $45.
@175 Street_ID 12.
@199 Street_Number $8.
@207 Customer_Type_ID 8.;
run;
```

Reading Selected Fields

In this program, the INPUT statement reads selected fields from the external file, and by default, these are written to the output data set.

```sas
data retail_customers;
infile rawdata;
infile rawdata;
input @1 Customer_ID 12.
@13 Country $2.
@15 Gender $1.
@31 Customer_Name $40.;
run;
```

General Recommendations

- When possible, read only the fields you need from an external data file to save CPU and real-time resources.
- To save CPU resources, avoid converting numerics that you do not need in further processing.

Note: Remember that numeric data is moved into the program data vector after being converted to real binary, floating point numbers; multiple digits are stored in one byte. Character data is moved into the program data vector with no conversion; one character is stored in one byte.

Storing Data in SAS Data Sets

In many cases, it is best practice for you to store data in SAS data sets. You can optimize performance if you know when you should create a SAS data set and when you should read data directly from an external file.

Before viewing the comparative example that illustrates different techniques for reading from a SAS data set versus from an external file, consider the following advantages of storing data in SAS data sets.
When you use SAS to repeatedly analyze or manipulate any particular group of data, it is more efficient to create a SAS data set than to read the raw data each time. Although SAS data sets can be larger than external files and can require more disk space, reading from SAS data sets saves CPU time that is associated with reading a raw data file.

Other reasons for storing data in SAS data sets, rather than external files, include:
- When the data is already in a SAS data set, you can use a SAS procedure, function, or routine on the data without further conversion.
- SAS data sets are self-documenting.

The descriptor portion of a SAS data set documents information about the data set such as:
- data set labels
- variable labels
- variable formats
- informats
- descriptive variable names.

*Note:* Create a temporary SAS data set if the data set is used for intermediate tasks such as merging and if it is needed in that SAS session only. Create a temporary SAS data set when the external file on which the data set is based might change between SAS sessions. △

---

**Comparative Example: Creating a SAS Data Set or Reading from an External File**

Suppose you want to create a SAS data set that contains a large number of variables. One way to accomplish this task is to read from raw data fields in an external file that is referenced by the fileref `Rawdata`. Another way to accomplish this is to read the same data values from an existing SAS data set named `Retail.Customer`.

Techniques for accomplishing this task include
1. Reading from an External File
2. Reading from a SAS Data Set.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your
results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for storing data in SAS data sets.

**Programming Techniques**

1. **Reading from an External File**

   In this program, the INPUT statement reads fields of data from an external file that is referenced by the fileref `Rawdata` and creates 12 variables. For benchmarking purposes, the DATA statement creates a `_NULL_` data set, testing for the effects of the reading operation rather than the output processing.

   ```sas
   data _null_;
   infile rawdata;
   input @1 Customer_ID 12.
   @13 Country $2.
   @15 Gender $1.
   @16 Personal_ID $15.
   @31 Customer_Name $40.
   @71 Customer_FirstName $20.
   @91 Customer_LastName $30.
   @121 Birth_Date date9.
   @130 Customer_Address $45.
   @175 Street_ID 12.
   @199 Street_Number $8.
   @207 Customer_Type_ID 8.;
   run;
   ```

2. **Reading from a SAS Data Set**

   In this program, the SET statement reads data directly from an existing SAS data set. As in the previous program, the DATA statement uses `_NULL_` instead of naming a data set.

   ```sas
   data _null_;
   set retail.customer;
   run;
   ```

**General Recommendations**

- To save CPU resources, you can read variables from a SAS data set instead of reading data from an external file.
- To reduce I/O operations, you can read variables from a SAS data set instead of reading data from an external file. However, savings in I/O operations are largely dependent on the block size of the external data file and on the page size of the SAS data set.

**Avoiding Unnecessary Procedure Invocation**

Best practices dictate that you avoid unnecessary procedure invocation. One way to do this is to take advantage of procedures that accomplish multiple tasks with one invocation.
Several procedures enable you to create multiple reports by invoking the procedure only once. These include:
- the SQL procedure
- the DATASETS procedure
- the FREQ procedure
- the TABULATE procedure.

*Note:* BY-group processing can also minimize unnecessary invocations of procedures.

To illustrate this principle, let’s examine features of the DATASETS procedure.

### Executing the DATASETS Procedure

The DATASETS procedure can use RUN-group processing to process multiple sets of statements. RUN-group processing enables you to submit groups of statements without ending the procedure.

When the DATASETS procedure executes,
- SAS reads the program statements that are associated with one task until it reaches a RUN statement or an implied RUN statement.
- SAS executes all of the preceding statements immediately, and then continues reading until it reaches another RUN statement or an implied RUN statement.

To execute the last task, you must use a RUN statement or a QUIT statement.

```sas
proc datasets lib=company;
   modify orders;
      rename quantity=Units_Ordered;
      format costprice_per_unit dollar13.2;
      label delivery_date='Date of Delivery';
   run;
   modify customers;
      format customer_birthdate mmddyy10.
   run;
quit;
```

You can terminate the PROC DATASETS execution by submitting
- a DATA statement
- a PROC statement
- a QUIT statement.

### RUN-Group Processing

If you can take advantage of RUN-group processing, you can avoid unnecessary procedure invocation. For best programming practices, you need to understand how RUN-group processing affects the execution of SAS statements. The procedures that support RUN-group processing include
- CHART, GCHART
- PLOT, GPLOT
- GIS, GMAP
- GLM
- REG
Using Different Types of RUN Groups with PROC DATASETS

To illustrate how RUN-group processing works, this discussion focuses on the DATASETS procedure. The comparative example that follows includes programs that use PROC DATASETS to modify the descriptor portion of data sets. Before you examine the code to consider efficient programming techniques, review how the principles associated with RUN-group processing apply to PROC DATASETS.

The DATASETS procedure supports four types of RUN groups. Each RUN group is defined by the statements that compose it and by what causes it to execute.

Some statements in PROC DATASETS act as implied RUN statements because they cause the RUN group that precedes them to execute.

The following list identifies which statements compose a RUN group and what causes each RUN group to execute:

- The PROC DATASETS statement always executes immediately. No other statement is necessary to cause the PROC DATASETS statement to execute. Therefore, the PROC DATASETS statement alone is a RUN group.
- The MODIFY statement and any of its subordinate statements form a RUN group. These RUN groups always execute immediately. No other statement is necessary to cause a MODIFY RUN group to execute.
- The APPEND, CONTENTS, and COPY statements (including EXCLUDE and SELECT, if present) form their own separate RUN groups. Every APPEND statement forms a single-statement RUN group; every CONTENTS statement forms a single-statement RUN group; and every COPY step forms a RUN group. Any other statement in the procedure, except those that are subordinate to either the COPY or MODIFY statement, causes the RUN group to execute.

Additionally, one or more of the following statements form a RUN group:

- AGE
- EXCHANGE
- CHANGE
- REPAIR.

If any of these statements appear in sequence in the PROC step, the sequence forms a RUN group. For example, if a REPAIR statement appears immediately after a SAVE statement, the REPAIR statement does not force the SAVE statement to execute; it becomes part of the same RUN group. To execute the RUN group, submit one of the following statements:

- PROC DATASETS
- MODIFY
- APPEND
- QUIT
- CONTENTS
- RUN
- COPY
- another DATA or PROC step.

Comparative Example: Modifying the Descriptor Portion of SAS Data Sets

Suppose you want to use the DATASETS procedure to modify the data sets NewCustomer, NewOrders, and NewItems.
Techniques for accomplishing this task include using

1 Multiple DATASETS Procedures
2 A Single DATASETS Procedure.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for avoiding unnecessary procedure invocation.

**Programming Techniques**

<table>
<thead>
<tr>
<th>Multiple DATASETS Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>This program invokes PROC DATASETS three times to modify the descriptor portion of the data set NewCustomer, two times to modify the descriptor portion of the data set NewOrders, and once to change the name of the data set NewItems.</td>
</tr>
</tbody>
</table>

```sas
proc datasets lib=company;
   modify newcustomer;
   rename Country_ID=Country
           Name=Customer_Name;
quit;

proc datasets lib=company;
   modify newcustomer;
   format birth_date date9.;
quit;

proc datasets lib=company;
   modify newcustomer;
   label birth_date='Date of Birth';
quit;

proc datasets lib=company;
   modify neworders;
   rename order=Order_ID
           employee=Employee_ID
           customer=Customer_ID;
quit;

proc datasets lib=company;
   modify neworders;
   format order_date date9.;
quit;

proc datasets lib=company;
   change newitems=NewOrder_Items;
quit;
```
Single DATASETS Procedure
This program invokes PROC DATASETS once to modify the descriptor portion of the data sets
*NewCustomer* and *NewOrders*, and to change the name of the data set *NewItems*. This technique is
more efficient.

```plaintext
proc datasets lib=company;
   modify newcustomer;
      rename country_ID=Country
           name=Customer_Name;
      format birth_date date9.;
      label birth_date='Date of Birth';
   modify neworders;
      rename order=Order_ID
           employee=Employee_ID
           customer=Customer_ID;
      format order_date date9.;
   change newitems=NewOrder_Items;
quit;
```

General Recommendations

- Invoke the DATASETS procedure once and process all the changes for a library in
  one step to save CPU and I/O resources—at the cost of memory resources.

  *Note:* Because the specified library could change between invocations of the
  DATASETS procedure, the procedure is reloaded into memory for each invocation.
Summary

Executing Only Necessary Statements

You minimize the CPU time that SAS uses when you execute the minimum number of statements in the most efficient order.

For a more efficient program, place the subsetting IF statement as soon as logically possible in a DATA step when creating a subset of data.

Review guidelines for using conditional logic efficiently with IF-THEN/ELSE statements or SELECT statements. Remember to minimize the number of statements that use SAS functions or arithmetic operators.

Conditionally execute only necessary statements by placing statements in groups that are associated with IF-THEN/ELSE statements or SELECT/WHEN statements. Groups of statements execute only when a particular condition is true. Review the criteria for using DO groups efficiently.

Review the related comparative examples:

- “Comparative Example: Creating a Subset of Data” on page 744
- “Comparative Example: Creating Variables Conditionally Using DO Groups” on page 747
- “Comparative Example: Creating Variables Conditionally When Calling Functions” on page 749
- “Comparative Example: Creating Data in DO Groups” on page 754.

Eliminating Unnecessary Passes Through the Data

You should avoid reading or writing data more than necessary in order to minimize I/O operations.

There are a variety of techniques that you can use. For example, use a single DATA step to create multiple output data sets from one pass of the input data, rather than using multiple DATA steps to process the input data each time that you create an output data set. Create sorted subsets by subsetting data with the SORT procedure rather than subsetting data in a DATA step and then sorting. Change variable attributes by using PROC DATASETS rather than a DATA step.

Review the related comparative examples:

- “Comparative Example: Creating Multiple Subsets of a SAS Data Set” on page 760
- “Comparative Example: Creating a Sorted Subset of a SAS Data Set” on page 761
- “Comparative Example: Changing the Variable Attributes of a SAS Data Set” on page 762.

Reading and Writing Only Essential Data

If you process fewer observations and variables, SAS performs fewer I/O operations. To limit the number of observations that are processed, you can use the subsetting IF statement and the WHERE statement. Best programming practices can be applied if you understand other differences between subsetting IF and WHERE statements. You can also improve performance by applying OBS= and FIRSTOBS= processing with a WHERE statement.

To select observations when reading data from external files, position a subsetting IF statement in a DATA step so that it reads only the variables that are needed to select the subset before reading all the data. This can reduce the overhead required to process data.
To limit the number of variables that are processed, you can use
- the DROP and KEEP statements
- the DROP= and KEEP= data set options.

In the SET statement, the DROP= or KEEP= data set option controls which variables are read and subsequently processed. In the DATA statement, the DROP= or KEEP= data set option controls which variables are written to a data set after processing. Using the SET statement with these options is the most efficient and best practice.

Review the related comparative examples:
- “Comparative Example: Creating a Subset of a SAS Data Set” on page 764
- “Comparative Example: Creating a Subset of Data by Reading Data from an External File” on page 767
- “Comparative Example: Creating a Report That Contains Average and Median Statistics” on page 771
- “Comparative Example: Creating a SAS Data Set That Contains Only Certain Variables” on page 773.

Storing SAS Data in SAS Data Sets

When you use SAS to repeatedly analyze or manipulate any particular group of data, create a SAS data set instead of reading the raw data each time.

Reading data from an external file versus reading from a SAS data set greatly increases CPU usage.

Review the related comparative example:
- “Comparative Example: Creating a SAS Data Set or Reading from an External File” on page 775.

Avoiding Unnecessary Procedure Invocation

Invoking procedures once rather than multiple times can be the most efficient way to process data. Several procedures enable you to create multiple reports by invoking the procedure only once.

Using a single DATASETS procedure instead of multiple DATASETS procedures to modify the descriptor portion of a data set results in a noticeable savings in both CPU and I/O operations. Also, you can take advantage of RUN-group processing to submit groups of statements without ending the procedure.

Review the related comparative example:
- “Comparative Example: Modifying the Descriptor Portion of SAS Data Sets” on page 778.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1 Placing the subsetting IF statement at the top rather than near the bottom of a DATA step results in a savings in CPU usage. What happens if the subset is large rather than small?
   - a The savings in CPU usage increases as the subset grows larger because the I/O increases.
   - b The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement at the top of a DATA step always uses less resources than placing it at the bottom.
c The savings in CPU usage remains constant as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set is preferable.
d The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set increases the I/O.

2 Which of the following statements is true about techniques that are used for modifying data and attributes?
   a You can use PROC DATASETS to modify both data values and variable attributes.
   b You can use PROC DATASETS to modify only data values.
   c You can use the DATA step to modify both data values and variable attributes.
   d You can use the DATA step to modify only variable attributes.

3 For selecting observations, is a subsetting IF statement or a WHERE statement more efficient? Why?
   a A subsetting IF statement is more efficient because it loads all observations sequentially into the program data vector.
   b A subsetting IF statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.
   c A WHERE statement is more efficient because it loads all observations sequentially into the program data vector.
   d A WHERE statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.

4 When is it more advantageous to create a temporary SAS data set rather than a permanent SAS data set?
   a When the external file on which the data set is based might change between SAS sessions.
   b When the external file on which the data set is based does not change between SAS sessions.
   c When the data set is needed for more than one SAS session.
   d When you are converting raw numeric values to SAS data values.

5 When you compare the technique of using multiple DATASETS procedures to using a single DATASETS procedure to modify the descriptor portion of a data set, which is true?
   a A one-step DATASETS procedure results in an increase in I/O operations.
   b Multiple DATASETS procedures result in a decrease in I/O operations.
   c A one-step DATASETS procedure results in a decrease in CPU usage.
   d Multiple DATASETS procedures result in a decrease in CPU usage.
CHAPTER

23

Selecting Efficient Sorting Strategies

Overview 786
Introduction 786
Objectives 787
Prerequisites 787
Avoiding Unnecessary Sorts 787
  Using BY-Group Processing with an Index 788
  Comparative Example: Using BY-Group Processing with an Index to Avoid a Sort 788
    Programming Techniques 789
    General Recommendations 789
  Using the NOTSORTED Option 789
    Example 790
  Using FIRST. and LAST. 792
    Example 792
  Using the GROUPFORMAT Option 793
    Example 794
  Using the CLASS Statement 794
    Example 795
  Comparative Example: Using a BY or CLASS Statement to Avoid a Sort 797
    Programming Techniques 797
    General Recommendations 798
  Using the SORTEDBY= Data Set Option 798
    Example 800
Using a Threaded Sort 800
  Using the CPUCOUNT= System Option 801
Calculating and Allocating Sort Resources 802
  Sort Space Requirements 802
  Example 802
  Using the SORTSIZE= Option 804
Handling Large Data Sets 805
  Dividing a Large Data Set 805
  Comparative Example: Dividing and Sorting a Large Data Set 1 805
    Programming Techniques 806
    General Recommendations 810
  Comparative Example: Dividing and Sorting a Large Data Set 2 810
    Programming Techniques 810
    General Recommendations 815
  Using the TAGSORT Option 815
    Example 816
Removing Duplicate Observations Efficiently 816
  Using the NODUPKEY Option 817
    Example 817
Introduction

Sometimes you need to group observations by the values of a particular variable or order the observations in a particular way, such as alphabetically, in order to

- reorder the data for reporting
- reduce data retrieval time
- enable BY-group processing in both DATA and PROC steps.

The SORT procedure is one technique that can be used to group or order data. However, the SORT procedure can use a high volume of resources. When an uncompressed data file is sorted using the SORT procedure, SAS requires enough space in the data library for two copies of the data file, plus a workspace that is approximately two to four times the size of the data file.
In some cases, you might be able to use techniques other than the SORT procedure to group or order observations. In other cases, you might be able to use options or techniques with the SORT procedure that enable you to minimize resource usage.

Note: This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SAS SPD Engine to improve performance, see the SAS documentation.

Objectives

In this chapter, you learn to
- apply techniques that enable you to avoid unnecessary sorts
- calculate and allocate sort resources
- use strategies for sorting large data sets
- eliminate duplicate observations efficiently.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

Part 1: SQL Processing with SAS
- Chapter 1, “Performing Queries Using PROC SQL,” on page 3
- Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
- Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
- Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125
- Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159
- Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221
- Chapter 7, “Creating and Managing Views Using PROC SQL,” on page 243
- Chapter 8, “Managing Processing Using PROC SQL,” on page 261.

Part 3: Advanced SAS Programming Techniques
- Chapter 13, “Creating Samples and Indexes,” on page 451
- Chapter 14, “Combining Data Vertically,” on page 481
- Chapter 15, “Combining Data Horizontally,” on page 513
- Chapter 16, “Using Lookup Tables to Match Data,” on page 559
- Chapter 17, “Formatting Data,” on page 603

Part 4: Optimizing SAS Programs
- Chapter 19, “Introduction to Efficient SAS Programming,” on page 679
- Chapter 20, “Controlling Memory Usage,” on page 689
- Chapter 21, “Controlling Data Storage Space,” on page 707

Avoiding Unnecessary Sorts

In some cases you can avoid a sort by using
- BY-group processing with an index
- BY-group processing with the NOTSORTED option
Using BY-Group Processing with an Index

BY-group processing is a method of processing observations from one or more SAS data sets that are grouped or ordered by the values of one or more common variables. You can use BY-group processing in both DATA steps and PROC steps.

The most common use of BY-group processing in the DATA step is to combine two or more SAS data sets by using the BY statement with a SET, MERGE, UPDATE, or MODIFY statement. When you use a SET, MERGE, or UPDATE statement, the data sets must first be ordered on the values of the BY variable unless you index the data sets. You can also use the NOTSORTED option in the BY statement with a SET statement.

When BY-group processing is used with an index that is based on one of the BY variables, the data can be sequenced without using the SORT procedure. The data can be sequenced by different variables if multiple indexes are used. Because indexes are updated automatically, there is no need to re-sort a data set when observations are modified or added.

However, using BY-group processing with an index has two disadvantages:

- It is generally less efficient than sequentially reading a sorted data set because processing BY groups typically means retrieving the entire file.
- It requires storage space for the index.

Note: A BY statement does not use an index if the BY statement includes the DESCENDING or NOTSORTED option or if SAS detects that the data file is physically stored in sorted order on the BY variables.

Note: If you use a MODIFY statement, the data does not need to be ordered. However, your program might run more efficiently with ordered data.

Comparative Example: Using BY-Group Processing with an Index to Avoid a Sort

Suppose you want to use an existing data set, Retail.Order_fact, to create a new SAS data set that is ordered by the variable Order_Date. You could accomplish this task using

1. BY-Group Processing with an Index, Data in Random Order
2. Presorted Data in a DATA Step
3. PROC SORT Followed by a DATA Step.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment.
results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for BY-group processing with an index or sort.

**Programming Techniques**

1. **BY-Group Processing with an Index, Data in Random Order**
   In this example, the SAS data set `Retail.Order_fact` is indexed on the variable `Order_Date`. The data in `Retail.Order_fact` is in random order.
   ```sas
   data _null_;
   set retail.order_fact;
   by order_date;
   run;
   ```

2. **Presorted Data in a DATA Step**
   In this example, the SAS data set `Retail.Order_fact` is sorted on the variable `Order_Date` before it is read using the DATA step.
   ```sas
   data _null_;
   set retail.order_fact;
   by order_date;
   run;
   ```

3. **PROC SORT Followed by a DATA STEP**
   In this example, the SAS data set `Retail.Order_fact` is sorted using the SORT procedure. The data is then read using the DATA step.
   ```sas
   proc sort data=retail.order_fact;
   by order_date;
   run;
   data _null_;
   set retail.order_fact;
   by order_date;
   run;
   ```

**General Recommendations**

- To conserve resources, use sort order rather than an index for BY-group processing.
- Although using an index for BY-group processing is less efficient than using sort order, it might be the best choice if resource limitations make sorting a file difficult.

**Using the NOTSORTED Option**

You can also use the `NOTSORTED option` with a BY statement to create ordered or grouped reports without sorting the data. The NOTSORTED option specifies that observations that have the same BY value are grouped together but are not necessarily sorted in alphabetical or numeric order.
General form, BY statement with the NOTSORTED option:

```
BY variable(s) NOTSORTED;
```

where

```
variable(s)
```

names each variable by which the data set is sorted or indexed.

The NOTSORTED option can appear anywhere in the BY statement and is useful if you have data that is in logical categories or groupings such as chronological order. The NOTSORTED option works best when observations that have the same BY value are stored together.

**CAUTION:**
The NOTSORTED option turns off sequence checking. If your data is not grouped, using the NOTSORTED option can produce a large amount of output.

**CAUTION:**
The NOTSORTED option cannot be used with the MERGE or UPDATE statements.

**Example**

Suppose you want to use the PRINT procedure to print the contents of the data set `Retail.Europe`, which contains data about European customers. The data set includes values for country names (`Country_Name`) as well as two-letter country codes (`Country_Code`).

The data is grouped and sorted by the values of `Country_Code`. However, you want the observations in the output to be grouped by the values of `Country_Name`.

**Table 23.1** Country Codes and Country Names

<table>
<thead>
<tr>
<th>Country Code</th>
<th>Country Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>Belgium</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
</tr>
<tr>
<td>ES</td>
<td>Spain</td>
</tr>
<tr>
<td>FI</td>
<td>Finland</td>
</tr>
<tr>
<td>FR</td>
<td>France</td>
</tr>
<tr>
<td>GB</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
</tr>
<tr>
<td>IE</td>
<td>Ireland</td>
</tr>
<tr>
<td>IT</td>
<td>Italy</td>
</tr>
</tbody>
</table>
You can use the NOTSORTED option with a BY statement to accomplish this task without using the SORT procedure.

```
proc print data=retail.europe;
  by country_name notsorted;
run;
```

PROC PRINT output shows that the data is grouped but is not in sorted order. For example, observations in which the value of `Country_Name` is `Germany` are followed by observations in which the value of `Country_Name` is `Denmark`. 
Using **FIRST.** and **LAST.**

The NOTSORTED option can be used with **FIRST.** variable and **LAST.** variable, which are temporary automatic variables in the PDV that identify the first and last observations in each BY group.

These temporary variables are available for DATA step programming but are not added to the output data set. Their values indicate whether an observation is:

- the first one in a BY group
- the last one in a BY group
- neither the first nor the last one in a BY group
- both first and last, as is the case when there is only one observation in a BY group.

You can take actions conditionally, based on whether you are processing the first observation of a BY group or the last.

When an observation is the first in a BY group, SAS sets the value of **FIRST.** variable to 1. For all other observations in the BY group, the value of **FIRST.** variable is 0. Likewise, if an observation is the last in a BY group, SAS sets the value of **LAST.** variable to 1. For all other observations in the BY group, the value of **LAST.** variable is 0.

---

**Example**

The following program creates a new SAS data set **Work.New**. In the input data set, observations that have the same value for **Ordername** (Retail, Catalog, or Internet) are grouped together.
data work.new;
set company.sales
    by ordernam notsorted;
run;

When the program is submitted, SAS creates the temporary variables `FIRST.Ordername` and `LAST.Ordername`. These variables can be used during the DATA step, but they do not become variables in the new data set. The value 1 flags the beginning and end of each value in the BY group:

<table>
<thead>
<tr>
<th>Obs</th>
<th>Customer_ID</th>
<th>OrderName</th>
<th>FIRST.Ordername</th>
<th>FIRST.Ordername</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11791</td>
<td>Retail</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>8406</td>
<td>Retail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>71020</td>
<td>Retail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>21735</td>
<td>Retail</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>82141</td>
<td>Catalog</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>30993</td>
<td>Catalog</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>579</td>
<td>Internet</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>77184</td>
<td>Internet</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Using the GROUPFORMAT Option

The GROUPFORMAT option uses the formatted values of a variable instead of the internal values to determine where a BY group begins and ends, and how `FIRST.variable` and `LAST.variable` are computed.

**General form, BY statement with the GROUPFORMAT option:**

```
BY variable(s) GROUPFORMAT;
```

where

- `variable(s)`
  - names each variable by which the data set is sorted or indexed.

The GROUPFORMAT option

- is available only in the DATA step
- is useful when you define formats for grouped data
- enables the DATA step to process the same groups of data as a summary procedure or PROC REPORT.

When the GROUPFORMAT option is used, the data set must be sorted by the GROUPFORMAT variable or grouped by the formatted values of the GROUPFORMAT variable.
Example

Suppose you want to create a summary report that includes the number of orders for each quarter in 2002. The data for the report is stored in the SAS data set `Company.Orders`.

Table 23.4  SAS Data Set Company.Orders, First Five Observations

<table>
<thead>
<tr>
<th>Obs</th>
<th>Order_ID</th>
<th>Order_Type</th>
<th>Employee_ID</th>
<th>Customer_ID</th>
<th>Order_Date</th>
<th>Delivery_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1230000033</td>
<td>Internet</td>
<td>99999999</td>
<td>8818</td>
<td>01JAN1998</td>
<td>07JAN1998</td>
</tr>
<tr>
<td>2</td>
<td>1230000204</td>
<td>Internet</td>
<td>99999999</td>
<td>47793</td>
<td>01JAN1998</td>
<td>04JAN1998</td>
</tr>
<tr>
<td>3</td>
<td>1230000268</td>
<td>Internet</td>
<td>99999999</td>
<td>71727</td>
<td>01JAN1998</td>
<td>03JAN1998</td>
</tr>
<tr>
<td>4</td>
<td>1230000487</td>
<td>Internet</td>
<td>99999999</td>
<td>74503</td>
<td>01JAN1998</td>
<td>04JAN1998</td>
</tr>
<tr>
<td>5</td>
<td>1230000494</td>
<td>Internet</td>
<td>99999999</td>
<td>8610</td>
<td>01JAN1998</td>
<td>07JAN1998</td>
</tr>
</tbody>
</table>

By creating a format for the data and using the GROUPFORMAT and NOTSORTED options, you cause SAS to create the variables `FIRST.Order_Date` and `LAST.Order_Date` based on the formatted values, not the internal values. This groups the data without requiring the creation of a new variable.

```sas
proc format;
  value qtrfmt '01jan2002' d - '31mar2002' d = '1'
               '01apr2002' d - '30jun2002' d = '2'
               '01jul2002' d - '30sep2002' d = '3'
               '01oct2002' d - '31dec2002' d = '4';
run;

data company.quarters(keep=count order_date rename=(order_date=Quarter));
  set retail.orders;
  format order_date qtrfmt.;
  by order_date groupformat notsorted;
  where year(order_date)=2002;
  if first.order_date then count=0;
  count +1;
  if last.order_date;
run;
```

Using the CLASS Statement

You can also use a CLASS statement to avoid a sort. Unlike the BY statement, the CLASS statement does not require the data to be presorted using the CLASS values, nor does it require an index that is based on the CLASS variables.
If the data cannot be sorted, the CLASS statement is more efficient than the BY statement in terms of CPU time, memory, and I/O usage. However, unlike using the BY statement, presorting the data for use with a CLASS statement does not provide a significant benefit.

General form, CLASS statement:

```
CLASS variable(s) < / options >;
```

where

```
variable(s)
```

specifies one or more variables that the procedure uses to group the data.

Remember that a CLASS statement specifies the variables whose values define the subgroup combinations for an analysis by a SAS procedure. You can use the CLASS statement with the following Base SAS procedures:

- MEANS
- TABULATE
- SUMMARY
- UNIVARIATE.

Variables in a CLASS statement are referred to as class variables. Class variables can be numeric or character. Class variables can have continuous values, but they typically have a few discrete values that define the classifications of the variable.

**CAUTION:**

The comparison of the use of CLASS and BY statements is appropriate for Base SAS procedures only.

**Example**

The *Company.Orders* data set contains the variable *Order_Type*, which has three discrete values:

- Retail
- Catalog
- Internet.

Suppose you want to show the average retail price, cost per unit, and discount for each value of *Order_Type*. You could use the MEANS procedure with either a BY statement or a CLASS statement to complete this task. The statistics created with either of these techniques are the same. However, the report layouts differ.

When the BY statement is used, SAS creates a report for each value of the BY variable. The statistics for each value of *Order_Type* appear in a separate tabular report.

```sas
proc sort data=company.order_fact(keep=order_type quantity total_retail_price costprice_per_unit discount) out=company.orders;
  by order_type;
run;
```

```sas
proc means data=company.orders mean;
```
by order_type;
var total_retail_price -- discount;
freq quantity;
run;

Table 23.5  Output, PROC MEANS with a BY Statement

<table>
<thead>
<tr>
<th>Variable</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total_Retail_Price</td>
<td>177.0073169</td>
</tr>
<tr>
<td>CostPrice_Per_Unit</td>
<td>38.3716413</td>
</tr>
<tr>
<td>Discount</td>
<td>0.38810067</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total_Retail_Price</td>
<td>196.3417829</td>
</tr>
<tr>
<td>CostPrice_Per_Unit</td>
<td>40.8694533</td>
</tr>
<tr>
<td>Discount</td>
<td>0.3889046</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total_Retail_Price</td>
<td>200.2136179</td>
</tr>
<tr>
<td>CostPrice_Per_Unit</td>
<td>41.6397101</td>
</tr>
<tr>
<td>Discount</td>
<td>0.3945766</td>
</tr>
</tbody>
</table>

When the CLASS statement is used, only one report is created. The statistics for each value of Order_Type are consolidated into one tabular report.

Output, PROC MEANS with a CLASS Statement

<table>
<thead>
<tr>
<th>Order_Type</th>
<th>N</th>
<th>Obs</th>
<th>Variable</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retail</td>
<td>1194238</td>
<td></td>
<td>Total_Retail_Price</td>
<td>177.0073169</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CostPrice_Per_Unit</td>
<td>38.3716413</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Discount</td>
<td>0.38810067</td>
</tr>
<tr>
<td>Catalog</td>
<td>222195</td>
<td></td>
<td>Total_Retail_Price</td>
<td>196.3417829</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CostPrice_Per_Unit</td>
<td>40.8694533</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Discount</td>
<td>0.3889046</td>
</tr>
<tr>
<td>Internet</td>
<td>190488</td>
<td></td>
<td>Total_Retail_Price</td>
<td>200.2136179</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CostPrice_Per_Unit</td>
<td>41.6397101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Discount</td>
<td>0.3945766</td>
</tr>
</tbody>
</table>
Comparative Example: Using a BY or CLASS Statement to Avoid a Sort

Suppose you want to create a summary report that shows the average retail price, cost per unit, and discount for each type of order in the Retail.Order_fact data set. Among the techniques you could use are

1. PROC MEANS with a BY Statement, Presorted
2. PROC MEANS with a CLASS Statement
3. PROC MEANS with a CLASS Statement, Presorted
4. PROC SORT and PROC MEANS with a BY Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using a BY or CLASS statement to avoid a sort.

Programming Techniques

1. PROC MEANS with a BY Statement, Presorted

The following program creates a report for each value of the BY variable Order_Type. Each report contains the mean value for the variables Total_Retail_Price, CostPrice_Per_Unit, and Discount. The input data is presorted by the value of Order_Type.

```plaintext
proc means data=retail.orders mean;
  by order_type;
  var total_retail_price costprice_per_unit discount;
  freq quantity;
run;
```

2. PROC MEANS with a CLASS Statement

The following program uses a CLASS statement to create a single, tabular report that includes the mean value of Total_Retail_Price, CostPrice_Per_Unit, and Discount for each category of Order_Type.

```plaintext
proc means data=retail.order_fact(keep=order_type quantity total_retail_price costprice_per_unit discount) mean;
  class order_type;
  var total_retail_price costprice_per_unit discount;
  freq quantity;
run;
```
PROC MEANS with a CLASS Statement, Presorted

In the following program, the input data set `Retail.Order_fact` is presorted by the value of `Order_Type`. The PROC MEANS step uses a CLASS statement to create a single, tabular report that includes the mean value of `Total_Retail_Price`, `CostPrice_Per_Unit`, and `Discount` for each category of `Order_Type`.

```
proc means data=retail.orders(keep=order_type quantity total_retail_price costprice_per_unit discount) mean;
   class order_type;
   var total_retail_price costprice_per_unit discount;
   freq quantity;
run;
```

PROC SORT and PROC MEANS with a BY Statement

In the following program, the PROC SORT step first sorts the input data set `Retail.Order_fact` by the value of `Order_Type`. The PROC MEANS step then creates a report for each value of `Order_Type`. Each report contains the mean value for the variables `Total_Retail_Price`, `CostPrice_Per_Unit`, and `Discount`.

```
proc sort data=retail.order_fact(keep=order_type quantity total_retail_price costprice_per_unit discount) out=retail.orders;
   by order_type;
run;

proc means data=retail.orders mean;
   by order_type;
   var total_retail_price costprice_per_unit discount;
   freq quantity;
run;
```

General Recommendations

- If you can presort the data, use just a BY statement rather than a CLASS statement or a BY statement with a separate SORT procedure.
- If you cannot sort the data, use the CLASS statement.
- Do not presort the data for use with a CLASS statement. Presorting the data does not provide a significant benefit.

Using the SORTEDBY= Data Set Option

If you are working with input data that is already sorted, you can specify how the data is ordered by using the SORTEDBY= data set option.
General form, SORTEDBY= data set option:

\[
\text{SORTEDBY=} \text{by-clause} < \text{/collate-name}> \mid \text{_NULL_}
\]

where

- \text{by-clause}
  - indicates the data order.
- \text{collate-name}
  - names the collating sequence that is used for the sort.
- \text{_NULL_}
  - removes any existing sort information.

\[\text{Note:}\] By default, the collating sequence is that of your operating environment. For details on collating sequences, see the SAS documentation for your operating environment. \\

Although the SORTEDBY= option does not sort a data set, it sets the value of the Sorted flag. It does not set the value of the Validated sort flag. (PROC SORT sets the Validated sort flag.) To see the values of these flags \text{(YES or NO)}, use PROC CONTENTS.

\begin{verbatim}
proc contents;
  data=company.transactions;
run;
\end{verbatim}

\textbf{Figure 23.1} Partial PROC CONTENTS Output

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>COMPANY.TRANSACTIONS</th>
<th>Observations</th>
<th>5504</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>3</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>22:55 Friday, May 2, 2004</td>
<td>Observation Length</td>
<td>32</td>
</tr>
<tr>
<td>Last Modified</td>
<td>22:55 Friday, May 2, 2004</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>WINDOWS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>wlatin1 Western (Windows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Sort Information} & & & \\
\textbf{Sortedby} & Invoice & & \\
\textbf{Validated} & NO & & \\
\textbf{Character Set} & ANSI & & \\
\hline
\end{tabular}
Note: Most SAS procedures and subsystems check the order of the data as it is processed unless the Validated sort flag is set on the file.

Note: Using the Sorted flag without the Validated sort flag improves the speed of index creation.

Example

Suppose you want to create a sorted SAS data set from an external file that contains invoice information. The external file is already sorted by invoice number. You can use the SORTEDBY= data set option to sort the data by the value of Invoice.

```sas
data company.transactions (sortedby=invoice);
  infile extdata;
  input Invoice 1-4 Item $6-20 Amount comma 6.;
run;
```

When the `Company.Transactions` data set is created, the sort information is stored with it. PROC SORT checks the sort information before it sorts a data set so that data is not re-sorted unnecessarily. If you attempt to re-sort the data, the log indicates that the data set is already sorted and that no additional sorting occurred.

```sas
proc sort data=company.transactions;
  by invoice;
run;
```

Table 23.6 SAS Log

```sas
667 proc sort data=work.transactions;
668 by invoice;
669 run;
```

**NOTE:** Input data set is already sorted, no sorting done.

**NOTE:** PROCEDURE SORT used (Total process time):
- real time 0.07 seconds
- cpu time 0.03 seconds

Note: You can specify SORTEDBY=_NULL_ to remove the Sorted flag. The Sorted flag is also removed if you change or add any values of the variables by which the data set is sorted.

Using a Threaded Sort

Threaded processing takes advantage of multiple CPUs by executing multiple threads in parallel (parallel processing). Threaded procedures are completed in less real time than if each task were handled sequentially, although the CPU time is generally increased.

Beginning with SAS 9, the SORT procedure can take advantage of threaded processing. A thread is a single, independent flow of control through a program or within a process.

Threaded sorting is enabled or disabled by using the SAS system option THREADS | NOTHREADS or the THREADS | NOTHREADS procedure option.
General form, SORT procedure with the THREADS | NOTHREADS option:

```
PROC SORT SAS-data-set-name THREADS | NOTHREADS;
```

where

- **SAS-data-set-name**
  - is a valid SAS data set name.
- **THREADS**
  - enables threaded sorting.
- **NOTHREADS**
  - disables threaded sorting.

**Note:** The THREAD | NOTHREADS procedure option overrides the value of the SAS system option THREADS | NOTHREADS. For information about the THREADS | NOTHREADS system option, see the SAS documentation.

When a threaded sort is used, the observations in the input data set are divided into equal temporary subsets, based on the number of processors that are allocated to the SORT procedure. Each subset is then sorted on a different processor. The sorted subsets are then interleaved to re-create the sorted version of the input data set.

### Using the CPUCOUNT= System Option

The performance of threaded sorting is affected by the value of the CPUCOUNT= system option. CPUCOUNT= specifies the number of processors that thread-enabled applications should assume will be available for concurrent processing. SAS uses this information to determine how many threads to start, not to restrict the number of CPUs that will be used.

General form, CPUCOUNT= system option:

```
CPUCOUNT= n | ACTUAL;
```

where

- **n**
  - is a number from 1 to 1024 that indicates how many CPUs SAS will assume are available for use by thread-enabled applications.
- **ACTUAL**
  - causes SAS to detect how many CPUs are available for a specific session.

**CAUTION:**

Setting CPUCOUNT= to a number greater than the actual number of available CPUs might result in reduced overall performance.

**Note:** For more information about the CPUCOUNT= system option and other options that are relevant to SAS threading technology, see the SAS documentation.
Calculating and Allocating Sort Resources

Sort Space Requirements

When data is sorted, SAS requires enough space in the data library for two copies of the data file that is being sorted as well as additional workspace.

In releases prior to SAS 9, the workspace required for an uncompressed data file is approximately three to four times the size of the data file. Beginning with SAS 9, the workspace required for an uncompressed data file is approximately twice the size of the data file. The workspace can be allocated in memory and/or on disk as a utility file, depending on the sort utility and on the options chosen.

You can use the following formula to calculate the amount of workspace that the SORT procedure requires:

### Formula for calculating the amount of workspace needed to sort a SAS data set:

\[
\text{bytes required} = (\text{key-variable-length} + \text{observation-length}) \\
\times \text{number-of-observations} \times 4
\]

where

- **key-variable-length** is the length of all key variables added together.
- **observation-length** is the maximum observation length.
- **number-of-observations** is the number of observations.

**Note:** The multiplier 4 applies only to utility files used in releases prior to SAS 9 when PROC SORT needs to use disk space in order to sort the data. For in-memory sorting and sorting with SAS 9 and later, the multiplier is 2 or less.

**Example**

Suppose you want to submit the following program under SAS 9:

```sas
proc sort data=company.customers;
    by customer_group customer_lastname;
run;
```

You can use the CONTENTS procedure or the DATASETS procedure to obtain the information that is required for the calculation.
In this case, the amount of workspace needed to sort the data set is 48,575,160 bytes.

\[48,575,160 \text{ bytes} = (70+200) \times 89954 \times 2\]

Note: The SORT procedure is very I/O intensive. If the file that you are sorting is located in the Work library, all of the I/O for the procedure takes place on the file system in which the Work library is stored because, by default, the utility files that the SORT procedure creates are created in the Work library.

Beginning with SAS 9, you can use the UTILLOC= system option to specify one or more file systems in which utility files can be stored. For information about the UTILLOC= option, see the SAS documentation.
Using the SORTSIZE= Option

The SORTSIZE= system option or procedure option specifies how much memory is available to the SORT procedure. Specifying the SORTSIZE= option in the PROC SORT statement temporarily overrides the SAS system option SORTSIZE=.

General form, SORTSIZE= option:

```
SORTSIZE=memory-specification;
```

where

- `memory-specification` specifies the maximum amount of memory that is available to PROC SORT. Valid values for `memory-specification` are as follows:
  - `MAX` specifies that all available memory can be used.
  - `n` specifies the amount of memory in bytes, where `n` is a real number.
  - `nK` specifies the amount of memory in kilobytes, where `n` is a real number.
  - `nM` specifies the amount of memory in megabytes, where `n` is a real number.
  - `nG` specifies the amount of memory in gigabytes, where `n` is a real number.

**Note:** The default value of the SORTSIZE= option depends on your operating environment. See the SAS documentation for your operating environment for more information.

Generally, the value of SORTSIZE= should be less than the physical memory that is available to your process.

If the required workspace is less than or equal to the value specified in the SORTSIZE= system option or procedure option, then the entire sort can take place in memory, which reduces processing time.

If the actual required workspace is greater than the value specified in the SORTSIZE= system option or procedure option, then processing time is increased because the SORT procedure must:

1. create temporary utility files in the Work directory or mainframe temporary area
2. request memory up to the value specified by SORTSIZE=
3. write a portion of the sorted data to a utility file.

This process is repeated until all of the data is sorted. The SORT procedure then interleaves the data in the utility files to create the final data set.

PROC SORT attempts to adapt to the constraint that is imposed by the SORTSIZE= option. Because PROC SORT uses memory as much as possible,

- a small SORTSIZE= value can increase CPU and I/O resource utilization
- a large SORTSIZE= value can decrease CPU and I/O resource utilization.
Handling Large Data Sets

Dividing a Large Data Set

A data set is too large to sort when there is insufficient room in the data library for a second copy of the data set or when there is insufficient disk space for three to four temporary copies of the data set.

One approach to this situation is to divide the large data set into smaller data sets. The smaller data sets can then be sorted and combined to re-create the large data set. This approach is similar to the process that is used in a threaded sort.

Techniques for dividing and sorting a large data set include

- using PROC SORT with the OUT= statement option and the FIRSTOBS= and OBS= data set options
- using PROC SORT with a WHERE statement
- using subsetting with IF-THEN/ELSE or SELECT-WHEN logic to create multiple output data sets, then sorting the output data sets.

Techniques that can be used to rebuild a large data from smaller, sorted data sets set include

- concatenating the smaller data sets with a SET statement
- interleaving the smaller data sets with SET and BY statements
- appending the smaller data sets with the APPEND procedure.

Comparative Example: Dividing and Sorting a Large Data Set 1

Suppose you want to sort the SAS data set Retail.Order_fact by the value of Order_Date. The data set is too large to sort using a single SORT procedure. You could accomplish this task by

1 Segmenting by Observation
2 Subsetting Using an IF Statement with the YEAR Function
3 Subsetting Using an IF Statement with a Date Constant
4 Subsetting Using a WHERE Statement with the YEAR Function
5 Subsetting Using a WHERE Statement with a Date Constant.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for dividing and sorting a large data set.
Programming Techniques

Segmentation by Observation

This program segments the data set Orion.Order_fact into three smaller data sets by observation number. The three smaller data sets, Work.One, Work.Two, and Work.Three, are sorted by the value of Order_Date. The large data set is then re-created by interleaving the three smaller, sorted data sets.

```sas
proc sort data=retail.order_fact
   (firstobs = 1 obs = 1500000)
   out=work.one;
   by order_date;
run;

proc sort data=retail.order_fact
   (firstobs = 1500001 obs = 3000000)
   out=work.two;
   by order_date;
run;

proc sort data=retail.order_fact
   (firstobs = 3000001)
   out=work.three;
   by order_date;
run;

data work.orders;
   set work.one work.two work.three;
   by order_date;
run;
```
Subsetting Using an IF Statement with the YEAR Function

This program segments the data set *Orion.Order_fact* into three smaller data sets by using a subsetting IF statement and the YEAR function. The three smaller data sets, *Work.One*, *Work.Two*, and *Work.Three*, are then sorted by the value of **Order_Date**. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key **Order_Date**.

```plaintext
data work.one work.two work.three;
  set retail.order_fact;
  year=year(order_date);
  if year in (1998,1999)
    then output work.one;
  else if year in (2000,2001)
    then output work.two;
  else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;

proc sort data=work.two;
  by order_date;
run;
proc sort data=work.three;
  by order_date;
run;

data work.orders;
  set work.one work.two work.three;
run;
```
Subsetting Using an IF Statement with a Date Constant

This program segments the data set Orion.Order_fact into three smaller data sets by using a subsetting IF statement and a date constant. The three smaller data sets, Work.One, Work.Two, and Work.Three, are then sorted by the value of Order_Date. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key Order_Date.

data work.one work.two work.three;
  set retail.order_fact;
  if order_date <= '31Dec1999'd then
    output work.one;
  else if '31dec1999'd < order_date < '01jan2002'd
    then output work.two;
  else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;

proc sort data=work.two;
  by order_date;
run;

proc sort data=work.three;
  by order_date;
run;

data work.orders;
  set work.one work.two work.three;
run;
Subsetting Using a WHERE Statement with the YEAR Function

This program segments the data set Orion.Order_fact into three smaller data sets by using a WHERE statement and the YEAR function. The three smaller data sets, Work.One, Work.Two, and Work.Three, are sorted by the value of Order_Date. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key Order_Date.

```sas
proc sort data=retail.order_fact
    out=work.one;
    by order_date;
    where year(order_date) in (1998, 1999);
run;

proc sort data=retail.order_fact
    out=work.two;
    by order_date;
    where year(order_date) in (2000, 2001);
run;

proc sort data=retail.order_fact
    out=work.three;
    by order_date;
    where year(order_date) in (2002);
run;

data work.orders;
    set work.one work.two work.three;
run;
```

Subsetting Using a WHERE Statement with a Date Constant

This program segments the data set Orion.Order_fact into three smaller data sets by using a WHERE statement with a date constant. The three smaller data sets, Work.One, Work.Two, and Work.Three, are sorted by the value of Order_Date. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key Order_Date.

```sas
proc sort data=retail.order_fact
    out=work.one;
    by order_date;
    where order_date le '31Dec1999'd;
run;

proc sort data=retail.order_fact
    out=work.two;
    by order_date;
    where order_date between '01jan2000'd and '31dec2001'd;
run;

proc sort data=retail.order_fact
    out=work.three;
    by order_date;
    where order_date ge '01jan2002'd;
run;

data work.orders;
    set work.one work.two work.three;
run;
```
General Recommendations

- Use a DATA step rather than PROC APPEND to re-create a large data set from smaller subsets.
- Use a constant rather than a SAS function because calling a function repeatedly increases CPU usage.
- Use a subsetting IF with either a constant or a function rather than a WHERE statement with a function.

Comparative Example: Dividing and Sorting a Large Data Set 2

Like the programs shown in the previous section, each of the following programs illustrates a method for dividing the large data set Retail.Order_fact into smaller data sets for sorting. However, in this example, the smaller data sets, Work.One, Work.Two, and Work.Three, are combined using the APPEND procedure rather than a DATA step in programs 2, 3, 4, and 5.

- Segmenting by Observation
- Subsetting Using an IF Statement with the YEAR Function
- Subsetting Using an IF Statement with a Date Constant
- Subsetting Using a WHERE Statement with the YEAR Function
- Subsetting Using a WHERE Statement with a Date Constant.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for dividing and sorting a large data set.

Programming Techniques
Segmenting by Observation

This program segments the data set Orion.Order_fact into three smaller data sets by observation number. The three smaller data sets, Work.One, Work.Two, and Work.Three, are sorted by the value of Order_Date. PROC APPEND cannot be used to re-create a large data set that was segmented using FIRSTOBS= and OBS=. Therefore, the large data set is re-created by interleaving the three smaller, sorted data sets.

```plaintext
proc sort data=retail.order_fact
  (firstobs = 1 obs = 1500000)
  out=work.one;
  by order_date;
run;

proc sort data=retail.order_fact
  (firstobs = 1500001 obs = 3000000)
  out=work.two;
  by order_date;
run;

proc sort data=retail.order_fact
  (firstobs = 3000001)
  out=work.three;
  by order_date;
run;

data work.orders;
  set work.one work.two work.three;
  by order_date;
run;
```
Subsetting Using an IF Statement with the YEAR Function

This program segments the data set Orion.Order_fact into three smaller data sets by using a subsetting IF statement and the YEAR function. The three smaller data sets, Work.One, Work.Two, and Work.Three, are then sorted by the value of Order_Date. The large data set is then re-created using the APPEND procedure.

data work.one work.two work.three;
  set retail.order_fact;
  year=year(order_date);
  if year in (1998,1999)
    then output work.one;
  else if year in (2000,2001)
    then output work.two;
  else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;

proc sort data=work.two;
  by order_date;
run;

proc sort data=work.three;
  by order_date;
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;
Subsetting Using an IF Statement with a Date Constant

This program segments the data set *Orion.Order_fact* into three smaller data sets by using a subsetting IF statement and a date constant. The three smaller data sets, *Work.One*, *Work.Two*, and *Work.Three*, are then sorted by the value of *Order_Date*. The large data set is then re-created using the APPEND procedure.

```plaintext
data work.one work.two work.three;
  set retail.order_fact;
  if order_date <= '31Dec1999'd then
    output work.one;
  else if '31dec1999'd < order_date < '01jan2002'd
    then output work.two;
  else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;

proc sort data=work.two;
  by order_date;
run;

proc sort data=work.three;
  by order_date;
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;
```
Subsetting Using a WHERE Statement with the YEAR Function

This program segments the data set Orion.Order_fact into three smaller data sets by using a WHERE statement and the YEAR function. The three smaller data sets, Work.One, Work.Two, and Work.Three, are sorted by the value of Order_Date. The large data set is then re-created using the APPEND procedure.

```
proc sort data=retail.order_fact
   out=work.one;
   by order_date;
   where year(order_date) in (1998, 1999);
run;

proc sort data=retail.order_fact
   out=work.two;
   by order_date;
   where year(order_date) in (2000, 2001);
run;

proc sort data=retail.order_fact
   out=work.three;
   by order_date;
   where year(order_date) in (2002);
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;
```
Subsetting Using a WHERE Statement with a Date Constant

This program segments the data set Orion.Order_fact into three smaller data sets by using a WHERE statement with a date constant. The three smaller data sets, Work.One, Work.Two, and Work.Three, are sorted by the value of Order_Date. The large data set is then re-created using the APPEND procedure.

```
proc sort data=retail.order_fact
  out=work.one;
  by order_date;
  where order_date le '31Dec1999'd;
run;

proc sort data=retail.order_fact
  out=work.two;
  by order_date;
  where order_date between '01jan2000'd and '31dec2001'd;
run;

proc sort data=retail.order_fact
  out=work.three;
  by order_date;
  where order_date ge '01jan2002'd;
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;
```

General Recommendations

- Use a DATA step rather than PROC APPEND to re-create a large data set from smaller subsets.

Using the TAGSORT Option

You can also use the TAGSORT option to sort a large data set. The TAGSORT option stores only the BY variables and the observation numbers in temporary files. The BY variables and the observation numbers are called tags. At the completion of the sorting process, PROC SORT uses the tags to retrieve records from the input data set in sorted order.

General form, SORT procedure with the TAGSORT option:

```
PROC SORT DATA=SAS-data-set-name TAGSORT;
```

where

- `SAS-data-set-name` is a valid SAS data set name.
When the total length of the BY variables is small compared to the record length, TAGSORT reduces temporary disk usage considerably because sorting just the BY variables means sorting much less data. However, processing time is usually higher than if a regular sort is used because TAGSORT increases CPU time and I/O usage in order to save memory and disk space. TAGSORT

- uses significantly more CPU time and I/O than a regular sort if the data is extremely out of order with regard to the BY variables
- uses slightly more CPU time and I/O than a regular sort if the data is mostly in order with regard to the BY variables.

**Example**

In the following program, only the BY variables, Customer_ID and Order_Date, and the tags are stored in temporary files within the sort workspace. SAS then

- sorts the temporary files
- uses the tags to retrieve the observations from the original data set
- re-creates the sorted data set.

```sas
proc sort data=company.orders tagsort;
   by customer_id order_date;
run;
```

CAUTION:
The TAGSORT option is not supported by the threaded sort.

---

**Removing Duplicate Observations Efficiently**

The SORT procedure can be used to remove duplicate observations when it is

- used with the NODUPKEY option
Generally, PROC SORT with the NODUPKEY option uses less I/O and CPU time than PROC SORT followed by a DATA step that uses FIRST. processing. Before viewing a comparative example, let's examine each of the techniques that are listed above.

### Using the NODUPKEY Option

The NODUPKEY option checks for and eliminates observations that have duplicate BY-variable values. If you specify this option, then PROC SORT compares *all BY-variable values* for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

**General form, PROC SORT with the NODUPKEY option:**

```
PROC SORT DATA=SAS-data-set-name NODUPKEY;
```

where

- `SAS-data-set-name` is a valid SAS data set name.

**Example**

The SAS data set `Company.Reorder` contains two duplicated observations. Observation 9 is a duplicate of observation 1, and observation 7 is a duplicate of observation 2. The duplicate observations are removed when the data is sorted by the values of `Product_Line` and `Product_Name` and when the NODUPKEY option is used.

```
proc sort data=company.reorder nodupkey;
  by product_line product_name;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Kids Children's Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>3</td>
<td>Clothes &amp; Shoes</td>
<td>Watchit 120 Sterling/Reflective</td>
<td>Eclipse Inc</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>Sparkle Spray Blue</td>
<td>CrystalClear Optics Inc</td>
</tr>
<tr>
<td>5</td>
<td>Outdoors</td>
<td>Money Purse, Black</td>
<td>Top Sports</td>
</tr>
<tr>
<td>6</td>
<td>Sports</td>
<td>Mayday Serious Down Jacket</td>
<td>Mayday Inc</td>
</tr>
<tr>
<td>7</td>
<td>Children</td>
<td>Kids Children's Fleece Hat</td>
<td>3Top Sports</td>
</tr>
</tbody>
</table>
Using the NODUPRECS Option

The NODUPRECS option also checks for and eliminates duplicate observations. However, unlike the NODUPKEY option, the NODUPRECS option compares all of the variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.
General form, PROC SORT with the NODUPRECS option:

\[
\text{PROC SORT DATA=}\text{SAS-data-set-name NODUPRECS;}
\]

where

\[
\text{SAS-data-set-name}
\]

is a valid SAS data set name.

Note: NODUP is an alias for NODUPRECS.

Because NODUPRECS checks only consecutive observations, some nonconsecutive duplicate observations might remain in the output data set. You can remove all duplicates with this option by sorting on all variables.

**Example**

When *Company.Reorder* is sorted using the BY variable *Product_Line* and the NODUPRECS option, the duplicate observation that contains the product name *Kids Children’s Fleece Hat* to the output data set. The duplicate observation that contains the product name *Ski Jacket w/Removable Fleece* to the output data set.

```sql
proc sort data=company.reorder noduprecs;
   by product_line;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Kids Children’s Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>3</td>
<td>Clothes &amp; Shoes</td>
<td>Watchit 120 Sterling/Reflective</td>
<td>Eclipse Inc</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>Sparkle Spray Blue</td>
<td>CrystalClear Optics Inc</td>
</tr>
<tr>
<td>5</td>
<td>Outdoors</td>
<td>Money Purse, Black</td>
<td>Top Sports</td>
</tr>
<tr>
<td>6</td>
<td>Sports</td>
<td>Mayday Serious Down Jacket</td>
<td>Mayday Inc</td>
</tr>
<tr>
<td>7</td>
<td>Children</td>
<td>Kids Children’s Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>8</td>
<td>Clothes &amp; Shoes</td>
<td>Tyfoon Linen Pants</td>
<td>Typhoon Clothing</td>
</tr>
<tr>
<td>9</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
</tbody>
</table>

Table 23.10  SAS Data Set Company.Reorder

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Kids Children’s Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>3</td>
<td>Children</td>
<td>Kids Children’s Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>4</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
</tbody>
</table>
## Table 23.12  SAS Data Set Company.Reorder, Duplicate Observation Remaining

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Clothes &amp; Shoes</td>
<td>Watchit 120 Sterling/Reflective</td>
<td>Eclipse Inc</td>
</tr>
<tr>
<td>6</td>
<td>Clothes &amp; Shoes</td>
<td>Typhoon Linen Pants</td>
<td>Typhoon Clothing</td>
</tr>
<tr>
<td>8</td>
<td>Sports</td>
<td>Sparkle Spray Blue</td>
<td>CrystalClear Optic Inc</td>
</tr>
<tr>
<td>9</td>
<td>Sports</td>
<td>Mayday Serious Down Jacket</td>
<td>Mayday Inc</td>
</tr>
</tbody>
</table>

Both duplicate observations are removed when `Company.Reorder` is sorted by both `Product_Line` and `Product_Name` and when the NODUPRECS option is used.

```sas
proc sort data=company.reorder noduprecs;
  by product_line product_name;
run;
```

## Table 23.13  SAS Data Set Company.Reorder, Both Duplicate Observations Removed

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Children</td>
<td>Kids Children’s Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
<tr>
<td>3</td>
<td>Clothes &amp; Shoes</td>
<td>Typhoon Linen Pants</td>
<td>Typhoon Clothing</td>
</tr>
<tr>
<td>4</td>
<td>Clothes &amp; Shoes</td>
<td>Watchit 120 Sterling/Reflective</td>
<td>Eclipse Inc</td>
</tr>
<tr>
<td>5</td>
<td>Clothes &amp; Shoes</td>
<td>Typhoon Linen Pants</td>
<td>Typhoon Clothing</td>
</tr>
<tr>
<td>6</td>
<td>Outdoors</td>
<td>Money Purse, Black</td>
<td>Top Sports</td>
</tr>
<tr>
<td>7</td>
<td>Sports</td>
<td>Sparkle Spray Blue</td>
<td>CrystalClear Optic Inc</td>
</tr>
<tr>
<td>8</td>
<td>Sports</td>
<td>Mayday Serious Down Jacket</td>
<td>Mayday Inc</td>
</tr>
</tbody>
</table>

**Note:** The SORTDUP= system option controls how NODUPRECS processing works. Specifying SORTDUP=PHYSICAL removes duplicates based on all variables in the data set. This is the default. Specifying SORTDUP=LOGICAL removes duplicates based only on the variables that remain after the DROP= and KEEP= data set options are processed. See the SAS documentation for more information.
Using the EQUALS | NOEQUALS Option

EQUALS | NOEQUALS is a SORT procedure option that helps to determine the order of observations in the output data set. When you use NODUPRECS or NODUPKEY to remove observations from the output data set, the choice of EQUALS or NOEQUALS can have an effect on which observations are removed.

EQUALS is the default. For observations that have identical BY-variable values, EQUALS maintains the order from the input data set in the output data set. NOEQUALS does not necessarily preserve this order in the output data set. NOEQUALS can save CPU time and memory resources.

Example

The following program uses PROC SORT with the NODUPKEY option and the NOEQUALS option to create an output data set that contains only the first observation in each BY group. Notice that the output data set Work.New contains different observations when the EQUALS option is used.

```
proc sort data=company.products out=work.new
   nodupkey noequals;
   by product_line;
run;
```

Table 23.14  SAS Data Set Company.Products

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Clothes &amp; Shoes</td>
<td>Big Guy Men's Ringer T</td>
<td>Eclipse Inc</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Boy's and Girl's Ski Pants with Braces</td>
<td>Scandinavian Clothing A/S</td>
</tr>
<tr>
<td>3</td>
<td>Outdoors</td>
<td>Cotton Moneybelt/Polyester 45x11</td>
<td>Prime Sports Ltd</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>Cougar Shorts</td>
<td>SD Sporting Goods Inc</td>
</tr>
<tr>
<td>5</td>
<td>Clothes &amp; Shoes</td>
<td>Far Out Teambag S</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>6</td>
<td>Children</td>
<td>Kid Basic Tracking Suit</td>
<td>Triple Sportswear Inc</td>
</tr>
<tr>
<td>7</td>
<td>Sports</td>
<td>Maxrun Ultra short Sprinter Tights</td>
<td>Force Sports</td>
</tr>
<tr>
<td>8</td>
<td>Clothes &amp; Shoes</td>
<td>Wa.leather Street Shoes</td>
<td>Fuller Trading Co.</td>
</tr>
</tbody>
</table>

Table 23.15  SAS Data Set Work.New, NOEQUALS Option Used

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Children</td>
<td>Kid Basic Tracking Suit</td>
<td>Triple Sportswear Inc</td>
</tr>
<tr>
<td>2</td>
<td>Clothes &amp; Shoes</td>
<td>Far Out Teambag S</td>
<td>3Top Sports</td>
</tr>
</tbody>
</table>
Using FIRST. LAST. Processing in the DATA Step

FIRST. LAST. processing in the DATA step can also be used to remove duplicate observations from a SAS data set.

In the data set `Company.Onorder`, the fourth observation contains a duplicate value for `Product_Name`. The following program removes the observation that contains the duplicate value by first sorting the input data set, `Company.Onorder`, by the value of `Product_Name`. The DATA step then selects only the first observation in the BY group.

```sas
proc sort data=company.onorder
   out=work.sorted;
   by product_name;
run;
data work.onorder2;
   set work.sorted;
   by product_name;
   if first.product_name;
run;
```

Table 23.17 SAS Data Set Company.Onorder

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Clothes &amp; Shoes</td>
<td>Big Guy Men's Ringer T</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Boy's and Girl's Ski Pants with Braces</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>Outdoors</td>
<td>Cotton Moneybelt/Polyester 45x11</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>Big Guy Men's Ringer T</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>Cougar Shorts</td>
<td>40</td>
</tr>
</tbody>
</table>
Comparative Example: Removing Duplicate Observations Efficiently

Suppose you want to remove observations from the data set Retail.Order_fact in which the value of Order_Date is duplicated. Among the techniques you could use are

1. The NODUPKEY Option and the EQUALS Option
2. The NODUPKEY Option and the NOEQUALS Option
3. PROC SORT and a DATA Step with BY-Group and FIRST. Processing

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendation for eliminating duplicates.

Programming Techniques

1. The NODUPKEY Option and the EQUALS Option

This program uses the NODUPKEY option and the EQUALS option to check for and eliminate observations that have duplicate BY-variable values. For observations that have identical BY-variable values, the EQUALS option maintains the order from the input data set in the output data set.

```
proc sort data=retail.order_fact
  out=work.sorted
  nodupkey equals;
by order_date;
run;
```
The NODUPKEY Option and the NOEQUALS Option

This program uses the NODUPKEY option and the NOEQUALS option to check for and eliminate observations that have duplicate BY-variable values. For observations that have identical BY-variable values, the NOEQUALS option does not necessarily maintain the order from the input data set in the output data set.

```sas
proc sort data=retail.order_fact
   out=work.sorted
   nodupkey noequals;
by order_date;
run;
```

PROC SORT and a DATA Step with BY-Group and FIRST. Processing

In this program, the input data set is first sorted using the SORT procedure. Duplicate observations are then removed using BY-group and FIRST. processing.

```sas
proc sort data=retail.order_fact
   out=work.sorted;
by order_date;
run;
data work.sorted;
   set work.sorted;
   by order_date;
   if first.order_date;
run;
```

General Recommendations

- To remove duplicate observations from a SAS data set, use PROC SORT with the NODUPKEY option rather than a PROC SORT step followed by a DATA step that uses FIRST. processing.
- Be careful not to confuse NODUPKEY with NODUPRECS.

Additional Features

Selecting a Host Sort Utility

Host sort utilities are third-party sort packages that are available in some operating environments. In some cases, using a host sort utility with PROC SORT might be more efficient than using the SAS sort utility with PROC SORT.

The following table lists the host sort utilities that might be available at your site. SAS uses the values that are set for the SORTPGM=, SORTCUT=, SORTCUTP=, and SORTNAME= system options to determine which sort to use.
Selecting Efficient Sorting Strategies

Using the SORTCUTP= System Option

<table>
<thead>
<tr>
<th>Operating Environment</th>
<th>Host Sort Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>z/OS</td>
<td>Dsor (default)</td>
</tr>
<tr>
<td></td>
<td>Syncsort</td>
</tr>
<tr>
<td>UNIX</td>
<td>Cosort</td>
</tr>
<tr>
<td></td>
<td>Syncsort (default)</td>
</tr>
<tr>
<td>Windows</td>
<td>Syncsort</td>
</tr>
</tbody>
</table>

**Note:** Ask your system administrator whether a host sort utility is available at your site. For more information about host sort utilities, see the SAS documentation for your operating environment.

Using the SORTPGM= System Option

The value specified in the SORTPGM= system option tells SAS whether to use the SAS sort, to use the host sort, or to determine which sort utility is best for the data set.

**General form, SORTPGM= system option:**

```plaintext
OPTIONS SORTPGM= BEST | HOST | SAS;
```

where

- **BEST**
  - specifies that SAS chooses the sort utility. This is the default.
- **HOST**
  - specifies that the host sort utility is always used.
- **SAS**
  - specifies that the SAS sort utility is always used.

Using the SORTCUTP= System Option

The SORTCUTP= system option specifies the number of bytes above which the host sort utility is used instead of the SAS sort utility.
General form, SORTCUTP= system option:

```
OPTIONS SORTCUTP=n / nK / nM / nG / MIN / MAX / hexX;
```

where

- `n / nK / nM / nG /` specifies the value in bytes, kilobytes, megabytes, or gigabytes, respectively.
- `MIN` specifies the minimum value.
- `MAX` specifies the maximum value.
- `hexX` specifies the value as a hexadecimal number of bytes.

Note: To determine the minimum and maximum values for SORTCUTP=, see the SAS documentation for your operating environment.

The following table lists the default values for SORTCUTP= in the z/OS, UNIX, and Windows operating environments.

<table>
<thead>
<tr>
<th>Operating Environment</th>
<th>Default SORTCUTP= Value</th>
<th>Default Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>z/OS</td>
<td>4M</td>
<td>SAS sort is used until this value is reached</td>
</tr>
<tr>
<td>UNIX</td>
<td>0</td>
<td>SAS sort is always used</td>
</tr>
<tr>
<td>Windows</td>
<td>0</td>
<td>SAS sort is always used</td>
</tr>
</tbody>
</table>

Using the SORTCUT= System Option

Beginning with SAS 9, the SORTCUT= system option can be used to specify the number of observations above which the host sort utility is used instead of the SAS sort utility.

Note: The SORTCUT= system option is not available in the z/OS operating environment.
General form, \texttt{SORTCUT=} system option:

\begin{verbatim}
OPTIONS SORTCUT=n | nK | nM | nG | MIN | MAX | hexX;
\end{verbatim}

where

\begin{itemize}
  \item \texttt{n} | \texttt{nK} | \texttt{nM} | \texttt{nG}
    \hspace{1cm} \text{specifies the number of observations.}
  \item \texttt{MIN}
    \hspace{1cm} \text{specifies 0 observations.}
  \item \texttt{MAX}
    \hspace{1cm} \text{specifies the maximum number of observations.}
  \item \texttt{hexX}
    \hspace{1cm} \text{specifies the number of observations in hexadecimal notation.}
\end{itemize}

\textit{Note:} To determine the maximum value for \texttt{SORTCUT=}, see the SAS documentation for your operating environment. \triangle

The default value of the \texttt{SORTCUT=} system option is 0.

\section*{Using the \texttt{SORTNAME=} System Option}

The \texttt{SORTNAME=} option specifies the host sort utility that will be used if the value of \texttt{SORTPGM=} is \texttt{BEST} or \texttt{HOST}.

\textit{Note:} The \texttt{SORTNAME=} system option is not available in the Windows operating environment. \triangle

General form, \texttt{SORTNAME=} option:

\begin{verbatim}
OPTIONS SORTNAME=host-sort-utility name;
\end{verbatim}

where

\begin{verbatim}
host-sort-utility name
\end{verbatim}

\hspace{1cm} \text{is the name of a valid sort host utility.}

\section*{Example}

When you specify \texttt{SORTPGM=}\texttt{BEST}, SAS uses the value of the \texttt{SORTCUT=} and \texttt{SORTCUTP=} options to determine whether to use the host sort or the SAS sort. If you specify values for both the \texttt{SORTCUT=} and \texttt{SORTCUTP=} options, and if either condition is true, SAS chooses the host sort.

In the program below, if the size of the SAS data set \textit{Company.Orders} is larger than 10,000 bytes, the host sort utility, Syncsort, will be used instead of the SAS sort utility.

\begin{verbatim}
options sortpgm=best sortcutp=10000
   sortname=syncsort;

proc sort data=company.orders out=company.deliveries;
\end{verbatim}
by delivery_date;
run;
Avoiding Unnecessary Sorts

When BY-group processing with an index is used, the data can be sequenced by different variables without having to repeat the SORT procedure if multiple indexes are used. Because indexes are updated automatically, there is no need to re-sort a data set when observations are modified or added. However, BY-group processing with an index is less efficient than reading a sorted data set sequentially, and storage space is required for the index.

You can also use the NOTSORTED option with a BY statement to create ordered or grouped reports without sorting the data. The NOTSORTED option specifies that observations that have the same BY value are grouped together but are not necessarily sorted in alphabetical or numeric order. The NOTSORTED option works best when observations that have the same BY value are stored together.

The NOTSORTED option can be used with FIRST. and LAST., which are temporary automatic variables in the PDV that identify the first and last observations in each BY group. These temporary variables are available for DATA step programming but are not added to the output data set.

The GROUPFORMAT option is useful when you have defined formats for grouped data. The GROUPFORMAT option uses the formatted values of a variable, instead of the internal values to determine where a BY group begins and ends, and how FIRST. and LAST. are computed. When the GROUPFORMAT option is used, the data set must be sorted by the GROUPFORMAT variable or grouped by the formatted values of the GROUPFORMAT variable.

You can use a CLASS statement to specify the variables whose values define the subgroup combinations for an analysis by a SAS procedure. Unlike the BY statement, when the CLASS statement is used with Base SAS procedures, it does not require the data to be presorted using the BY-variable values or that you have an index based on the BY variables. If the data cannot be sorted, the CLASS statement is more efficient than the BY statement in terms of CPU time, memory, and I/O usage.

If you are working with input data that is already sorted, you can specify how the data is ordered by using the SORTEDBY= data set option. Although the SORTEDBY= option does not sort a data set, it sets the Sorted flag on the data set.

Review the related comparative examples:

- “Comparative Example: Using BY-Group Processing with an Index to Avoid a Sort” on page 788
- “Comparative Example: Using a BY or CLASS Statement to Avoid a Sort” on page 797.

Using a Threaded Sort

Beginning with SAS 9, the SORT procedure can take advantage of threaded processing. Threaded jobs are completed in substantially less real time than if each task is handled sequentially. However, the CPU time for threaded jobs is generally increased.

Threaded sorting is enabled or disabled by using the THREADS | NOTHREADS SAS system option or procedure option. The procedure option overrides the value of the system option.

When a threaded sort is used, the observations in the input data set are divided into equal temporary subsets, based on how many processors are allocated to the SORT procedure. Each subset is then sorted on a different processor. The sorted subsets are then interleaved to re-create the sorted version of the input data set.
The performance of a threaded sort is affected by the value of the CPUCOUNT= system option. CPUCOUNT= specifies the number of processors that thread-enabled applications should assume will be available for concurrent processing. SAS uses this information to determine how many threads to start, not to restrict the number of CPUs that will be used.

Calculating and Allocating Sort Resources

When data is sorted, SAS requires enough space in the data library for two copies of the data file that is being sorted, as well as additional workspace.

In releases prior to SAS 9, the required workspace is approximately three to four times the size of the data file. Beginning with SAS 9, the required workspace is approximately twice the size of the data file. The workspace can be allocated in memory and/or on disk as a utility file, depending on which sort utility and options are specified.

The SORTSIZE= option specifies how much memory is available to the SORT procedure. Generally, the value of SORTSIZE= should be less than the physical memory that is available to your process. If the required workspace is less than or equal to the value specified in the SORTSIZE= system option or procedure option, then the entire sort can take place in memory, which reduces processing time.

Handling Large Data Sets

A data set is too large to sort when there is insufficient room in the data library for a second copy of the data set or when there is insufficient disk space for three to four temporary copies of the data set.

One approach to this situation is to divide the large data set into smaller subsets. The subsets can then be sorted and combined to re-create the large data set.

You can also use the TAGSORT option to sort a large data set. The TAGSORT option stores only the BY variables and the observation numbers in temporary files. The BY variables and the observation numbers are called tags. At the completion of the sorting process, PROC SORT uses the tags to retrieve records from the input data set in sorted order.

When the total length of the BY variables is small compared to the record length, TAGSORT reduces temporary disk usage considerably because sorting just the BY variables means sorting much less data. However, processing time might be much higher because the TAGSORT option increases CPU and I/O usage in order to save memory and disk space.

Review the related comparative examples:

- “Comparative Example: Dividing and Sorting a Large Data Set 1” on page 805
- “Comparative Example: Dividing and Sorting a Large Data Set 2” on page 810.

Removing Duplicate Observations Efficiently

The NODUPKEY option checks for and eliminates observations that have duplicate BY-variable values. If you specify this option, then PROC SORT compares all BY-variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

The NODUPRECS option checks for and eliminates duplicate observations. However, unlike the NODUPKEY option, the NODUPRECS option compares all of the variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.
EQUALS | NOEQUALS is a procedure option that helps to determine the order of observations in the output data set. When you use NODUPRECS or NODUPKEY to remove observations from the output data set, the choice of EQUALS or NOEQUALS can have an effect on which observations are removed.

EQUALS is the default. For observations that have identical BY-variable values, EQUALS maintains the order from the input data set in the output data set. NOEQUALS does not necessarily preserve this order in the output data set. NOEQUALS can save CPU time and memory resources.

FIRST, LAST, processing in the DATA step can also be used to remove duplicate observations in a SAS data set.

Review the related comparative example:

- “Removing Duplicate Observations Efficiently” on page 816.

**Additional Features**

Depending on your operating environment, you might be able to use additional sorting options, called host sort utilities. Host sort utilities are third-party sort packages. In some cases, using a host sort utility might be more efficient than using the SAS sort utility with PROC SORT.

SAS uses the values that are set for the SORTPGM=, SORTCUTP=, SORTCUT= and SORTNAME= system options to determine which sort utility to use.

---

**Quiz**

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. When the following program is submitted, what is the value of `FIRST.Product_Line` for the third observation in the data set `Work.Products`?

   ```sas
   data new.products;
   set work.products
   by product_line notsorted;
   run;
   ```

<table>
<thead>
<tr>
<th>Table 23.19 SAS Data Set Work.Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

   a 1
   b 3
   c 0
   d Clothes & Shoes
2 Which option is used with the SORT procedure to store only the BY variables and the observation numbers in temporary files?
   a NOTSORTED
   b GROUPFORMAT
   c TAGSORT
   d SORTEDBY=

3 Which of the following is not an advantage of BY-group processing with an index that is based on the BY variables?
   a The data can be sequenced without using the SORT procedure.
   b There is no need to re-sort a data set when observations are modified or added.
   c It is generally more efficient than reading a sorted data set sequentially.
   d The data can be sequenced by different variables if multiple indexes are used.

4 Which SORT procedure option compares all of the variable values for each observation to those for the previous observation that was written to the output data set?
   a NODUPKEY
   b NODUPRECS
   c_EQUALS
   d NOEQUALS

5 What happens if the workspace that is required for completing a sort is less than or equal to the value that is specified in the SORTSIZE= system option or procedure option?
   a CPU time is increased.
   b I/O is increased.
   c The entire sort can take place in memory.
   d A temporary utility file is created in the Work directory or in a mainframe temporary area.
CHAPTER 24
Querying Data Efficiently

Overview 834
Introduction 834
Objectives 835
Prerequisites 835
Using an Index for Efficient WHERE Processing 836
Accessing Data Sequentially 837
Example 837
Accessing Data Directly 837
Example 838
Benefits and Costs of Using an Index 838
How SAS Selects an Access Method 838
Identifying Available Indexes 839
Example: Identifying One Available Index 840
Example: Identifying Multiple Available Indexes 840
Compound Optimization 841
Example: Composite Index That Can Be Used to Optimize Multiple Conditions 841
Example: Composite Index That Can Be Used to Optimize One Condition 841
Example: Composite Index That Cannot Be Used for Optimizing 842
Identifying Conditions That Can Be Optimized 842
Requirements for Optimizing a Single WHERE Condition 842
WHERE Conditions That Cannot Be Optimized 844
Requirements for Compound Optimization 844
Example: Compound Optimization 844
Estimating the Number of Observations 845
Printing Centile Information 846
Example 846
Comparing Probable Resource Usage 848
How SAS Compares Resource Usage 848
Factors That Affect I/O 848
Subset Size Relative to Data Set Size 849
Number of Pages in the Data File 849
Order of the Data 849
Cost to Uncompress a Compressed File for a Sequential Read 850
Other Factors That Affect Resource Usage 850
Deciding Whether to Create an Index 850
Guidelines for Deciding Whether to Create an Index 850
Example: Selecting Subsets of Various Sizes from Data Sets of Various Sizes 851
Query 1: Small Subset from a Large Data Set 851
Query 2: Large Subset from a Large Data Set 852
Query 3: Small Subset from a Small Data Set 852
Using the Options IDXWHERE= and IDXNAME= to Control Index Usage 852
Overview

Introduction

SAS provides a variety of techniques for querying data that enable you to create the results that you want in different ways. In this chapter, you learn to select the most efficient query techniques from those listed below, based on comparisons of resource usage.
**Task** | **Techniques**
--- | ---
selecting a subset | • WHERE statement that references a data set that has been indexed
creating a detail report | • PRINT procedure
creating a summary report for one class variable or a combination of class variables | • MEANS procedure (or SUMMARY procedure)
 | • TABULATE procedure
 | • REPORT procedure
 | • SQL procedure
 | • DATA step

**Note:** This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SAS SPD Engine to improve performance, see the SAS documentation.

**Objectives**

In this chapter, you learn to

• identify the costs and benefits of using an index
• identify the factors that affect whether SAS uses an index for WHERE processing
• determine whether SAS is likely to use an index to process a particular WHERE expression
• identify the main features of compound optimization
• identify the effect of indexing and order of data on WHERE processing
• print centile information for a data file
• identify the relative efficiency of the PRINT procedure and the SQL procedure for creating detail reports
• identify the relative efficiency of five tools for summarizing data for one categorical variable
• identify the relative efficiency of three ways of using the MEANS procedure to summarize data for selected combinations of categorical variables.

**Prerequisites**

Before beginning this chapter, you should complete the following chapters:

**Part 1: SQL Processing with SAS**

• Chapter 1, “Performing Queries Using PROC SQL,” on page 3
• Chapter 2, “Performing Advanced Queries Using PROC SQL,” on page 25
• Chapter 3, “Combining Tables Horizontally Using PROC SQL,” on page 79
• Chapter 4, “Combining Tables Vertically Using PROC SQL,” on page 125
• Chapter 5, “Creating and Managing Tables Using PROC SQL,” on page 159
• Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221
Using an Index for Efficient WHERE Processing

When processing a WHERE expression, SAS determines which of the following access methods is likely to be most efficient:

### Sequential Access

<table>
<thead>
<tr>
<th>SAS Data Set</th>
<th>Anderson 09JAN2000 X 34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baker 14CCT2001 X 54</td>
</tr>
<tr>
<td></td>
<td>Davis 30MAR2000 Y 49</td>
</tr>
<tr>
<td></td>
<td>Edwards 28JUN2002 X 52</td>
</tr>
<tr>
<td>Smith</td>
<td>15JAN2000 Y 62</td>
</tr>
<tr>
<td>Yates</td>
<td>04AUG2002 X 59</td>
</tr>
</tbody>
</table>

SAS searches through all observations sequentially (in the order in which they are stored in the data file).

### Direct Access

<table>
<thead>
<tr>
<th>SAS Data Set</th>
<th>Anderson 09JAN2000 X 34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baker 14CCT2001 X 54</td>
</tr>
<tr>
<td></td>
<td>Davis 30MAR2000 Y 49</td>
</tr>
<tr>
<td></td>
<td>Edwards 28JUN2002 X 52</td>
</tr>
<tr>
<td>Smith</td>
<td>15JAN2000 Y 62</td>
</tr>
<tr>
<td>Yates</td>
<td>04AUG2002 X 59</td>
</tr>
</tbody>
</table>

SAS uses an index to access specific observations directly. Using an index to process a WHERE expression is referred to as optimizing the WHERE expression.

Using an index to process a WHERE expression improves performance in some situations but not in others. For example, it is more efficient to use an index to select a small subset than a large subset. In addition, an index conserves some resources at the expense of others.
By deciding whether to create an index, you also play a role in determining which access method SAS can use. When your program contains a WHERE expression, you should determine which access method is likely to be more efficient. If direct access is likely to be more efficient, you can make sure that an index is available by creating a new index or by maintaining an existing index.

To help you make a more effective decision about whether to create an index, this topic and the next few topics provide you with a closer look at the following:

- steps that SAS performs for sequential access and direct access
- benefits and costs of index usage
- steps that SAS performs to determine which access method is most efficient
- factors affecting resource usage for indexed access
- guidelines for deciding whether to create, use, and maintain an index.

Note: You should already know how to create and maintain indexes by using the INDEX= data set option in the DATA statement, the DATASETS procedure, and the SQL procedure. To review these SAS elements, see Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221 and Chapter 13, “Creating Samples and Indexes,” on page 451.

Note: SAS can also use an index to process a BY statement. BY processing enables you to process observations in a specific order according to the values of one or more variables that are specified in a BY statement. Indexing a data file enables you to use a BY statement without sorting the data file. When you specify a BY statement, SAS checks the value of the Sorted flag. If the Sorted flag is set to NO, then SAS looks for an appropriate index. If an appropriate index exists, the software automatically retrieves the observations from the data file in indexed order. Using an index to process a BY statement might not always be more efficient than simply sorting the data file. Therefore, using an index for a BY statement is generally for convenience, not for performance.

---

**Accessing Data Sequentially**

When accessing observations sequentially, SAS must search through all observations in the order in which they are stored in the data file.

**Example**

Suppose you want to create a new data set, `Company.D02jul2000`, that contains a subset of observations from the data set `Company.Dates`. The following DATA step uses a WHERE statement to select all observations in which the value of `Date_ID` is `02JUL2000`:

```sas
data company.d02jul2000;
  set company.dates;
  where date_id='02JUL2000'd;
run;
```

The data set `Company.Dates` does not have an index defined on the variable `Date_ID`, so SAS must use **sequential access** to process the WHERE statement.

---

**Accessing Data Directly**

When using an index for WHERE processing, SAS goes straight to each observation that contains the value without having to read every observation in the data set.
Example

Suppose you have defined an index on the variable Date_ID in the Company.Dates data set. This time, when you submit the following DATA step, SAS uses the index to process the WHERE statement:

```sas
data company.d02jul2000;
  set company.dates;
  where date_id='02JUL2000'd;
run;
```

The process of retrieving data via an index (direct access) is more complicated than sequentially processing data, so direct access requires more CPU time per observation retrieved than sequential access. However, for a small subset, using an index can decrease the number of pages that SAS has to load into input buffers, which reduces the number of I/O operations.

Note: When the values in the data set are sorted in the order in which they occur in the index, the qualified observations are adjacent to each other. In this situation, SAS loads fewer pages into the input buffer than if the data is randomly distributed throughout the data set. Therefore, fewer I/O operations are required when the data set is sorted. However, there is a greater chance that SAS will need to load the same page of data multiple times, and that more I/O operations will be required, when the data values are distributed randomly and more than one value needs to be selected to satisfy the WHERE statement (using an operator other than the equals operator).

Benefits and Costs of Using an Index

As the preceding examples show, both benefits and costs are associated with using an index. Weighing these benefits and costs is an important part of deciding whether using an index is efficient.

The main benefits of using an index include the following:
- Provides fast access to a small subset of observations
- Returns values in sorted order
- Can enforce uniqueness.

The main costs of using an index include the following:
- Requires extra CPU cycles and I/O operations for creating and maintaining an index
- Requires increased CPU time and I/O activity for reading the data
- Requires extra disk space for storing the index file
- Requires extra memory for loading index pages and extra code for using the index.

Note: SAS requires additional buffers when an index file is used. When a data file is opened, SAS opens the index file, but not the indexes. Buffers are not required unless SAS uses an index, but SAS allocates the buffers to prepare for using the index. The number of levels of an index determines the number of buffers that are allocated. The maximum number of buffers is three for data files that are open for input; the maximum number is four for data that is open for update. These buffers can be used for other processing if they are not used for indexes.

How SAS Selects an Access Method

When SAS processes a WHERE expression, it first determines whether to use direct access or sequential access by performing the following steps:
1 identifies available indexes
2 identifies conditions that can be optimized
3 estimates the number of observations that qualify
4 compares probable resource usage for both methods.

In the next few sections, each step of this process is explained in detail.

---

**Identifying Available Indexes**

The first step for SAS is to determine whether there are any existing indexes that might be used to process the WHERE expression. Specifically, SAS checks the variable in each condition in the WHERE expression to determine whether the variable is a key variable in an index.

SAS can use either a simple index or a composite index to optimize a WHERE expression. To be considered for use in optimizing a single WHERE condition, one of the following requirements must be met:

- the variable in the WHERE condition is the key variable in a simple index
- the variable in the WHERE condition is the first key variable in a composite index.

SAS identifies all indexes that are defined on any variable in the WHERE expression. However, no matter how many indexes are available, SAS can use only one index to process a WHERE expression. So, if multiple indexes are available, SAS must choose between them.

When SAS looks for available indexes, there are three possible outcomes:

<table>
<thead>
<tr>
<th>If...</th>
<th>Then...</th>
</tr>
</thead>
<tbody>
<tr>
<td>there is no index defined on any variables in the WHERE expression</td>
<td>SAS does not continue with the decision process. SAS must use sequential access to process the WHERE expression.</td>
</tr>
<tr>
<td>there is one available index that is defined on one or more variables in the WHERE expression</td>
<td>SAS continues with the decision process and determines whether using the available index is more efficient than using sequential access.</td>
</tr>
<tr>
<td>there are multiple available indexes, each of which is defined on one or more of the variables in the WHERE expression</td>
<td>SAS continues with the decision process. SAS must choose between the available indexes in the next few steps. SAS tries to select the index that satisfies the most conditions and that selects the smallest subset of observations.</td>
</tr>
</tbody>
</table>

Note: If a program specifies both a WHERE expression and a BY statement, SAS looks for one index that satisfies conditions for both. If such an index is not found, the
BY statement takes precedence so that SAS can ensure that the data is returned in sorted order. With a BY statement, SAS cannot use an index to optimize a WHERE expression if the optimization invalidates the BY order.

Example: Identifying One Available Index

Suppose you submit a program that contains the following WHERE statement, and suppose that the data set has one index, as shown below:

<table>
<thead>
<tr>
<th>WHERE Statement</th>
<th>Available Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>where delivery_date='02jul2000'd</td>
<td>simple index defined on Delivery_Date</td>
</tr>
</tbody>
</table>

This WHERE expression has one condition, and the variable in that condition (Delivery_Date) is the key variable in the simple index. If all other requirements for optimization are met in later steps, then SAS can use this index to optimize the WHERE expression.

Likewise, if the only available index is a composite index in which Delivery_Date is the first key variable, then SAS can use the index if all other requirements for optimization are met.

Even if a WHERE statement has multiple conditions, SAS can use either a simple index or a composite index to optimize just one of the conditions. For example, suppose your program contains a WHERE statement that has two conditions, and suppose that the data set has one index, as shown below:

<table>
<thead>
<tr>
<th>WHERE Statement</th>
<th>Available Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>where order_date='01jan2000'd and delivery_date='02jul2000'd;</td>
<td>simple index defined on Delivery_Date</td>
</tr>
</tbody>
</table>

Assuming that all other requirements for optimization are met, SAS can use this index to optimize the second condition in this WHERE expression.

Example: Identifying Multiple Available Indexes

Suppose your program contains a WHERE statement with two conditions, and suppose that each condition references a key variable in a different index, as shown below:

<table>
<thead>
<tr>
<th>WHERE Statement</th>
<th>Available Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>where order_date='01jan2000'd and delivery_date='02jul2000'd;</td>
<td>simple index defined on Order_Date, simple index defined on Delivery_Date</td>
</tr>
</tbody>
</table>

Although two indexes are available, SAS can use only one index to optimize a WHERE statement. In a later step of the process, SAS will try to select the index that satisfies the most conditions and that selects the smallest subset of observations.
Compound Optimization

SAS usually uses an index to process just one condition, no matter how many conditions and variables a WHERE expression contains. However, in a process called compound optimization, SAS can use a composite index to optimize multiple conditions on multiple variables, which are joined with a logical operator such as AND. Constructing your WHERE expression to take advantage of multiple key variables in a single index can greatly improve performance.

In order for compound optimization to occur, at least the first two key variables in the composite index must be used in the WHERE conditions. Later in this chapter, you will learn about other requirements that must be met in order for compound optimization to occur.

Note: The WHERE expression can also contain non-indexed variables, and the key variables and non-indexed variables can appear in any order in the expression.

Example: Composite Index That Can Be Used to Optimize Multiple Conditions

Suppose your program contains a WHERE statement that has two conditions, and suppose that each condition references one of the first two key variables in a composite index:

```
WHERE order_date='01jan2000'd and delivery_date='02jul2000'd;
```

Because the two variables that are referenced in the WHERE expression are the first two key variables in the composite index, SAS can use the composite index for compound optimization if the WHERE conditions meet all other requirements for optimization.

Example: Composite Index That Can Be Used to Optimize One Condition

The following WHERE statement also contains two conditions, and each condition references one of the variables in the composite index:
WHERE Statement | Available Index
--- | ---
where order_date='01jan2000'd and product_id='220101400106'; | composite index defined on the following variables:
- Order_Date (first key variable)
- Delivery_Date (second key variable)
- Product_ID (third key variable)

As in the previous WHERE statement, Order_Date is the first key variable in the index. However, in this situation, the composite index can be used to optimize only the first condition. The second condition references the third key variable, Product_ID, but the WHERE expression does not reference the second key variable, Delivery_Date. Without a reference to both the first and second key variables, compound optimization cannot occur.

**Example: Composite Index That Cannot Be Used for Optimizing**

Now suppose your program contains a WHERE statement that references only the second and third key variables in the composite index, as shown below:

WHERE Statement | Available Index
--- | ---
where delivery_date='02jul2000'd and product_id='220101400106'; | composite index defined on the following variables:
- Order_Date (first key variable)
- Delivery_Date (second key variable)
- Product_ID (third key variable)

In this situation, SAS cannot use the index for optimization at all because the WHERE statement does not reference the first key variable.

**Identifying Conditions That Can Be Optimized**

In addition to containing key variables, WHERE conditions must meet other requirements in order to be candidates for optimization. SAS considers using an index only for WHERE conditions that contain certain operators and functions. Therefore, the next step for SAS is to consider the operators and functions in the conditions that contain key variables.

**Requirements for Optimizing a Single WHERE Condition**

SAS considers using an index for a WHERE condition that contains any of the following operators and functions:

*Note:* For all of the following examples, assume that the data set has simple indexes on the variables Quarter, Date_ID, and Region.
### Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparison operators and the IN operator</td>
<td><code>where quarter = '1998Q1'</code>;</td>
</tr>
<tr>
<td></td>
<td><code>where date_id &lt; '03JUL2000'd;</code></td>
</tr>
<tr>
<td></td>
<td><code>where quarter in ('1998Q2','1998Q3');</code></td>
</tr>
<tr>
<td>comparison operators with NOT</td>
<td><code>where quarter ne '1999Q1'</code>;</td>
</tr>
<tr>
<td></td>
<td><code>where quarter not in ('1999Q1','1999Q4');</code></td>
</tr>
<tr>
<td>comparison operators with the colon modifier</td>
<td><code>where quarter =: '1998'</code>;</td>
</tr>
<tr>
<td>You can add a colon modifier (:) to any comparison</td>
<td></td>
</tr>
<tr>
<td>operator to compare only a specified prefix of a</td>
<td></td>
</tr>
<tr>
<td>character string.</td>
<td></td>
</tr>
<tr>
<td>The colon modifier cannot be used with PROC SQL;</td>
<td></td>
</tr>
<tr>
<td>use the LIKE operator instead.</td>
<td></td>
</tr>
<tr>
<td>CONTAINS operator</td>
<td><code>where quarter contains 'Q4'</code>;</td>
</tr>
<tr>
<td>fully-bounded range conditions that specify both</td>
<td><code>where '01Jan1999'd &lt; date_id &lt; '31Dec1999'd;</code></td>
</tr>
<tr>
<td>an upper and lower limit, which includes the</td>
<td><code>where date_id between '01Jan1999'd and '31Dec1999'd</code></td>
</tr>
<tr>
<td>BETWEEN-AND operator</td>
<td></td>
</tr>
<tr>
<td>pattern-matching operator LIKE</td>
<td><code>where quarter like '%Q%'</code>;</td>
</tr>
<tr>
<td>IS NULL or IS MISSING operator</td>
<td><code>where quarter is null;</code></td>
</tr>
<tr>
<td></td>
<td><code>where quarter is missing;</code></td>
</tr>
</tbody>
</table>

### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIM function</td>
<td><code>where trim(region) = 'Queensland';</code></td>
</tr>
<tr>
<td>SUBSTR function in the form of</td>
<td><code>where substr(quarter,1,4) = '1998';</code></td>
</tr>
<tr>
<td><strong>WHERE</strong> SUBSTR</td>
<td></td>
</tr>
<tr>
<td>(variable.position,length)='string';</td>
<td></td>
</tr>
<tr>
<td>with these conditions:</td>
<td></td>
</tr>
<tr>
<td>□ position = 1</td>
<td></td>
</tr>
<tr>
<td>□ length is less than or equal to the length of</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td></td>
</tr>
<tr>
<td>□ length is equal to the length of the string.</td>
<td></td>
</tr>
</tbody>
</table>

**CAUTION:**

Most but not all of the requirements listed above also apply to compound optimization. Requirements for compound optimization are covered later in this chapter. △
WHERE Conditions That Cannot Be Optimized

SAS does not use an index to process a WHERE condition that contains any of the elements listed below.

Note: For all of the following examples, assume that the data set has simple indexes on the variables Date_ID, Quarter, and Quantity.

<table>
<thead>
<tr>
<th>Element in WHERE Condition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>any function other than TRIM or SUBSTR</td>
<td>where weekday(date_id)=2;</td>
</tr>
<tr>
<td>a SUBSTR function that searches a string beginning at any position after the first</td>
<td>where substr(quarter,6,1)=‘1’;</td>
</tr>
<tr>
<td>the sounds-like operator (=* )</td>
<td>where quarter=‘1900Q0’;</td>
</tr>
<tr>
<td>arithmetic operators</td>
<td>where quantity=quantity+1;</td>
</tr>
<tr>
<td>a variable-to-variable condition</td>
<td>where quantity gt threshold;</td>
</tr>
</tbody>
</table>

Requirements for Compound Optimization

Most of the same operators that are acceptable for optimizing a single condition are also acceptable for compound optimization. However, compound optimization has special requirements for the operators that appear in the WHERE expression:

- The WHERE conditions must be connected by using either the AND operator or, if all conditions refer to the same variable, the OR operator.
- At least one of the WHERE conditions that contains a key variable must contain the EQ or IN operator.

Also, SAS cannot perform compound optimization for WHERE conditions that include any of the following:

- the CONTAINS operator
- the pattern-matching operators LIKE and NOT LIKE
- the IS NULL and IS MISSING operators
- any functions.

Example: Compound Optimization

Suppose your program contains the following WHERE statement, which selects all people whose name is John Smith. The WHERE statement contains two conditions, each of which references a different variable:

```
where lastname eq 'Smith' and
    firstname eq 'John';
```

Suppose Lastname is the first key variable and Firstname is the second key variable in a compound index. This WHERE statement meets all requirements for compound optimization:

- The WHERE expression references at least the first two key variables in one composite index.
The two WHERE conditions are connected by the AND operator.

At least one of the conditions contains the EQ operator.

If the two conditions in the WHERE statement are reversed, as shown below, the statement still meets all requirements for compound optimization. The order in which the key variables appear does not matter.

```
where firstname eq 'John' and
    lastname eq 'Smith';
```

Now suppose that the conditions in the WHERE statement are joined by the operator OR instead of AND:

```
where firstname eq 'John' or
    lastname eq 'Smith';
```

These conditions cannot be optimized because they are joined by OR but they do not reference the same variable.

---

**Estimating the Number of Observations**

It is more efficient to use indexed access for a small subset and to use sequential access for a large subset. Therefore, after identifying any available indexes and evaluating the conditions in the WHERE expression, SAS estimates the number of observations that will be qualified by the index. Whether or not SAS uses an index depends on the percentage of observations that are qualified (the size of the subset relative to the size of the data set), as shown below:

- **If the subset is less than 3% of the data set**, direct access is almost certainly more efficient than sequential access, and SAS will use an index. In this situation, SAS does not go on to compare probable resource usage.

- **If the subset is between 3% and 33% of the data set**, direct access is likely to be more efficient than sequential access, and SAS will probably use an index.

- **If the subset is greater than 33% of the data set**, it is less likely that direct access is more efficient than sequential access, and SAS might or might not use an index.

When multiple indexes exist, SAS selects the one that appears to produce the fewest qualified observations (the smallest subset). SAS does this even when each index returns a subset that is less than 3% of the data set.
Printing Centile Information

To help SAS estimate the number of observations that would be selected by a WHERE expression, each index stores 21 statistics called cumulative percentiles, or centiles. Centiles provide information about the distribution of values for the indexed variable. Understanding the distribution of values in a data set can help you improve the efficiency of WHERE processing in your programs. You can print centile information for an indexed data file by specifying the CENTILES option in either of these places:

- the CONTENTS procedure
- the CONTENTS statement in the DATASETS procedure.

PROC CONTENTS <options>;  PROC DATASETS <options>;
RUN;                     CONTENTS <options>;
QUIT;

Example

The following SAS program prints centile information for the data set Company.Organization:

    proc contents data=company.organization centiles;
    run;

Partial output from this program is shown below. As indicated on the left, an index is defined on the variable Employee_ID. The 21 centile values are listed on the right.
Figure 24.1  Partial PROC CONTENTS Output

<table>
<thead>
<tr>
<th>Index</th>
<th>Unique Option</th>
<th>Owned by IC</th>
<th>Update Centiles</th>
<th>Current Update Percent</th>
<th># of Unique Values</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee_ID</td>
<td>YES</td>
<td>YES</td>
<td>5</td>
<td>0</td>
<td>1049</td>
<td>120101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120152</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120205</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120257</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120310</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120362</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120415</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120467</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120520</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120572</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120625</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120677</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120729</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120782</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120834</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120887</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120939</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120992</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>121044</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>121097</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99999999</td>
</tr>
</tbody>
</table>

The 21 centile values consist of the following:
Comparing Probable Resource Usage

Once SAS estimates the number of qualified observations and selects the index that qualifies the fewest observations, SAS must then determine whether it is faster (more efficient) to satisfy the WHERE expression by using the index or by reading all of the observations sequentially. Specifically, SAS predicts how many I/O operations will be required in order to satisfy the WHERE expression for each of the access methods. Then it compares the two resource costs.

Note: Remember, if SAS estimates that a subset contains fewer than 3% of the observations in the data set, SAS does not need to estimate resource usage. In this situation, SAS will use the index to process the WHERE statement.

How SAS Compares Resource Usage

To compare resource usage, SAS performs the following steps:

1. SAS predicts how many I/O operations will be required if it uses the index to satisfy the WHERE expression. To do so, SAS positions the index at the first entry that contains a qualified value. In a buffer management simulation that takes into account the current number of available buffers, the RIDs (record identifiers) on that index page are processed, indicating how many I/Os will be required in order to read the observations in the data file.

2. SAS calculates the I/O cost of a sequential pass of the entire data file.

3. SAS compares the two resource costs and determines which access method has a lower cost.

Note: If comparing resource costs results in a tie, SAS chooses the index.

Factors That Affect I/O

Several factors affect the number of I/O operations that are required for WHERE processing, including the following:

- subset size relative to data set size
- number of pages in the data file
- order of the data
- cost to uncompress a compressed file for a sequential read.

These factors are discussed in more detail below.

### Subset Size Relative to Data Set Size

As explained earlier in this chapter, SAS is more likely to use an index to access a small subset of observations. The process of retrieving data with an index is inherently more complicated than sequentially processing the data. This is why using an index requires more I/O operations and CPU time when a large subset is read.

For small subsets, however, the benefit of reading only a few observations outweighs the cost of the complex processing. The smaller the subset, the larger the performance gains. Remember that SAS will use an index if the subset is less than 3% of the data set, and SAS will probably use an index if the subset is between 3% and 33% of the data set.

### Number of Pages in the Data File

For a small data file, sequential processing is often just as efficient as index processing. If the data file’s page count is less than three pages, then sequential access is faster even if the subset is less than 3% of the entire data set.

*Note:* The amount of data that can be transferred to one buffer in a single I/O operation is referred to as page size. To see how many pages are in a data file, use either the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure. For more information about reporting the page size for a data file, see Chapter 20, “Controlling Memory Usage,” on page 689.

### Order of the Data

The order of the data (sort order) affects the number of I/O operations as described below:

<table>
<thead>
<tr>
<th>Order of the Data</th>
<th>Effect on I/O Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>observations are <em>randomly</em> distributed throughout the data file</td>
<td>The observations are located on a larger number of data file pages. An I/O operation is required each time that SAS loads a page. Therefore, the more random the data in the data file, the more I/O operations are needed to use the index.</td>
</tr>
<tr>
<td>observations are <em>sorted</em> on the indexed variable(s)</td>
<td>The data is ordered more like the index (in ascending value order), and the observations will be located on fewer data file pages. Therefore, the less random the data in the data file, the fewer I/O operations are needed to use the index.</td>
</tr>
</tbody>
</table>

*Note:* In general, sorting the data set by the key variable before indexing will result in greater efficiency. The more ordered the data file is with respect to the key variable, the more efficient the use of the index. If the data file has more than one index, then
sorting the data by the most frequently used key variable is most efficient. Sorting the
data set results in more efficient WHERE processing even when SAS does not use an
index. To learn more about sorting and efficiency, see Chapter 23, “Selecting Efficient
Sorting Strategies,” on page 785.

Cost to Uncompress a Compressed File for a Sequential Read

When SAS reads a compressed data file, SAS automatically uncompresses the
observations as they are read into the program data vector. This requires additional
CPU resources, but fewer I/O operations are required because there are fewer data set
pages. When performing a sequential read of a compressed data file, SAS must
uncompress all observations in the file. However, when using direct access, SAS must
uncompress only the qualified observations. Therefore, the resource cost of
uncompressing observations is greater for a sequential read than for direct access.

Note: Compressing a file is a process that reduces the number of bytes that are
required for representing each observation. By default, a SAS data file is not
compressed. For more information about compressing files, see Chapter 21, “Controlling
Data Storage Space,” on page 707.

Other Factors That Affect Resource Usage

Data type and length are two other factors that can affect index efficiency. Numeric
key variables typically result in more CPU usage than character key variables, because
numeric variables must be converted to collatable formats (formats that can be
sequenced) when values are read into the index or retrieved from the index. Character
values are already intrinsically collatable, so they do not have to be converted. A page
cannot contain as many values if the values are longer. When the values are longer, the
index structure is not as efficient, which might lead to more I/O and search time.

Deciding Whether to Create an Index

In previous sections, you learned how SAS determines whether sequential access or
direct access is likely to be most efficient for WHERE processing. You also learned
about a variety of factors that you can assess to determine which access method is most
efficient. Once you have made your determination, you can use the following guidelines
to decide whether it is efficient to create an index.

Guidelines for Deciding Whether to Create an Index

- Minimize the number of indexes to reduce disk storage and update costs. Create
  indexes only on variables that are often used in queries or (when data cannot be
  sorted) in BY-group processing.
- Create an index when you intend to retrieve a small subset of observations from a
  large data file.
- Do not create an index if the data file’s page count is less than three pages. It is
  faster to access the data sequentially.
- Create indexes on variables that are discriminating. Discriminating variables
  have many different values that precisely identify observations. A WHERE
expression that subsets based on a discriminating variable results in a smaller subset than a WHERE expression that references a non-discriminating variable (a variable that has only a few values).

- To reduce the number of I/O operations that are performed when you create an index, first sort the data by the key variable. Then, to improve performance, maintain the data file in sorted order by the key variable.

  Note: If you choose not to use an index and the data set is large, it is still more efficient to sort the data set on the variable(s) that are specified in the WHERE statement.

- Consider how often your applications use an index. An index must be used often in order to compensate for the resources that are used in creating and maintaining it.

- Consider the cost of an index for a data file that is frequently changed.

- When you create an index to process a WHERE expression, do not try to create one index that is used to satisfy all queries.

Let’s look at three sample queries to see how you can apply the guidelines that are listed in the previous section. These queries illustrate the effect of one factor—the size of the subset relative to the size of the data set—on the choice of an access method. For each query, you will learn:

- which access method SAS is likely to select
- whether you could improve performance by creating an index.

Example: Selecting Subsets of Various Sizes from Data Sets of Various Sizes

Suppose you are working with the following two data sets, each of which contains information about a company’s orders:

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Pages</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company.Orders_large</td>
<td>285,500</td>
<td>19,033,380</td>
</tr>
<tr>
<td>Company.Orders_small</td>
<td>2</td>
<td>140</td>
</tr>
</tbody>
</table>

You want to create queries to generate three subset detail reports, one for each of the following types of subsets:

- small subset from a large data set
- large subset from a large data set
- small subset from a small data set.

In all three queries, the WHERE expression specifies the variable Order_Date. You know that this variable will be used frequently in queries for the company, and that it is a discriminating variable. According to the guidelines in the previous section, these are both criteria for creating an index on the variable. However, there is currently no index defined on this variable in either data set.

Query 1: Small Subset from a Large Data Set

The first report that you want to generate shows all orders in Company.Orders_large that were made on January 10, 1998. Your query is shown below, along with the subset size that you have estimated:
Query 2: Large Subset from a Large Data Set

The second report shows all orders in `Company.Orders_large` that were made before January 1, 1998. Your query and the estimated subset size are shown below:

<table>
<thead>
<tr>
<th>Query</th>
<th>Subset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data _null_;</code></td>
<td></td>
</tr>
<tr>
<td><code>set company.orders_large;</code></td>
<td>2232 observations (out of 19,033,380) =</td>
</tr>
<tr>
<td><code>where order_date='10JAN1998'd;</code></td>
<td>&lt; .02% of the data set</td>
</tr>
<tr>
<td><code>run;</code></td>
<td></td>
</tr>
</tbody>
</table>

Because the subset is less than 3% of the entire data set, using an index on `Order_Date` should be more efficient than using sequential access. SAS will use an index for WHERE processing, if an index is available. To improve performance, you should create an index on `Order_Date` before running this program.

Query 3: Small Subset from a Small Data Set

The third report shows all orders in the smaller data set `Company.Orders_small` that were made on June 30, 1998. Your query and the estimated subset size are shown below:

<table>
<thead>
<tr>
<th>Query</th>
<th>Subset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data _null_;</code></td>
<td></td>
</tr>
<tr>
<td><code>set company.orders_small;</code></td>
<td>12,752,365 observations (out of 19,033,380) =</td>
</tr>
<tr>
<td><code>where order_date='30JUN1998'd;</code></td>
<td>approximately 67% of the data set</td>
</tr>
<tr>
<td><code>run;</code></td>
<td></td>
</tr>
</tbody>
</table>

Because the subset is less than 3% of the entire data set, SAS will use the index for WHERE processing. However, the data file's page count is less than three pages, so it is more efficient to use sequential access. In this situation, it is best not to create an index.

Using the Options IDXWHERE= and IDXNAME= to Control Index Usage

In most situations, it is best to let SAS determine whether or not to use an index for WHERE processing. However, sometimes you might want to control whether or not
SAS uses an existing index. For example, if you know that your query will select a large subset and that indexed access will therefore not be efficient, you can tell SAS to ignore any index and to satisfy the conditions of the WHERE expression with a sequential search of the data set. Or, if your query will select a small subset and there are multiple available indexes, you can make sure that SAS uses a particular index to process your WHERE statement. Finally, you might want to force SAS to use (or not use) an index when you are benchmarking.

You should be familiar with the data set options IDXWHERE= and IDXNAME=, which you can use to control index usage:

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDXWHERE=</td>
<td>specifies whether or not SAS should use an index to process the WHERE expression, no matter which access method SAS estimates is faster. You cannot use IDXWHERE= to override the use of an index for processing a BY statement.</td>
</tr>
<tr>
<td>IDXNAME=</td>
<td>causes SAS to use a specific index.</td>
</tr>
</tbody>
</table>

**CAUTION:**

You can use either IDXWHERE= or IDXNAME=, but not both at the same time, to control index usage. △

**Note:** For more information about the IDXWHERE= and IDXNAME= data set options, see Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221. △

**Specifying MSGLEVEL=I to Determine Whether SAS Is Using an Index**

To determine whether SAS is using an index to process a WHERE expression, you can specify I as the value of the MSGLEVEL= system option. Using MSGLEVEL=I causes SAS to display information about index usage in the SAS log.

**Note:** To make the most efficient use of resources, use MSGLEVEL=I only for debugging and for verifying index usage. △

**Note:** For more information about the MSGLEVEL= system option, see Chapter 13, “Creating Samples and Indexes,” on page 451 or Chapter 6, “Creating and Managing Indexes Using PROC SQL,” on page 221. △

**Example: Using IDXWHERE=NO to Prevent Index Usage**

Suppose you write the following query, which lists all employees who work in the Sales department of a company:

```sas
proc print data=company.organization;
  where department='Sales';
run;
```

Now suppose an index is defined on the variable Department in the data set Company.Organization. You know that Department has the value Sales in 65% of the observations, so it is not efficient for SAS to use an index for WHERE processing. To ensure that SAS does not use an index, specify IDXWHERE=NO after the data set
name. At the beginning of the program, you can also add an OPTIONS statement that specifies MSGLEVEL=I to display a message about index usage in the SAS log. The revised program is shown below:

```sas
options msglevel=i;
proc print data=company.organization (idxwhere=no);
   where department='Sales';
run;
```

When you run this program, the SAS log indicates that the index was not used for processing.

**Table 24.1 SAS Log**

```
INFO: Data set option (IDXWHERE=NO) forced a sequential pass of the data rather than use of an index for where-clause processing.
```

---

### Comparing Procedures That Produce Detail Reports

When you want to use a query to produce a detail report, you can choose between the *PRINT procedure* and the *SQL procedure*:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC PRINT</td>
<td>□ produces data listings quickly</td>
</tr>
<tr>
<td></td>
<td>□ can supply titles, footnotes, and column sums</td>
</tr>
<tr>
<td>PROC SQL</td>
<td>□ combines SQL and SAS features such as formats</td>
</tr>
<tr>
<td></td>
<td>□ can manipulate data and create a SAS data set in the same step that creates the report</td>
</tr>
<tr>
<td></td>
<td>□ can produce column and row statistics</td>
</tr>
<tr>
<td></td>
<td>□ does not offer as much control over output as PROC PRINT</td>
</tr>
</tbody>
</table>

To perform a particular task, a single-purpose tool like PROC PRINT generally uses fewer computer resources than a multi-purpose tool like PROC SQL. However, PROC SQL often requires fewer and shorter statements to achieve the results that you want.

To illustrate the differences in resource usage between PROC PRINT and PROC SQL, let’s look at some sample queries.

---

### Example: Using PROC PRINT and PROC SQL to Create Detail Reports

Suppose you are working with the data set *Company.Products* and that you want to generate four types of detail reports:

□ simple detail report

□ subset detail report

□ sorted detail report

□ sorted subset detail report.
For the first three reports, the PROC PRINT program is likely to use fewer resources than the PROC SQL program. For the last report, the resource usage for the two programs is likely to be about the same.

### Report 1: Simple Detail Report

The simple detail report lists the product ID, product name, and supplier name for all products. The PROC PRINT program and PROC SQL program for producing this report are shown below:

**PROC PRINT**

```latex
proc print data=company.products;
  var product_id product_name supplier_name;
run;
```

**PROC SQL**

```latex
proc sql;
  select product_id product_name supplier_name
  from company.products;
quit;
```

In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

### Report 2: Subset Detail Report

The subset detail report lists the product ID, product name, and supplier name for all products that come from Sweden (SE). The PROC PRINT program and PROC SQL program for producing this report are shown below:

**PROC PRINT**

```latex
proc print data=company.products;
  var product_id product_name supplier_name;
  where supplier_country='SE';
run;
```

**PROC SQL**

```latex
proc sql;
  select product_id product_name supplier_name
  from company.products
  where supplier_country='SE';
quit;
```

Both steps use WHERE processing to subset the data. In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

### Report 3: Sorted Detail Report

The sorted detail report lists the product ID, product name, and supplier name for all products, with observations sorted by the supplier country. The PROC PRINT program and PROC SQL program for producing this report are shown below:
To sort the data, a PROC SORT step has been added to the PROC PRINT program, and an ORDER BY clause has been added to the PROC SQL program. In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

### Report 4: Sorted Subset Detail Report

The sorted subset detail report lists the product ID, product name, and supplier name for all products that come from Sweden (SE), with observations sorted by the supplier name. The PROC PRINT program and PROC SQL program for producing this report are shown below:

<table>
<thead>
<tr>
<th>PROC PRINT</th>
<th>PROC SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc sort data=company.products (keep=Product_ID Product_Name Supplier_Name Supplier_Country) out=product; where supplier_country='SE'; by supplier_name; run;</td>
<td>proc sql;</td>
</tr>
<tr>
<td>proc print data=product; var product_id product_name supplier_name; run;</td>
<td>select product_id product_name supplier_name from company.products where supplier_country='SE' order by supplier_name; quit;</td>
</tr>
</tbody>
</table>

To sort the data, a PROC SORT step has been added to the PROC PRINT program. The PROC SORT step uses the KEEP= option to subset the observations, which improves efficiency. The PROC SQL step uses an ORDER BY clause for sorting and a WHERE clause for subsetting. In this situation, the CPU and memory usage for the PROC PRINT program and the PROC SQL program are about the same.

### Comparing Tools for Summarizing Data

SAS provides a variety of tools for summarizing data. These summarization tools generate similar but not identical output, and they vary in efficiency. This section discusses the relative efficiency of the following summarization tools.
Note: Throughout this section, all references to the MEANS procedure apply also to the SUMMARY procedure.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEANS procedure or SUMMARY</td>
<td>□ computes descriptive statistics for numeric variables</td>
</tr>
<tr>
<td>procedure</td>
<td>□ can produce a printed report and create an output data set</td>
</tr>
<tr>
<td>TABULATE procedure</td>
<td>□ produces descriptive statistics in a tabular format</td>
</tr>
<tr>
<td></td>
<td>□ can produce multidimensional tables with descriptive statistics</td>
</tr>
<tr>
<td></td>
<td>□ can also create an output data set</td>
</tr>
<tr>
<td>REPORT procedure</td>
<td>□ combines features of the PRINT, MEANS, and TABULATE procedures with features of the DATA step in a single report-writing tool that can produce a variety of reports</td>
</tr>
<tr>
<td></td>
<td>□ can also create an output data set</td>
</tr>
<tr>
<td>SQL procedure</td>
<td>□ computes descriptive statistics for one or more SAS data sets or DBMS tables</td>
</tr>
<tr>
<td></td>
<td>□ can produce a printed report or create a SAS data set</td>
</tr>
<tr>
<td>DATA step</td>
<td>□ can produce a printed report</td>
</tr>
<tr>
<td></td>
<td>□ can also create an output data set</td>
</tr>
</tbody>
</table>

Note: You can also use the FREQ procedure and the UNIVARIATE procedure to generate summary data and create summary reports, but these procedures are not covered in this chapter. For more information about any of these summarization tools, see the SAS documentation.

You can use these tools to summarize data at the following levels:

<table>
<thead>
<tr>
<th>Level of Summarization</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>entire data set</td>
<td>any of the above</td>
</tr>
<tr>
<td>one class variable</td>
<td>any of the above</td>
</tr>
<tr>
<td></td>
<td>To group the data, PROC SQL uses the GROUP BY statement, and the DATA step uses the BY statement.</td>
</tr>
<tr>
<td>one or more combinations of class variables</td>
<td>□ PROC MEANS (or PROC SUMMARY)</td>
</tr>
<tr>
<td></td>
<td>□ PROC TABULATE</td>
</tr>
</tbody>
</table>

Comparing Resource Usage across Summarization Tools

When summarizing data for one or more class variables, the tools in each of the following groups are similar in resource usage:

□ **PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE**
However, the relative efficiency of the two groups of tools varies according to the shape of the data, as shown below:

<table>
<thead>
<tr>
<th>Shape of the Data</th>
<th>Most Efficient Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>a relatively small number of distinct</td>
<td>□ PROC MEANS (or PROC SUMMARY)</td>
</tr>
<tr>
<td>combinations of values of the class</td>
<td>□ PROC REPORT</td>
</tr>
<tr>
<td>variable—the summarized data contains</td>
<td>□ PROC TABULATE</td>
</tr>
<tr>
<td>10% or fewer of the observations in the</td>
<td></td>
</tr>
<tr>
<td>original data set</td>
<td></td>
</tr>
<tr>
<td>a relatively large number of distinct</td>
<td>□ PROC SQL</td>
</tr>
<tr>
<td>combinations of values of the class</td>
<td>□ DATA step and PROC SORT</td>
</tr>
<tr>
<td>variable—the summarized data contains</td>
<td></td>
</tr>
<tr>
<td>more than 10% of the observations in the</td>
<td></td>
</tr>
<tr>
<td>original data set</td>
<td></td>
</tr>
</tbody>
</table>

Using a GROUP BY statement in PROC SQL is similar in CPU usage to a PROC SORT step followed by a DATA step.

Comparative Example: Displaying Summary Statistics for One Class Variable

Suppose you want to summarize the data set Retail.Orders by calculating the average quantity of products sold for each order type (each value of the class variable Order_Type). You can use the following techniques to produce the summary report:

1. PROC MEANS
2. PROC REPORT
3. PROC SORT and a DATA step
4. PROC SQL
5. PROC TABULATE.

The following programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for summarizing data for one class variable.

Programming Techniques

1. PROC MEANS

This PROC MEANS step creates an output report that displays the mean quantity of products sold (the analysis variable Quantity) for each order type (the class variable Order_Type).

```plaintext
proc means data=retail.orders
   (keep=order_type quantity)
   mean maxdec=2;
   class order_type;
   var quantity;
run;
```
### PROC REPORT

This PROC REPORT step creates an output report that displays the mean quantity of products sold (the analysis variable `Quantity`) for each order type (the class variable `Order_Type`).

```sas
proc report data=retail.orders
  (keep=order_type quantity);
  column order_type quantity;
  define order_type / group width=13
    'Order Type';
  define quantity / mean format=5.2
    'Average Quantity'
    width=8;
run;
```

### PROC SORT and a DATA Step

This program uses a PROC SORT step and a DATA step to create an output report. The PROC SORT step specifies the variables to be included in the report, sorts the observations by the values of the variable `Order_Type`, and generates the temporary output data set `Orders`. The DATA step calculates the mean quantity of products sold (the analysis variable `Quantity`) for each order type (the class variable `Order_Type`) and displays these values in the temporary output data set.

```sas
proc sort data=retail.orders
  (keep=order_type quantity)
  out=orders;
by order_type;
run;

data _null_;    
set orders;
by order_type;
format average_order 5.2;
if first.order_type then do;
num=0;
sum=0;
end;
num+1;
sum+quantity;
if last.order_type then do;
  average_order=sum / num;
  file print ods=(var=(order_type average_order));
    put ods_;
  end;
run;
```

### PROC SQL

This PROC SQL step creates an output report that displays the mean quantity of products sold (the analysis variable `Quantity`) for each order type (the class variable `Order_Type`).

```sas
proc sql;
  select order_type,
    avg(quantity) label='Average Order'
  format=5.2
  from retail.orders
  group by order_type;
quit;
```
Using PROC MEANS to Display Summary Statistics for Combinations of Class Variables

To produce summary statistics for combinations of class variables, you can use PROC MEANS in the following ways. These techniques differ in resource usage.

<table>
<thead>
<tr>
<th>Combinations of Class Variables</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>all possible combinations:</td>
<td>basic PROC MEANS step</td>
<td>proc means data=lib.dataset mean; class a b c; var salary; output out=summary1 mean=average; run;</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b * c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a * c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a * b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a * b * c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>specific combinations:</td>
<td>TYPES statement in PROC MEANS</td>
<td>proc means data=lib.dataset mean; class a b c; var salary; types a<em>b a</em>c; output out=summary2 mean=average; run;</td>
</tr>
<tr>
<td>a * b and a * c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Combinations of Class Variables

<table>
<thead>
<tr>
<th>Specific Combinations</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \times b) and (a \times c)</td>
<td>NWAY option in multiple PROC MEANS steps</td>
<td><code>proc means data=lib.dataset nway;</code>&lt;br&gt;<code>class a b;</code>&lt;br&gt;<code>var salary;</code>&lt;br&gt;<code>output out=summary3a</code>&lt;br&gt;<code>mean=average;</code>&lt;br&gt;<code>run;</code>&lt;br&gt;<code>proc means data=lib.dataset nway;</code>&lt;br&gt;<code>class a c;</code>&lt;br&gt;<code>var salary;</code>&lt;br&gt;<code>output out=summary3b</code>&lt;br&gt;<code>mean=average;</code>&lt;br&gt;<code>run;</code></td>
</tr>
<tr>
<td>Specific Combinations: (a \times b) and (a \times c)</td>
<td>WHERE= option in the OUTPUT statement in PROC MEANS</td>
<td><code>proc means data=lib.dataset;</code>&lt;br&gt;<code>class a b c;</code>&lt;br&gt;<code>var salary;</code>&lt;br&gt;<code>output out=summary4</code>&lt;br&gt;<code>(where=(_type_ in (5,3)))</code>&lt;br&gt;<code>n=employees</code>&lt;br&gt;<code>mean=average;</code>&lt;br&gt;<code>run;</code></td>
</tr>
</tbody>
</table>

### Comparing Resource Usage across Three Techniques for Using PROC MEANS

The three techniques for summarizing data for specific combinations of class variables (all but the basic PROC MEANS step) differ in resource usage as follows:

- The TYPES statement in a PROC MEANS step uses the fewest resources.
- A program that contains the NWAY option in multiple PROC MEANS steps uses the most resources because SAS must read the data set separately for each PROC MEANS step.
- The WHERE= data set option in a PROC MEANS step uses more resources than the TYPES statement in PROC MEANS because SAS must calculate all possible combinations of class variables before subsetting. However, the WHERE= data set option in PROC MEANS uses fewer resources than the NWAY option in multiple PROC MEANS steps.

Let's take a closer look at how to use a basic PROC MEANS step and the three other techniques that are listed above.

### Using a Basic PROC MEANS Step to Combine All Class Variables

By default, PROC MEANS (or PROC SUMMARY) creates the following:

- an output report that groups data and displays summary statistics for the combination of all the class variables
- an output data set that groups data and displays summary statistics for all possible combinations of the \(n\) class variables (from 1-way to \(n\)-way), as well as for the entire data set.
Example: Displaying Summary Statistics for All Combinations of the Class Variables

Suppose you want to calculate average employee salaries and to group results for the combination of the three class variables `Employee_Country`, `Department`, and `Employee_Gender`.

The following PROC MEANS program creates both a report and a SAS data set:

```plaintext
proc means data=company.organization mean;
  class employee_country department employee_gender;
  var salary;
  output out=summary mean=average;
run;
```

The report groups data and displays summary statistics for the combination of the three class variables. A partial view of the output report is shown below:

```
<table>
<thead>
<tr>
<th>Analysis Variable : Salary Annual Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Country</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>AU</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>BE</td>
</tr>
</tbody>
</table>
```

The output data set groups data and displays summary statistics for both of the following:

- all possible combinations (1-way, 2-way, and 3-way) of the three class variables:
  - `Employee_Gender`
  - `Department`
  - `Department` and `Employee_Gender`
  - `Employee_Country`
  - `Employee_Country` and `Employee_Gender`
  - `Employee_Country` and `Department`
  - `Employee_Country` and `Department` and `Employee_Gender`
- the entire data set.

A partial view of the output data set is shown below:
Each combination of class variables that is used to calculate and group statistics for PROC MEANS is called a type.

For example, the following basic PROC MEANS step specifies the three class variables a, b, and c:

```
proc means data=lib.dataset mean;
   class a b c;
   var salary;
   output out=summary1
       mean=average;
run;
```

This PROC MEANS step generates seven possible types (combinations of the three variables):

<table>
<thead>
<tr>
<th>Variables Combined</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1-way</td>
</tr>
<tr>
<td>b</td>
<td>1-way</td>
</tr>
<tr>
<td>b * c</td>
<td>2-way</td>
</tr>
<tr>
<td>a</td>
<td>1-way</td>
</tr>
<tr>
<td>a * c</td>
<td>2-way</td>
</tr>
<tr>
<td>a * b</td>
<td>2-way</td>
</tr>
<tr>
<td>a * b * c</td>
<td>3-way</td>
</tr>
</tbody>
</table>

SAS uses the _TYPE_ variable to specify the combination of class variables that PROC MEANS uses to calculate the statistics for each observation in the output data set. The _TYPE_ variable has a unique value for each combination. SAS always combines the class variables in a particular sequence, based on the order in which they are listed in the CLASS statement, in order to assign the _TYPE_ values. For example, for each of the seven types (seven possible combinations of three class variables) shown above, SAS assigns a value to _TYPE_ as follows:
As the number of class variables increases, so does the number of types. However, the highest **_TYPE_** value (7, in this example) always indicates the combination of all class variables.

SAS includes the **_TYPE_** variable in the output data set that is generated by PROC MEANS. In the output from the basic PROC MEANS step that was shown in the previous section, you can see that the observations are listed in order of increasing values of the **_TYPE_** variable:

<table>
<thead>
<tr>
<th><strong><em>TYPE</em></strong> Value</th>
<th>Description of Combination</th>
<th>Variables Combined</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rightmost variable only</td>
<td>c</td>
<td>1-way</td>
</tr>
<tr>
<td>2</td>
<td>middle variable only</td>
<td>b</td>
<td>1-way</td>
</tr>
<tr>
<td>3</td>
<td>rightmost variable and middle variable</td>
<td>b * c</td>
<td>2-way</td>
</tr>
<tr>
<td>4</td>
<td>leftmost variable</td>
<td>a</td>
<td>1-way</td>
</tr>
<tr>
<td>5</td>
<td>leftmost variable and rightmost variable</td>
<td>a * c</td>
<td>2-way</td>
</tr>
<tr>
<td>6</td>
<td>leftmost variable and middle variable</td>
<td>a * b</td>
<td>2-way</td>
</tr>
<tr>
<td>7</td>
<td>rightmost variable and middle variable and leftmost variable</td>
<td>a * b * c</td>
<td>3-way</td>
</tr>
</tbody>
</table>

The first observation in the output data set has a **_TYPE_** value of 0, which indicates that the statistics are generated for the entire data set.

*Note:* SAS calculates the **_TYPE_** variable even if no output data set is requested.

By default, the output data set that is generated by PROC MEANS contains a separate observation for each unique combination of values of the class variables for each type. Each unique combination of values within a type is called a level of that type. In the output data set linked above, for example, there are 17 levels for type 2 (17 observations that have a **_TYPE_** value of 2).

The output report that is generated by the basic PROC MEANS step contains only the observations that represent a combination of all of the class variables (the observations for which **_TYPE_**=7). The **_TYPE_** variable is not displayed in the report.
Using the TYPES Statement in PROC MEANS to Combine Class Variables

You can use the TYPES statement in PROC MEANS to specify which combinations of the class variables are used for grouping data and for calculating statistics. The CLASS statement is required in order to use the TYPES statement.

General form, TYPES statement:

```
TYPES request(s);
```

where

- `request(s)` specifies the combination(s) of class variables that PROC MEANS uses to create the types. A request is composed of one of the following:
  - one class variable name
  - several class variable names separated by asterisks
  - `()` to request the combination of all variables (\(_\text{TYPE}_\) = 0).

To request combinations of class variables more concisely, you can use a grouping syntax by placing parentheses around several variables and joining other variables or variable combinations. The following examples of TYPES statements illustrate the use of grouping syntax:

<table>
<thead>
<tr>
<th>Example with Grouping Syntax</th>
<th>Equivalent Example without Grouping Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>types a*(b c);</code></td>
<td><code>types a*b a*c;</code></td>
</tr>
<tr>
<td><code>types (a b)*(c d);</code></td>
<td><code>types a*c a*d b*c b*d;</code></td>
</tr>
<tr>
<td><code>types (a b c)*d;</code></td>
<td><code>types a*d b*d c*d;</code></td>
</tr>
<tr>
<td><code>types () a*(b c);</code></td>
<td><code>types a*b*c a*b a*c;</code></td>
</tr>
</tbody>
</table>

Example: Using the TYPES Statement in PROC MEANS

Suppose you want to calculate average employee salaries, as in the previous example. This time, you want to group results for the two combinations of class variables shown below:

- Employee_Country and Department
- Employee_Country and Employee_Gender.

To do this, you can add a TYPES statement to the PROC MEANS step:

```plaintext
proc means data=company.organization mean;
    class employee_country department employee_gender;
    var salary;
    types employee_country*department
```
```
employee_country*employee_gender;
output out=summary mean=average;
run;
```

This PROC MEANS step generates both an output report and an output data set. The report, shown below, has a separate table for each of the two combinations that were specified in the TYPES statement:

The output data set summarizes and reports data for only the combinations (types) that are specified in the TYPES statement. A partial view of the output data set is shown below:

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee_Country</th>
<th>Department</th>
<th>Employee_Gender</th>
<th>TYPE</th>
<th>FREQ</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AU</td>
<td>F</td>
<td>M</td>
<td></td>
<td>5</td>
<td>$27,961</td>
</tr>
<tr>
<td>2</td>
<td>AU</td>
<td>F</td>
<td>M</td>
<td></td>
<td>60</td>
<td>$33,289</td>
</tr>
<tr>
<td>3</td>
<td>BE</td>
<td>F</td>
<td>M</td>
<td></td>
<td>27</td>
<td>$30,342</td>
</tr>
<tr>
<td>4</td>
<td>BE</td>
<td>M</td>
<td>M</td>
<td></td>
<td>33</td>
<td>$29,754</td>
</tr>
<tr>
<td>5</td>
<td>DE</td>
<td>F</td>
<td>M</td>
<td></td>
<td>45</td>
<td>$34,060</td>
</tr>
<tr>
<td>6</td>
<td>DE</td>
<td>M</td>
<td>M</td>
<td></td>
<td>53</td>
<td>$27,362</td>
</tr>
<tr>
<td>7</td>
<td>DK</td>
<td>F</td>
<td>M</td>
<td></td>
<td>23</td>
<td>$27,657</td>
</tr>
<tr>
<td>8</td>
<td>DK</td>
<td>M</td>
<td>M</td>
<td></td>
<td>28</td>
<td>$34,694</td>
</tr>
<tr>
<td>9</td>
<td>ES</td>
<td>F</td>
<td>M</td>
<td></td>
<td>29</td>
<td>$33,353</td>
</tr>
<tr>
<td>10</td>
<td>ES</td>
<td>M</td>
<td>M</td>
<td></td>
<td>37</td>
<td>$29,323</td>
</tr>
</tbody>
</table>
Using the NWAY Option in PROC MEANS to Combine Class Variables

Another way to specify a combination of class variables is to use the NWAY option in PROC MEANS:

General form, NWAY option in the PROC MEANS statement:

```
PROC MEANS NWAY;
```

where

NWAY

specifies that the output will contain statistics for the combination of all specified class variables (the observations that have the highest _TYPE_ value).

The NWAY option enables you to generate summary statistics for one particular combination of class variables—all of the class variables—in a single PROC MEANS step. Therefore, to generate statistics for several different combinations of class variables, you can specify a separate PROC MEANS step that contains the NWAY option for each combination.

Example: Using the NWAY Option in Multiple PROC MEANS Steps

Suppose you want to calculate average employee salaries and to group results for the following combinations of class variables:

- Employee_Country and Department
- Employee_Country and Employee_Gender.

You can use two PROC MEANS steps, each containing the NWAY option, as shown below. The first PROC MEANS step generates statistics for the first combination of class variables, and the second PROC MEANS steps generates statistics for the second combination of class variables.

```
proc means data=company.organization nway;
  class employee_country department;
  var salary;
  output out=summary1
       n=employees
       mean=average;
run;

proc means data=company.organization nway;
  class employee_country employee_gender;
  var salary;
  output out=summary2
       n=employees
       mean=average;
run;
```

When processing this program, SAS must read the data set once for each PROC MEANS step.

This program generates an output report and two output data sets. The report, shown in part below, has a separate table for each PROC MEANS step:
Each output data set summarizes and reports data for one of the types that are specified in the TYPES statement. A partial view of each output data set is shown below:

**Figure 24.2** SAS Data Set Work.Summary1
Using the WHERE= Option in PROC MEANS to Combine Class Variables

Yet another way to specify a combination of class variables is to use the WHERE= data set option in the OUTPUT statement:

General form, WHERE= data set option in a basic OUTPUT statement:

```
OUTPUT <OUT=SAS-data-set> <output-statistic-specification(s)> (WHERE=
(where-expression-1 <logical-operator where-expression-n>));
```

where

- **SAS-data-set** specifies the new output data set as a 1-level or 2-level name.
- **output-statistic-specification(s)** specifies the statistic(s) to store in the output data set and names one or more variables that contain the statistics.
- **where-expression** is an arithmetic or logical expression that consists of a sequence of operators, operands, and SAS functions. An operand is a variable, a SAS function, or a constant. An operator is a symbol that requests a comparison, logical operation, or arithmetic calculation. The expression must be enclosed in parentheses.
- **logical-operator** can be AND, AND NOT, OR, or OR NOT.

When you use the WHERE= option in the OUTPUT statement, SAS must calculate all possible combinations of class variables, and subsetting does not occur until the results are written to output.

Example: Using the WHERE= Option in PROC MEANS

Suppose you want to calculate average employee salaries and to group results for two 2-way combinations of the three class variables **Employee_Country**, **Department**, and
Employee_Gender. All possible combinations of these variables are listed below, and the two combinations that you want are highlighted:

<table>
<thead>
<tr>
<th><em>TYPE</em> Value</th>
<th>Variables Combined</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Employee_Gender</td>
<td>1-way</td>
</tr>
<tr>
<td>2</td>
<td>Department</td>
<td>1-way</td>
</tr>
<tr>
<td>3</td>
<td>Department * Employee_Gender</td>
<td>2-way</td>
</tr>
<tr>
<td>4</td>
<td>Employee_Country</td>
<td>1-way</td>
</tr>
<tr>
<td>5</td>
<td>Employee_Country * Employee_Gender</td>
<td>2-way</td>
</tr>
<tr>
<td>6</td>
<td>Employee_Country * Department</td>
<td>2-way</td>
</tr>
<tr>
<td>7</td>
<td>Employee_Country * Department * Employee_Gender</td>
<td>3-way</td>
</tr>
</tbody>
</table>

To specify the types by _TYPE_ value, you can use the WHERE= option in the OUTPUT statement as shown below:

```plaintext
proc means data=company.organization;
  class employee_country department employee_gender;
  var salary;
  output out=summary
      (where=( _type_ in (5,6)))
      n=employees
      mean=average;
run;
```

A partial view of the output report is shown below:

<table>
<thead>
<tr>
<th>Employee Country</th>
<th>Department</th>
<th>Employee Gender</th>
<th>N</th>
<th>Obs</th>
<th>N</th>
<th>Mean</th>
<th>Std Dev</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>Administration</td>
<td>Female</td>
<td>8</td>
<td>8</td>
<td>30334.38</td>
<td>6678.40</td>
<td>26469.00</td>
<td>46230.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Engineering</td>
<td>Male</td>
<td>5</td>
<td>5</td>
<td>27644.00</td>
<td>1210.34</td>
<td>26503.00</td>
<td>29250.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sales</td>
<td>Female</td>
<td>36</td>
<td>36</td>
<td>27191.53</td>
<td>1831.47</td>
<td>24015.00</td>
<td>30800.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sales Management</td>
<td>Male</td>
<td>42</td>
<td>42</td>
<td>27607.98</td>
<td>2344.65</td>
<td>24100.00</td>
<td>36005.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stock &amp; Shipping</td>
<td>Female</td>
<td>3</td>
<td>3</td>
<td>30795.00</td>
<td>4441.63</td>
<td>27365.00</td>
<td>59935.00</td>
<td></td>
</tr>
<tr>
<td>Belgium</td>
<td>Administration</td>
<td>Female</td>
<td>6</td>
<td>6</td>
<td>23007.50</td>
<td>1797.21</td>
<td>26105.00</td>
<td>30855.00</td>
<td></td>
</tr>
</tbody>
</table>

A partial view of the output data set Work.Summary is shown below:

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee_Country</th>
<th>Department</th>
<th>Employee_Gender</th>
<th><em>TYPE</em></th>
<th><em>FREQ</em></th>
<th>employees</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Australia</td>
<td>Female</td>
<td>5</td>
<td>48</td>
<td>48</td>
<td>$27,961</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Australia</td>
<td>Male</td>
<td>5</td>
<td>60</td>
<td>60</td>
<td>$33,289</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Belgium</td>
<td>Female</td>
<td>5</td>
<td>27</td>
<td>27</td>
<td>$32,342</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Belgium</td>
<td>Male</td>
<td>5</td>
<td>33</td>
<td>33</td>
<td>$29,754</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Germany</td>
<td>Female</td>
<td>5</td>
<td>45</td>
<td>45</td>
<td>$34,060</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Germany</td>
<td>Male</td>
<td>5</td>
<td>53</td>
<td>53</td>
<td>$27,362</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Denmark</td>
<td>Female</td>
<td>5</td>
<td>23</td>
<td>23</td>
<td>$27,657</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Denmark</td>
<td>Male</td>
<td>5</td>
<td>28</td>
<td>28</td>
<td>$34,694</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Spain</td>
<td>Female</td>
<td>5</td>
<td>29</td>
<td>29</td>
<td>$33,353</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Spain</td>
<td>Male</td>
<td>5</td>
<td>37</td>
<td>37</td>
<td>$29,323</td>
<td></td>
</tr>
</tbody>
</table>
Next, let’s compare the resources that are used by these summarization techniques:

- the TYPES statement in PROC MEANS
- the NWAY option in multiple PROC MEANS steps
- the WHERE= option in PROC MEANS.

### Comparative Example: Displaying Summary Statistics for Combinations of Class Variables

Suppose you want to summarize the data set _Retail.Organization_ by calculating average employee salaries for two 3-way combinations of four class variables:

- _Employee_Country_, _Department_, and _Employee_Gender_
- _Department_, _Section_, and _Employee_Gender_.

You can use the following techniques to produce an output report and one or more output data sets:

1. TYPES Statement in PROC MEANS
2. NWAY Option in Two PROC MEANS Steps
3. WHERE= Option in PROC MEANS

The following programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for summarizing data for combinations of class variables.

### Programming Techniques

1. **TYPES Statement in PROC MEANS**

   This program calculates the average employee salary for two 3-way combinations of the class variables _Employee_Country_, _Department_, _Employee_Gender_, and _Section_. The TYPES statement specifies the two combinations. The program generates an output report and an output data set named _Summary_.

   ```
   proc means data=retail.organization mean;
      class employee_country department employee_gender section;
      var salary;
      types employee_country*department*employee_gender department*section*employee_gender;
      output out=summary n=employees mean=average;
   run;
   ```
NWAY Option in Two PROC MEANS Steps

Each of the two PROC MEANS steps in this program calculates the average employee salary for a combination of three of the four class variables Employee_Country, Department, Employee_Gender, and Section. In each step, the NWAY option specifies that all three variables that are specified in the CLASS statement should be combined. The program generates an output report and two output data sets named Summary1 and Summary2.

```sas
proc means data=retail.organization nway;
  class employee_country department employee_gender;
  var salary;
  output out=summary1
    n=employees
    mean=average;
run;

proc means data=retail.organization nway;
  class department section employee_gender;
  var salary;
  output out=summary2
    n=employees
    mean=average;
run;
```

WHERE= Option in PROC MEANS

This program calculates the average employee salary for two 3-way combinations of the class variables Employee_Country, Department, Employee_Gender, and Section. The WHERE= data set option in the OUTPUT statement specifies the two combinations by their _TYPE_ values. The program generates an output report and an output data set named Summary3.

```sas
proc means data=retail.organization;
  class employee_country department employee_gender section;
  var salary;
  output out=summary3 (where=(_type_ in (7,14)));
    n=employees
    mean=average;
run;
```

General Recommendations

- To summarize data for particular combinations of class variables, use the TYPES statement in PROC MEANS.

Additional Features

The WAYS statement in PROC MEANS provides yet another way to display summary statistics for combinations of class variables. In the WAYS statement, you specify one or more integers that define the number of class variables to combine in order to form all the unique combinations of class variables.

For example, the following program uses the WAYS statement to create summary statistics for the following combinations of the three class variables Employee_Country, Department, and Employee_Gender:
- each individual variable (all 1-way combinations)
- all 2-way combinations (Employee_Country and Department, Employee_Country and Employee_Gender, and Employee_Gender and Department).

```sas
proc means data=company.organization mean;
  class employee_country department employee_gender;
  var salary;
  ways 1 2;
  output out=summary
    mean=average;
run;
```

The WAYS statement can be used instead of or in addition to the TYPES statement.

*Note:* For more information about the WAYS statement, see the SAS documentation.
Chapter 24

Summary

Using an Index for Efficient WHERE Processing

When processing a WHERE expression, SAS determines whether it is more efficient to access observations in a data set sequentially, by searching through all observations, or directly, by using an index to access specific observations. Using an index to process a WHERE expression might improve performance and is referred to as optimizing the WHERE expression. By deciding whether to create an index, you play a role in determining which access method SAS can use.

In order to decide whether to use an index, you must evaluate the benefits and costs of using an index.

SAS performs a series of steps to determine whether to process a WHERE expression by using an index or by reading all the observations in the data file sequentially.

Identifying Available Indexes

First, SAS determines whether there are any existing indexes that might be used to process the WHERE expression. Specifically, SAS checks the variable in each condition in the WHERE expression to determine whether the variable is a key variable in either a simple index or a composite index. No matter how many indexes are available, SAS can use only one index to process a WHERE expression. So, if multiple indexes are available, SAS must choose between them.

It is most common for SAS to use an index to process just one condition in a WHERE expression. However, in a process called compound optimization, SAS can use a composite index to optimize multiple conditions on multiple variables, which are joined with a logical operator such as AND.

Identifying Conditions That Can Be Optimized

Second, SAS looks for operators and functions that can be optimized in the WHERE conditions that contain key variables. There are also certain operators and functions that cannot be optimized. For compound optimization, WHERE conditions must meet slightly different criteria in order to be candidates for optimization.

Estimating the Number of Observations

Third, SAS estimates how many observations will be qualified by the index. When multiple indexes exist, SAS selects the one that appears to produce the fewest qualified observations (the smallest subset). Whether or not SAS uses an index depends on the percentage of observations that are qualified (the size of the subset relative to the size of the data set). It is more efficient to use indexed access for a small subset and sequential access for a large subset. If SAS estimates that the number of qualified observations is less than 3% of the data file, SAS automatically uses the index and does not go on to compare probable resource usage.

To help SAS estimate how many observations would be selected by a WHERE expression, each index stores 21 statistics called cumulative percentiles, or centiles. Centiles provide information about the distribution of values for the indexed variable.

Comparing Probable Resource Usage

Fourth, SAS decides whether it is faster (cheaper) to satisfy the WHERE expression by using the index or by reading all of the observations sequentially. To make the decision, SAS predicts how many I/O operations will be required in order to satisfy the
WHERE expression for each of the access methods, and then compares the two resource costs.

Several factors affect the number of I/O operations that are required for WHERE processing, including the following:

- subset size relative to data set size
- number of pages in the data file
- order of the data
- cost to uncompress a compressed file for a sequential read.

Data type and length are two other factors that affect index efficiency.

**Deciding Whether to Create an Index**

When you use a WHERE expression to select a subset, you can use specific guidelines to decide whether it is efficient to create an index. Depending on factors such as the size of the subset relative to the size of the data set, you might or might not choose to create an index.

In most situations, it is best to let SAS determine whether or not to use an index for WHERE processing. However, sometimes you might want to control whether or not SAS uses an existing index. You can use either of the data set options IDXWHERE= or IDXNAME=, but not both at the same time, to control index usage. You can specify MSGLEVEL=| to tell SAS to display information about index usage in the SAS log.

**Comparing Procedures That Produce Detail Reports**

When you want to use a query to produce a detail report, you can choose between the PRINT procedure and the SQL procedure. To perform a particular task, a single-purpose tool like PROC PRINT generally uses fewer computer resources than a multi-purpose tool like PROC SQL. However, PROC SQL often requires fewer and shorter statements to achieve the results that you want.

For detail reports, a PROC PRINT step often, but not always, uses fewer resources than a PROC SQL step:

- PROC PRINT is usually more efficient than PROC SQL for generating a simple detail report, a subset detail report, and a sorted detail report.
- PRINT PRINT and PROC SQL will likely have similar resource usage for generating a sorted subset detail report.

**Comparing Tools for Summarizing Data**

SAS provides a variety of tools for summarizing data, including the MEANS procedure (or SUMMARY procedure), the TABULATE procedure, the REPORT procedure, the SQL procedure, and the DATA step.

If you are summarizing data for one class variable, the tools in each of the following groups are similar in resource usage:

- PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE
- PROC SQL and the DATA step.

However, the relative efficiency of the two groups of tools varies according to the shape of the data.

You can use PROC MEANS in a variety of ways to produce summary statistics for combinations of class variables. Each combination of class variables is called a type.
To summarize data for all combinations of class variables, you can use a basic PROC MEANS step (or PROC SUMMARY step). To produce summary statistics for specific combinations of class variables, you can use PROC MEANS in the following ways:

- the TYPES statement in a PROC MEANS step
- the NWAY option in multiple PROC MEANS steps
- the WHERE= option in a PROC MEANS step.

These three techniques vary in efficiency; the TYPES statement in PROC MEANS is the most efficient.

You can also use the WAYS statement in PROC MEANS to produce summary statistics for specific combinations of class variables.

Review the related comparative examples:
- “Comparative Example: Displaying Summary Statistics for One Class Variable” on page 858
- “Comparative Example: Displaying Summary Statistics for Combinations of Class Variables” on page 871.

**Quiz**

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Why can using an index reduce the number of I/O operations that are required for accessing a small subset?
   - a. Using an index requires larger input buffers, which can hold more pages.
   - b. The index does not have to be loaded into an input buffer.
   - c. The number of observations that SAS has to load into the program data vector (PDV) is decreased.
   - d. The number of pages that SAS has to load into input buffers is decreased.

2. You want to select a subset of observations in the data set *Company.Products*, and you have defined a simple index on the variable *Rating*. SAS cannot use the index to process which of the following WHERE statements?
   - a. where rating is missing;
   - b. where rating=int(rating);
   - c. where rating between 3.5 and 7.5;
   - d. where rating=5.5;

3. In which of the following situations is sequential access likely to be more efficient than direct access for WHERE processing?
   - a. The subset contains over 75% of the observations in the data set.
   - b. The WHERE expression specifies both key variables in a single composite index.
   - c. The data is sorted on the key variable.
   - d. The data set is very large.

4. You want to summarize data for one class variable, and you are trying to decide whether to use PROC MEANS (or PROC SUMMARY), PROC REPORT, PROC TABULATE, PROC SQL, or the DATA step with PROC SORT. Which of the following statements about the efficiency of these summarization tools is not true?
   - a. PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE have similar resource usage.
b The efficiency of all these tools is affected by the shape of the data.
c The SQL procedure is always the least efficient because it is a
general-purpose tool.
d PROC SQL and the DATA step with PROC SORT have similar resource
usage.

5 Which of the following techniques is most efficient for summarizing data for
specific combinations of class variables?

a the NWAY option in multiple PROC MEANS steps
b the TYPES statement in a PROC MEANS step
c the WHERE= option in a PROC MEANS step
d a basic PROC MEANS step
Quiz Answer Keys

Appendix 1. . . . . . . . . Quiz Answer Keys  881
Chapter 1: Performing Queries Using PROC SQL

1 Which of the clauses in the PROC SQL program below is written incorrectly?

```sql
proc sql;
   select style sqfeet bedrooms
       from choice.houses
       where sqfeet ge 800;
```

a SELECT
b FROM
c WHERE
d both a and c

Correct answer: a
The SELECT clause in the program is written incorrectly. Columns that are listed in the clause must be separated by commas, not just blanks.

2 How many statements does the program below contain?

```sql
proc sql;
  select grapes, oranges,
       grapes + oranges as sumsales
  from sales.produce
  order by sumsales;
```

a two  
b three  
c four  
d five

Correct answer: a

There are two statements, the PROC SQL statement and the SELECT statement. The SELECT statement contains three clauses.

3 Complete the following PROC SQL query to select the columns Address and SqFeet from the table List.Size and to select Price from the table List.Price. (Only the Address column appears in both tables.)

```sql
proc sql;
  select address, sqfeet, price
  from list.size, list.price;
```

a select address, sqfeet, price  
b select size.address, sqfeet, price  
c select price.address, sqfeet, price  
d either b or c

Correct answer: b

The SELECT clause lists the columns from both tables to be queried. You must use a prefix with the Address column because it appears in both tables. The prefix specifies the table from which you want the column to be read.

4 Which of the clauses below correctly sorts rows by the values of the columns Price and SqFeet?

a order price, sqfeet  
b order by price, sqfeet  
c sort by price sqfeet  
d sort price sqfeet

Correct answer: b

The ORDER BY clause specifies how the rows are to be sorted. You follow the keywords ORDER BY by one or more column names or numbers, separated by commas.

5 Which clause below specifies that the two tables Produce and Hardware be queried? Both tables are located in a library to which the libref Sales has been assigned.

a select sales.produce sales.hardware  
b from sales.produce sales.hardware  
c from sales.produce, sales.hardware
d where sales.produce, sales.hardware

Correct answer: c

In the FROM clause, you list the names of the tables to be queried, separated by commas.

6 Complete the SELECT clause below to create a new column named **Profit** by subtracting the values of the column **Cost** from those of the column **Price**.

```sql
select fruit,cost,price, 
________________
```

a Profit=price-cost  
b price-cost as Profit  
c profit=price-cost  
d Profit as price-cost

Correct answer: b

To create a new column and assign a column alias to the column, you specify the following in the SELECT clause, in the order shown here: an expression, (optionally) the keyword AS, and a column alias. The case that you use when you create the column name is the one that will be displayed in the output.

7 What happens if you use a GROUP BY clause in a PROC SQL step without a summary function?

a The step does not execute.  
b The first numeric column is summed by default.  
c The GROUP BY clause is changed to an ORDER BY clause.  
d The step executes but does not group or sort data.

Correct answer: c

The GROUP BY clause is used in queries that include one or more summary functions. If you specify a GROUP BY clause in a query that does not contain a summary function, your clause is changed to an ORDER BY clause.

8 If you specify a CREATE TABLE statement in your PROC SQL step,

a the results of the query are displayed, and a new table is created.  
b a new table is created, but it does not contain any summarization that was specified in the PROC SQL step.  
c a new table is created, but no report is displayed.  
d results are grouped by the value of the summarized column.

Correct answer: c

The CREATE TABLE statement enables you to store your results in a SAS table instead of displaying the query results as a report.

9 Which statement is true regarding the use of the PROC SQL step to query data that is stored in two or more tables?

a When you join multiple tables, the tables must contain a common column.  
b You must specify the table from which you want each column to be read.  
c The tables that are being joined must be from the same type of data source.  
d If two tables that are being joined contain a same-named column, then you must specify the table from which you want the column to be read.

Correct answer: d

If you are joining two tables that contain a same-named column, then you must use a prefix to specify the table(s) from which you want the column to be read. Remember that if you join tables that don't contain columns that have matching
data values, you can produce a huge amount of output. Be sure to specify a WHERE clause to select only the rows that you want.

10 Which clause in the following program is incorrect?

```sql
proc sql;
  select sex, mean(weight) as avgweight
  from company.employees company.health
  where employees.id=health.id
  group by sex;
```

a SELECT  
b FROM  
c WHERE  
d GROUP BY

Correct answer: b

The table names that are specified in the FROM clause must be separated by commas. Note that you can specify columns in the WHERE clause that are not specified in the SELECT clause.

---

Chapter 2: Performing Advanced Queries Using PROC SQL

1 Which PROC SQL query will remove duplicate values of MemberType from the query output, so that only the unique values are listed?

a proc sql nodup;
  select membertype
  from sasuser.frequentflyers;

b proc sql;
  select distinct(membertype)
    as MemberType
  from sasuser.frequentflyers;

c proc sql;
  select unique membertype
  from sasuser.frequentflyers
  group by membertype;

d proc sql;
  select distinct membertype
  from sasuser.frequentflyers;

Correct answer: d

To remove duplicate values from PROC SQL output, you specify the DISTINCT keyword before the column name in the SELECT clause.

2 Which of the following will cause PROC SQL to list rows that have no data in the Address column?

a WHERE address is missing  
b WHERE address not exists  
c WHERE address is null  
d both a and c

Correct answer: d
To list rows that have no data (that is, missing data), you can use either of these other conditional operators: IS MISSING or IS NULL. The NOT EXISTS operator is used specifically with a subquery, and resolves to true if the subquery returns no values to the outer query.

3 You are creating a PROC SQL query that will list all employees who have spent (or overspent) their allotted 120 hours of vacation for the current year. The hours that each employee used are stored in the existing column `Spent`. Your query defines a new column, `Balance`, to calculate each employee’s balance of vacation hours.

Which query will produce the report that you want?

a proc sql;
   select name, spent,
       120-spent as calculated Balance
   from Company.Absences
   where balance <= 0;

b proc sql;
   select name, spent,
       120-spent as Balance
   from Company.Absences
   where calculated balance <= 0;

c proc sql;
   select name, spent,
       120-spent as Balance
   from Company.Absences
   where balance <= 0;

d proc sql;
   select name, spent,
       120-spent as calculated Balance
   from Company.Absences
   where calculated balance <= 0;

Correct answer: b

When a WHERE clause references a new column that was defined in the SELECT clause, the WHERE clause must specify the keyword CALCULATED before the column name.

4 Consider this PROC SQL query:

proc sql;
   select flightnumber,
       count(*) as Flights,
       avg(boarded)
       label="Average Boarded"
       format=3.
   from sasuser.internationalflights
   group by flightnumber
   having avg(boarded) > 150;

The table `Sasuser.InternationalFlights` contains 201 rows, 7 unique values of `FlightNumber`, 115 unique values of `Boarded`, and 4 different flight numbers that have an average value of `Boarded` that is greater than 150. How many rows of output will the query generate?

a 150
b 7
c 4
Correct answer: c

To determine how PROC SQL calculates and displays output from summary functions, consider the key factors. This PROC SQL query has a GROUP BY clause, and it does not specify any columns that are outside of summary functions. Therefore, PROC SQL calculates and displays the summary function for each group. There are 7 unique values of FlightNumber, but the HAVING clause specifies only the flights that have an average number of boarded passengers greater than 150. Because 4 of the 7 flight numbers meet this condition, the output will contain 4 rows.

5 You are writing a PROC SQL query that will display the names of all library cardholders who work as volunteers for the library, and the number of books that each volunteer currently has checked out. You will use one or both of the following tables:

- Library.Circulation lists the name and contact information for all library cardholders, and the number of books that each cardholder currently has checked out.
- Library.Volunteers lists the name and contact information for all library volunteers.

Assume that the values of Name are unique in both tables.

Which of the following PROC SQL queries will produce your report?

a proc sql;
    select name, checkedout
    from library.circulation
    where * in
        (select *
         from library.volunteers);

b proc sql;
    select name, checkedout
    from library.circulation
    where name in
        (select name
         from library.volunteers);

c proc sql;
    select name
    from library.volunteers
    where name, checkedout in
        (select name, checkedout
         from library.circulation);

d proc sql;
    select name, checkedout
    from library.circulation
    where name in
        (select name
         from library.volunteers);

Correct answer: b

Your PROC SQL query needs to use data from both tables. The outer query reads the name and number of books checked out from Library.Circulation. The multiple-value noncorrelated subquery selects the names of volunteers from Library.Volunteers and passes these names back to the outer query. The outer
query then selects data for only the volunteers whose names match names returned by the subquery. The subquery is indented under the outer query’s WHERE clause, is enclosed in parentheses, and does not require a semicolon inside the closing parenthesis.

6 By definition, a noncorrelated subquery is a nested query that
   a. returns a single value to the outer query.
   b. contains at least one summary function.
   c. executes independently of the outer query.
   d. requires only a single value to be passed to it by the outer query.

Correct answer: c

A noncorrelated subquery is a nested query that executes independently of the outer query. The outer query passes no values to the subquery.

7 Which statement about the following PROC SQL query is false?

```sql
proc sql;
   validate
   select name label='Country',
       rate label='Literacy Rate'
   from world.literacy
   where 'Asia' =
      (select continent
       from world.continents
       where literacy.name =
           continents.country)
   order by 2;
```

   a. The query syntax is not valid.
   b. The outer query must pass values to the subquery before the subquery can return values to the outer query.
   c. PROC SQL will not execute this query when it is submitted.
   d. After the query is submitted, the SAS log will indicate whether the query has valid syntax.

Correct answer: a

The syntax in this PROC SQL query is valid, so the first statement is false. The query contains a correlated subquery, so the second statement is true. The VALIDATE keyword is used after the PROC SQL statement, so the third statement is true. And the last statement correctly indicates that the VALIDATE keyword causes the SAS log to display a special message if the query syntax is valid, or standard error messages if the syntax is not valid.

8 Consider the following PROC SQL query:

```sql
proc sql;
   select lastname, firstname, 
       total, since 
   from charity.donors 
   where not exists 
      (select lastname 
       from charity.current 
       where donors.lastname =
           current.lastname);
```

The query references two tables:

- `Charity.Donors` lists name and contact information for all donors who have made contributions since the charity was founded. The table also contains
these two columns: **Total**, which shows the total dollars given by each donor, and **Since**, which stores the first year in which each donor gave money.

- **Charity.Current** lists the names of all donors who have made contributions in the current year, and the total dollars each has given this year (**YearTotal**).

Assume that the values of **LastName** are unique in both tables.

The output of this query displays

a. all donors whose rows do not contain any missing values.
b. all donors who made a contribution in the current year.
c. all donors who did not make a contribution in the current year.
d. all donors whose current year’s donation in **Charity.Current** has not yet been added to **Total** in **Charity.Donors**.

Correct answer: c

In this PROC SQL query, the outer query uses the operator NOT EXISTS with a correlated subquery. The outer query selects all rows from **Charity.Donors** whose names do not appear in **Charity.Current**. In other words, this PROC SQL query output lists all donors who did not make a contribution in the current year.

9 Which statement about data remerging is true?

a. When PROC SQL remerges data, it combines data from two tables.
b. By using data remerging, PROC SQL can avoid making two passes through the data.
c. When PROC SQL remerges data, it displays a related message in the SAS log.
d. PROC SQL does not attempt to remerge data unless a subquery is used.

Correct answer: c

The third statement about data remerging is correct.

10 A public library has several categories of books. Each book in the library is assigned to only one category. The table **Library.Inventory** contains one row for each book in the library. The **Checkouts** column indicates the number of times that each book has been checked out.

You want to display only the categories that have an average circulation (number of checkouts) that is less than 2500. Does the following PROC SQL query produce the results that you want?

```sql
proc sql;
title 'Categories with Average Circulation';
title2 'Less Than 2500';
select category,
    avg(checkouts) as AvgCheckouts
from library.inventory
having avg(checkouts) < 2500
order by 1;
```

a. No. This query will not run because a HAVING clause cannot contain a summary function.
b. No. This query will not run because the HAVING clause must include the CALCULATED keyword before the summary function.
c. No. Because there is no GROUP BY clause, the HAVING clause treats the entire table as one group.
d. Yes.

Correct answer: c

PROC SQL can execute this query, but the query will not produce the results that you want. If you omit the GROUP BY clause in a query that contains a HAVING clause, then the HAVING clause and any summary functions treat the
entire table as one group. Without a GROUP BY clause, the HAVING clause in this example calculates the average circulation for the table as a whole (all books in the library), not for each group (each category of books). The output contains either all the rows in the table (if the average circulation for the entire table is less than 2500) or none of the rows in the table (if the average circulation for the entire table is greater than 2500).

Chapter 3: Combining Tables Horizontally Using PROC SQL

1. A Cartesian product is returned when
   a. join conditions are not specified in a PROC SQL join.
   b. join conditions are not specified in a PROC SQL set operation.
   c. more than two tables are specified in a PROC SQL join.
   d. the keyword ALL is used with the OUTER UNION operator.
   Correct answer: a
   A Cartesian product is returned when join conditions are not specified in a PROC SQL join. In a Cartesian product, each row from the first table is combined with every row from the second table.

2. Given the PROC SQL query and tables shown below, which output is generated?

   ```sql
   proc sql;
   select *
     from store1,
     store2
   where store1.wk=
     store2.wk;
   ```

<table>
<thead>
<tr>
<th>Wk</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$515.07</td>
</tr>
<tr>
<td>2</td>
<td>$772.29</td>
</tr>
<tr>
<td>3</td>
<td>$888.88</td>
</tr>
<tr>
<td>4</td>
<td>$1000.01</td>
</tr>
<tr>
<td>5</td>
<td>$43.00</td>
</tr>
<tr>
<td>Wk</td>
<td>Sales</td>
</tr>
<tr>
<td>----</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>$1368.99</td>
</tr>
<tr>
<td>2</td>
<td>$1506.23</td>
</tr>
<tr>
<td>3</td>
<td>$1200.57</td>
</tr>
<tr>
<td>4</td>
<td>$1784.11</td>
</tr>
<tr>
<td>5</td>
<td>$43.00</td>
</tr>
</tbody>
</table>

   a
Correct answer: b

This PROC SQL query is an inner join. It combines the rows from the first table that match rows from the second table, based on the matching criteria specified in the WHERE clause. Columns are not overlaid, so all columns from the referenced tables (including any columns with duplicate names) are displayed. Any unmatched rows from either table are not displayed.

3 Given the PROC SQL query and tables shown below, which output is generated?

```
proc sql;
  select s.*, bonus
  from bonus as b
  right join
  salary as s
  on b.id=
  s.id;
```

<table>
<thead>
<tr>
<th>Wk</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$515.07</td>
</tr>
<tr>
<td>2</td>
<td>$772.29</td>
</tr>
<tr>
<td>3</td>
<td>$888.88</td>
</tr>
<tr>
<td>4</td>
<td>$1000.01</td>
</tr>
<tr>
<td>1</td>
<td>$1368.99</td>
</tr>
<tr>
<td>2</td>
<td>$1506.23</td>
</tr>
<tr>
<td>3</td>
<td>$1200.57</td>
</tr>
<tr>
<td>4</td>
<td>$1784.11</td>
</tr>
</tbody>
</table>
4. Which PROC SQL query produces the same output as the query shown here?

```
proc sql;
  select a.*,
  duration
  from groupa as a,
  groupb as b
  where a.obs=b.obs;
```

**Note:** Assume that the table `Groupa` contains the columns `Obs` and `Med`. `Groupb` contains the columns `Obs` and `Duration`.

a. `proc sql;
  select a.obs label='Obs',`
med
b.obs label='Obs',
duration
from groupa as a, groupb as b
where a.obs=b.obs;

b proc sql;
select coalesce(a.obs, b.obs)
label='Obs', med, duration
from groupa as a
full join
groupb as b
on a.obs=b.obs;

c proc sql;
select a.*, duration
from groupa as a
left join
groupb as b
where a.obs=b.obs;

d proc sql;
select a.*, duration
from groupa as a
inner join
groupb as b
on a.obs=b.obs;

Correct answer: d

There are two valid formats for writing a PROC SQL inner join. The PROC SQL query shown at the top of this question uses the first inner join format, which does not use a keyword to indicate the type of join. The alternate format is similar to an outer join and uses the keyword INNER JOIN.

5 Which output will the following PROC SQL query generate?

proc sql;
select *
from table1
left join
table2
on table1.g3=
table2.g3;

a

<table>
<thead>
<tr>
<th>G3 Z</th>
<th>G3 R</th>
</tr>
</thead>
<tbody>
<tr>
<td>89 FL</td>
<td>.</td>
</tr>
<tr>
<td>46 UI</td>
<td>46 BC</td>
</tr>
<tr>
<td>47 BA</td>
<td>.</td>
</tr>
</tbody>
</table>
b

Correct answer: a

This PROC SQL query is a left outer join, which retrieves all rows that match across tables (based on the join conditions in the ON clause), plus nonmatching rows from the left (first) table. No columns are overlaid, so all columns from both tables are displayed.

6 In order for PROC SQL to perform an inner join,
   a the tables being joined must contain the same number of columns.
   b the tables must be sorted before they are joined.
   c the columns that are specified in a join condition in the WHERE clause must have the same data type.
   d the columns that are specified in a join condition in the WHERE clause must have the same name.

Correct answer: c

Inner joins combine the rows from the first table that match rows from the second table, based on one or more join conditions in the WHERE clause. The columns being matched must have the same data type, but they are not required to have the same name. For joins, the tables being joined can have different numbers of columns, and the rows do not need to be sorted.

7 Which statement about in-line views is false?
   a Once defined, an in-line view can be referenced in any PROC SQL query in the current SAS session.
   b An in-line view can be assigned a table alias but not a permanent name.
   c In-line views can be combined with tables in PROC SQL joins and set operations.
   d This PROC SQL query contains an in-line view that uses valid syntax:

\[
\begin{align*}
\text{proc sql;} \\
& \quad \text{select name, numvisits} \\
& \quad \quad \text{from (select name, sum(checkin) as numvisits} \\
& \quad \quad \quad \text{from facility as f, members as m}
\end{align*}
\]
where area='POOL' and
  f.id=m.id
group by name)
where numvisits<=10
order by 1;

Correct answer: a

Unlike a table, an in-line view exists only during query execution. Because it is
temporary, an in-line view can be referenced only in the query in which it is
defined.

8 Which PROC SQL query will generate the same output as the DATA step
match-merge and PRINT step shown below?

```
data merged;
  merge table1 table2;
  by g3;
run;

proc print data=merged
noobs;
  title 'Merged';
run;
```

a proc sql;
title 'Merged';
  select a.g3, z, r
  from table1 as a
  full join
  table2 as b
  on a.g3 = b.g3
  order by 1;

b proc sql;
title 'Merged';
  select a.g3, z, r
  from table1 as a
  full join corr
  table2 as b
  on a.g3 = b.g3
  order by 1;

c proc sql;
title 'Merged';
  select coalesce(a.g3, b.g3)
    label='G3', z, r
  from table1 as a
  full join
  table2 as b
  on a.g3 = b.g3
  order by 1;

d proc sql;
title 'Merged';
  select g3, z, r
  from table1 as a
full join
  table2 as b
  on a.g3 = b.g3
  order by 1;

Correct answer: c

In order to generate the same output as the DATA step and PRINT steps, the PROC SQL full outer join must use the COALESCE function with the duplicate columns specified as arguments.

9 A PROC SQL inner join can combine

- a a maximum of 2 tables or in-line views, but multiple joins can be chained together.
- b a maximum of 32 tables or 2 in-line views.
- c a maximum of 32 tables, which includes any tables referenced by an in-line view.
- d a maximum of 2 tables and 32 columns.

Correct answer: c

A maximum of 32 tables can be combined in a single inner join. If the join involves views (either in-line views or PROC SQL views), it is the number of tables that underlie the views, not the number of views, that counts towards the limit of 32.

10 Which statement about the use of table aliases is false?

- a Table aliases must be used when referencing identical table names from different libraries.
- b Table aliases can be referenced by using the keyword AS.
- c Table aliases (or full table names) must be used when referencing a column name that is the same in two or more tables.
- d Table aliases must be used when using summary functions.

Correct answer: d

The use of summary functions does not require the use of table aliases. All of the other statements about table aliases that are shown here are true.

---

Chapter 4: Combining Tables Vertically Using PROC SQL

1 Which statement is false with respect to a set operation that uses the EXCEPT, UNION, or INTERSECT set operator without a keyword?

- a Column names in the result set are determined by the first table.
- b To be overlaid, columns must be of the same data type.
- c To be overlaid, columns must have the same name.
- d By default, only unique rows are displayed in the result set.

Correct answer: c

In set operations that use the operator EXCEPT, INTERSECT, or UNION, and no keyword, columns are overlaid based on their position in the SELECT clause. It does not matter whether the overlaid columns have the same name. When columns are overlaid, the column name is taken from the first table that is specified in the SELECT clause.

2 The keyword ALL cannot be used with which of the following set operators?

- a EXCEPT
b INTERSECT

c UNION

d OUTER UNION

Correct answer: d

By default, when processing a set operation that contains the EXCEPT, INTERSECT, and UNION set operators, PROC SQL makes an extra pass through the data to eliminate duplicate rows. The keyword ALL is used to suppress that additional pass through the tables, allowing duplicate rows to appear in the result set. Because the OUTER UNION set operator displays all rows, the keyword ALL is invalid and cannot be used with OUTER UNION.

3 Which PROC SQL step combines the tables Summer and Winter to produce the output displayed below?

<table>
<thead>
<tr>
<th>Month</th>
<th>Temp</th>
<th>Precip</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>78</td>
<td>.05</td>
</tr>
<tr>
<td>8</td>
<td>85</td>
<td>.04</td>
</tr>
<tr>
<td>9</td>
<td>83</td>
<td>.15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Month</th>
<th>Temp</th>
<th>Precip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
<td>.15</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>.17</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>.20</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>.17</td>
</tr>
</tbody>
</table>

```
\begin{tabular}{|c|c|c|}
\hline
Month & Temp & Precip \\
\hline
7 & 78 & .05 \\
8 & 85 & .04 \\
9 & 83 & .15 \\
\hline
\end{tabular}
```

a proc sql;
  select *
    from summer
    intersect all
  select *
    from winter;

b proc sql;
  select *
    from summer
    outer union
  select *
    from winter;

c proc sql;
  select *
    from summer
    union corr
  select *
    from winter;

d proc sql;
  select *
    from summer
union
select *
from winter;

Correct answer: d

The output shown above contains all rows that are unique in the combined set of rows from both tables, and the columns have been overlaid by position. This output is generated by a set operation that uses the set operator UNION without keywords.

4 Which PROC SQL step combines tables but does not overlay any columns?

a proc sql;
select *
from groupa
outer union
select *
from groupb;

b proc sql;
select *
from groupa as a
outer union corre
select *
from groupb as b;

c proc sql;
select coalesce(a.obs, b.obs)
  label='Obs', med, duration
from groupa as a
full join
groupb as b
  on a.obs=b.obs;

d proc sql;
select *
from groupa as a
intersect
select *
from groupb as b;

Correct answer: a

The PROC SQL set operation that uses the set operator OUTER UNION without a keyword is the only code shown that does not overlay any columns in output.

5 Which statement is false regarding the keyword CORRESPONDING?

a It cannot be used with the keyword ALL.
b It overlays columns by name, not by position.
c When used in EXCEPT, INTERSECT, and UNION set operations, it removes any columns not found in both tables.
d When used in OUTER UNION set operations, it causes same-named columns to be overlaid.

Correct answer: a

The keyword CORRESPONDING (CORR) can be used alone or together with the keyword ALL.

6 Which PROC SQL step generates the following output from the tables Dogs and Pets?
a proc sql;
  select name, price
  from pets
  except all
  select *
  from dogs;

b proc sql;
  select name, price
  from pets
  except
  select *
  from dogs;

c proc sql;
  select name, price
  from pets
  except corr all
  select *
  from dogs;

d proc sql;
  select *
  from dogs
  except corr
  select name, price
  from pets;

Correct answer: b

This PROC SQL output includes all rows from the table *Pets* that do not appear in the table *Dogs*. No duplicates are displayed. A PROC SQL set operation that contains the set operator EXCEPT without keywords produces these results.

7 The *PROG1* and *PROG2* tables list students who took the PROG1 and PROG2 courses, respectively. Which PROC SQL step will give you the names of the students who took only the PROG1 class?
a proc sql;
   select fname, lname
   from prog1
   intersect
   select fname, lname
   from prog2;

b proc sql;
   select fname, lname
   from prog1
   except all
   select fname, lname
   from prog2;

c proc sql;
   select *
   from prog2
   intersect corr
   select *
   from prog1;

d proc sql;
   select *
   from prog2
   union
   select *
   from prog1;

Correct answer: b

The set operator EXCEPT returns all the rows in the first table that do not appear in the second table. The keyword ALL suppresses the extra pass that PROC SQL makes through the data to eliminate duplicate rows. The EXCEPT operator when used alone will also produce the output specified in the question.

8 Which PROC SQL step will return the names of all the students who took PROG1, PROG2, or both classes?
The set operator **UNION** returns all rows that are *unique* in the combined set of rows from both tables.

**9** Which PROC SQL step will return the names of all the students who took both the PROG1 and PROG2 classes?
Quiz Answer Keys  Chapter 4: Combining Tables Vertically Using PROC SQL

a proc sql;
   select fname, lname
   from progl
   union
   select fname, lname
   from prog2;

b proc sql;
   select fname, lname
   from progl
   except corr
   select fname, lname
   from prog2;

c proc sql;
   select fname, lname
   from progl
   intersect all
   select fname, lname
   from prog2;

d proc sql;
   select fname, lname
   from progl
   union corr
   select fname, lname
   from prog2;

Correct answer: c

The set operator INTERSECT returns all rows that are common to both tables. Specifying the keyword ALL suppresses PROC SQL’s additional pass through the data to eliminate duplicate rows.
10 Which PROC SQL step will generate the same results as the following DATA step?

```sql
proc print noobs;
run;
```

a proc sql;
  select fname, lname
  from prog1
  outer union corr
  select fname, lname
  from prog2
  order by lname;

b proc sql;
  select fname, lname
  from prog1
  union
  select fname, lname
  from prog2
  order by lname;

c proc sql;
  select fname, lname
  from prog2
  outer union
  select fname, lname
  from prog1
  order by lname;

d proc sql;
  select fname, lname
  from prog2
  union corr
  select fname, lname
  from prog1
  order by lname;

Correct answer: a

The DATA step returns all rows from the first table along with all rows from the second table, maintaining the order specified in the BY statement. Same-named columns are overlaid by default. The set operator OUTER UNION returns all rows from both tables. The CORR keyword causes same-named columns to be overlaid. The ORDER BY clause causes the result rows to be ordered by values of the specified column (LName).
Chapter 5: Creating and Managing Tables Using PROC SQL

1 Which of the following PROC SQL steps creates a new table by copying only the column structure (but not the rows) of an existing table?
   a proc sql;
      create table work.newpayroll as
      select *
      from sasuser.payrollmaster;
   b proc sql;
      create table work.newpayroll
      like sasuser.payrollmaster;
   c proc sql;
      create table work.newpayroll
      copy sasuser.payrollmaster;
   d proc sql;
      create table work.newpayroll
      describe sasuser.payrollmaster;

Correct answer: b

The CREATE TABLE statement that includes a LIKE clause copies the column names and attributes from an existing table into a new table. No rows of data are inserted.

2 Which of the following PROC SQL steps creates a table that contains rows for the level-1 flight attendants only?
   a proc sql;
      create table work.newpayroll as
      select *
      from sasuser.payrollmaster
      where jobcode='FA1';
   b proc sql;
      create work.newpayroll as
      select *
      from sasuser.payrollmaster
      where jobcode='FA1';
   c proc sql;
      create table work.newpayroll
      copy sasuser.payrollmaster
      where jobcode='FA1';
   d proc sql;
      create table work.newpayroll as
      sasuser.payrollmaster
      where jobcode='FA1';

Correct answer: a

The CREATE TABLE statement that includes the AS keyword and query clauses creates a table and loads the results of the query into the new table. The WHERE clause selects only the rows for the level-1 flight attendants.

3 Which of the following statements is true regarding the UNDO_POLICY=REQUIRED option?
904  Chapter 5: Creating and Managing Tables Using PROC SQL  Appendix 1

a  It must be used with the REQUIRED integrity constraint.
b  It ignores the specified integrity constraints if any of the rows that you want
to insert or update do not meet the constraint criteria.
c  It restores your table to its original state if any of the rows that you try to
insert or update do not meet the specified integrity constraint criteria.
d  It allows rows that meet the specified integrity constraint criteria to be
inserted or updated, but rejects rows that do not meet the integrity constraint
criteria.

Correct answer: c

UNDO POLICY=REQUIRED is the default setting for PROC SQL. This setting
undoes all inserts or updates if 1 or more rows violate the integrity constraint
criteria, and restores the table to its original state before the inserts or updates.

4  Which of the following is not a type of integrity constraint?

a  CHECK
b  NULL
c  UNIQUE
d  PRIMARY KEY

Correct answer: b

The NOT NULL integrity constraint specifies that data is required and cannot
have a null (missing) value.

5  Which of the following PROC SQL steps deletes rows for all frequent-flyer
program members who traveled less than 10,000 miles?

a  proc sql;
   delete rows
   from work.frequentflyers
   where milestraveled < 10000;
b  proc sql;
   drop rows
   from work.frequentflyers
   where milestraveled < 10000;
c  proc sql;
   drop table
   from work.frequentflyers
   where milestraveled < 10000;
d  proc sql;
   delete
   from work.frequentflyers
   where milestraveled < 10000;

Correct answer: d

The DELETE statement deletes rows that are specified in the WHERE clause
from the table. If no WHERE clause is specified, all rows are deleted. The DROP
TABLE statement drops (deletes) an entire table; the syntax shown in option c is
not valid.

6  Which of the following PROC SQL steps gives bonuses (in points) to frequent-flyer
program members as follows:
   □ a 50% bonus for members who traveled less than 10,000 miles
   □ a 100% bonus for members who traveled 10,000 miles or more?

a  proc sql;
   update work.frequentflyers
set pointsearned=pointsearned* 
  case if milestraveled < 10000 
    then 1.5 
    if milestraveled >= 10000 
    then 2 
    else 1 
  end;

b proc sql;
  update work.frequentflyers 
  set pointsearned=pointsearned* 
  case when milestraveled < 10000 
    then 1.5 
    when milestraveled >= 10000 
    then 2 
    else 1 
  end;

c proc sql;
  update work.frequentflyers 
  set pointsearned=pointsearned* 
  case if milestraveled < 10000 
    then pointsearned*1.5 
    if milestraveled >= 10000 
    then pointsearned*2 
    else 1 
  end;

d proc sql;
  update work.frequentflyers 
  set pointsearned=pointsearned* 
  case if milestraveled < 10000 
    then pointsearned*1.5 
    if milestraveled >= 10000 
    then pointsearned*2 
    else pointsearned*1 
  end;

Correct answer: b

The UPDATE statement that includes a SET clause is used to modify rows in a table. WHEN-THEN clauses in the CASE expression enable you to update a column value based on specified criteria.

7 Which of the following statements is used to add new rows to a table?

a INSERT
b LOAD
c VALUES
d CREATE TABLE

Correct answer: a

The INSERT statement is used to insert new rows into a new or existing table. There is no LOAD statement in PROC SQL, VALUES is a clause, and the CREATE TABLE statement is used to create a table.

8 Which of the following statements regarding the ALTER TABLE statement is false?

a It allows you to update column attributes.
b It allows you to add new columns in your table.
c It allows you to drop columns in your table.
Chapter 6: Creating and Managing Indexes Using PROC SQL

1 Which of the following will create an index on the column EmpID for the table Sasuser.Staffmaster?
Quiz Answer Keys  Chapter 6: Creating and Managing Indexes Using PROC SQL

a proc sql;
create simple index(empid)
on sasuser.staffmaster;

b proc sql;
create empid index
on sasuser.staffmaster(empid);

c proc sql;
create simple index
on empid from sasuser.staffmaster;

d proc sql;
create index empid
on sasuser.staffmaster(empid);

Correct answer: d

The index specified above is based on one column, so it is a simple index. In the
CREATE INDEX statement, you specify the index name after the keywords
CREATE INDEX. You do not include a keyword to specify that this is a simple
index. The name of the key column is specified in parentheses after the table name.
The name of a simple index must be the same as the name of the key column.

2 Which keyword must you add to your index definition in the CREATE INDEX
statement to ensure that no duplicate values of the key column can exist?

a KEY
b UNIQUE
c NODUPS
d NODUPKEY

Correct answer: b

To create a unique index, the UNIQUE keyword is added to the CREATE
INDEX statement, between the keywords CREATE and INDEX.

3 Which of the following will create a composite index for the table
Sasuser.Flightdelays? (Sasuser.Flightdelays contains the following columns: Date,
FlightNumber, Origin, Destination, DelayCategory, DestinationType,
DayOfWeek, and Delay.)

a proc sql;
create index destination
on sasuser.flightdelays(flightnumber, destination);

b proc sql;
create composite index places
on sasuser.flightdelays (flightnumber, destination);

c proc sql;
create index on flightnumber,destination
from sasuser.flightdelays (places);

d proc sql;
create index places
on sasuser.flightdelays (flightnumber, destination);

Correct answer: d

A composite index is based on two or more columns. In the CREATE INDEX
statement, you specify the index name after the keywords CREATE INDEX. You
do not include a keyword to specify that this is a composite index. The names of the key columns are specified in parentheses after the table name. The name of a composite index cannot be the same as the name of any columns in the table.

4 Which of the following will write a message to the SAS log that shows whether PROC SQL has used an index?

a. options msglevel=i;
   proc sql;
   select *
   from sasuser.internationalflights
   where date between '01mar2000'd
   and '07mar2000'd;

b. options index=yes;
   proc sql;
   select *
   from sasuser.internationalflights
   where date between '01mar2000'd
   and '07mar2000'd;

c. proc sql;
   select * (idxwhere=yes)
   from sasuser.internationalflights
   where date between '01mar2000'd
   and '07mar2000'd;

d. proc sql;
   select * (msglevel=i)
   from sasuser.internationalflights
   where date between '01mar2000'd
   and '07mar2000'd;

Correct answer: a

Specifying the option MSGLEVEL=I causes informational messages about index usage to be written to the SAS log.

5 Which of the following will drop (delete) an index from a table?

a. proc sql;
   drop composite index flights
   from sasuser.marchflights;

b. proc sql;
   delete index flights
   on sasuser.staffmaster(flightnumber, date);

c. proc sql;
   drop index flights
   from sasuser.marchflights;

d. proc sql;
   delete index
   on sasuser.marchflights(flightnumber,
   flightdate);

Correct answer: c

The DROP INDEX statement drops one or more specified indexes from a table. You specify the name of each index to be dropped after the keywords DROP INDEX. The table name is specified after the keyword FROM. The type of index and the names of the indexed columns are not specified in the statement.
6. Which of the following statements will show you all the indexes that are defined for a table?
   a. DESCRIBE INDEX
   b. DESCRIBE TABLE
   c. SELECT
   d. IDXNAME
   Correct answer: b
   The DESCRIBE TABLE statement lists all indexes for one or more tables that you specify, along with other information about the table(s).

7. What is the purpose of specifying the data set option IDXWHERE=YES?
   a. It forces SAS to use the best available index to process the WHERE expression.
   b. It creates an index from the expression in the WHERE clause.
   c. It writes messages about index usage to the SAS log.
   d. It stops SAS from using any index.
   Correct answer: a
   The IDXWHERE=YES data set option tells SAS to use the best available index, even if the index does not optimize performance.

8. Which of the following is false regarding the use of an index?
   a. Equijoins can be performed without internal sorts.
   b. Indexes provide fast access to a small subset of data.
   c. Indexes can be created for numeric columns only.
   d. Indexes can enforce uniqueness.
   Correct answer: c
   Indexes can be created on either character or numeric columns.

9. Using an index is not likely to optimize a PROC SQL query in which of the following situations?
   a. The query contains an IN subquery that references the key column.
   b. The key column is specified in a WHERE clause expression that contains a comparison operator, the TRIM or SUBSTR function, the CONTAINS operator, or the LIKE operator.
   c. The query is an equijoin, and all the columns in the join expression are indexed in one of the tables being joined.
   d. The key column is specified only in a SELECT clause.
   Correct answer: d
   Using an index will optimize specific classes of PROC SQL queries. A query in which the key column is specified only in a SELECT clause is not one of these queries.

10. Which of the following is false regarding the IDXNAME= data set option?
    a. The specified index must exist.
    b. The specified index must be suitable by having at least its first or only column match a condition in the WHERE expression.
    c. The option allows you to create and name an index on the table.
    d. The option directs SAS to use an index that you specify.
    Correct answer: c
    The IDXNAME= data set option directs PROC SQL to use an index that you specify. The specified index must exist and must be suitable by having at least its first or only column match the condition in the WHERE expression.
Chapter 7: Creating and Managing Views Using PROC SQL

1 Which of the following statements is false regarding a PROC SQL view?
   a A view cannot be used in a join.
   b A view accesses the most current underlying data.
   c A view follows the same naming conventions as a table.
   d A view can be used in SAS programs in place of an actual SAS data file.
   Correct answer: a
   A PROC SQL view accesses the most current underlying data and can be joined with tables or other views. In addition, a PROC SQL view can
   □ be used in SAS programs in place of an actual SAS data file
   □ be derived from one or more tables, PROC SQL views, or DATA step views.

2 Which of the following statements describes an advantage of using a PROC SQL view?
   a Views often save space, because a view is usually quite small compared with the data that it accesses.
   b Views prevent users from continually submitting queries to omit unwanted columns or rows.
   c Views hide complex joins or queries from users.
   d all of the above
   Correct answer: d
   PROC SQL views are useful because they
   □ often save space (a view is usually quite small compared with the data that it accesses)
   □ prevent users from continually submitting queries to omit unwanted columns or rows
   □ hide complex joins or queries from users.
   In addition, PROC SQL views
   □ ensure that input data sets are always current, because data is derived from tables at execution time
   □ can be used to shield sensitive or confidential columns from users while enabling the same users to view other columns in the same table.

3 Which PROC SQL step creates a view that queries the table Sasuser.Payrollmaster?
   a proc sql;
      insert into sasuser.newview
      select * from sasuser.payrollmaster;
   b proc sql;
      create sasuser.newview as
      select * from sasuser.payrollmaster;
   c proc sql;
      create view sasuser.newview as
      select * from sasuser.payrollmaster;
   d proc sql;
      select * from sasuser.payrollmaster
      into view sasuser.newview;
Correct answer: c
You use the CREATE VIEW statement to create a view. The keywords CREATE VIEW are followed by the name of the view and the keyword AS.

4 Which of the following PROC SQL steps enables you to see a description of the view definition?

- a proc sql;
  select * from sasuser.payrollmasterv;
- b proc sql;
  describe view sasuser.payrollmasterv;
- c proc sql;
  list sasuser.payrollmasterv;
- d proc sql;
  contents view=sasuser.payrollmasterv;

Correct answer: b
The DESCRIBE VIEW statement displays the view definition in the SAS log.

5 Which PROC SQL step correctly references the view Data.Empview?

- a proc sql;
  select * from data.empview;
- b proc sql;
  select * from view data.empview;
- c proc sql;
  select view * from data.empview;
- d proc sql;
  select * from data where view='empview';

Correct answer: a
A view can be used in a PROC SQL step just as you would use an actual SAS table.

6 Which of the following PROC SQL steps correctly embeds a LIBNAME statement with a view definition?

- a proc sql;
  insert into sasuser.newview
  select * from airline.supervisors
  libname airline ‘c:\mysql’;
- b proc sql;
  create view sasuser.newview as
  from airline.supervisors
  embed libname airline ‘c:\mysql’;
- c proc sql;
  using airline ‘c:\mysql’;
  insert into sasuser.newview
  select * from airline.supervisors;
d proc sql;
   create view sasuser.newview as
   select * from airline.supervisors
       using libname airline 'c:\mysql';

Correct answer: d

The USING clause enables you to embed a LIBNAME statement in your view definition. The USING clause must be the last clause in the CREATE VIEW statement.

7 PROC SQL views can access data from
   a a SAS data file.
   b another PROC SQL view.
   c a relational database table.
   d all of the above

Correct answer: d

PROC SQL views can access data from a SAS data file, a DATA step view, a PROC SQL view, or a relational database table.

8 When you are working with PROC SQL views, it is best to
   a avoid using an ORDER BY clause in a view.
   b avoid creating views that are based on tables whose structure might change.
   c specify a one-level name in the FROM clause if the view resides in the same SAS data library as the contributing table(s).
   d all of the above

Correct answer: d

When you are working with PROC SQL views, it is best to
   □ avoid using an ORDER BY clause in a view. If you specify an ORDER BY clause, the data must be sorted each time the view is referenced.
   □ avoid creating views that are based on tables whose structure might change. A view is no longer valid when it references a nonexistent column.
   □ specify a one-level name in the FROM clause if the view resides in the same SAS data library as the contributing table(s). Using a one-level name in the FROM clause prevents you from having to change the view if you assign a different libref to the SAS data library that contains the view and its contributing table or tables.

9 You can update the data underlying PROC SQL view using the INSERT, DELETE, and UPDATE statements under which of the following conditions:
   a The view is joined or linked to another table.
   b The view contains a subquery.
   c The view contains a WHERE clause.
   d all of the above

Correct answer: c

You can update a PROC SQL view provided that the view does not join or link to another table, the view does not have a subquery, or you try to update a derived column. 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 an ORDER BY or a HAVING clause.

10 Which of the following programs drops (deletes) a view?
   a proc sql;
      delete sasuser.newview;
Quiz Answer Keys  Δ  Chapter 8: Managing Processing Using PROC SQL  913

b  proc sql;
   drop view sasuser.newview;

c  proc sql;
   erase view sasuser.newview;

d  proc sql;
   remove newview from sasuser;

Correct answer: b
The DROP VIEW statement drops a view from the specified library.

Chapter 8: Managing Processing Using PROC SQL

1  PROC SQL options are specified in
   a  the PROC SQL statement.
   b  an OPTIONS statement.
   c  a SELECT statement.
   d  the OPTIONS procedure.
Correct answer: a
PROC SQL options are specified in the PROC SQL statement. After you specify an option, it remains in effect until you change it or you re-invoke PROC SQL.

2  Which of the following options restricts the number of rows that PROC SQL takes as input from any single source?
   a  OUTOBS=
   b  INOBS=
   c  OBS=
   d  none of the above
Correct answer: b
The INOBS= option restricts the number of rows that PROC SQL takes as input from any single source. The INOBS= option is similar to the SAS system option OBS= and is useful for debugging queries on large tables. The OUTOBS= option restricts the number of rows that PROC SQL displays or writes to a table.

3  Which PROC SQL step creates the output shown below?

<table>
<thead>
<tr>
<th>Row</th>
<th>EmpID</th>
<th>JobCode</th>
<th>LastName</th>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1574</td>
<td>FA2</td>
<td>CAHILL</td>
<td>MARSHALL</td>
</tr>
<tr>
<td>2</td>
<td>1125</td>
<td>FA2</td>
<td>DUNLAP</td>
<td>DONNA</td>
</tr>
<tr>
<td>3</td>
<td>1475</td>
<td>FA2</td>
<td>EATON</td>
<td>ALICIA</td>
</tr>
<tr>
<td>4</td>
<td>1368</td>
<td>FA2</td>
<td>JEPSEN</td>
<td>RONALD</td>
</tr>
<tr>
<td>5</td>
<td>1411</td>
<td>FA2</td>
<td>JOHNSON</td>
<td>JACKSON</td>
</tr>
<tr>
<td>6</td>
<td>1441</td>
<td>FA2</td>
<td>LAWRENCE</td>
<td>KATHY</td>
</tr>
<tr>
<td>7</td>
<td>1477</td>
<td>FA2</td>
<td>MEYERS</td>
<td>FRESTON</td>
</tr>
<tr>
<td>8</td>
<td>1424</td>
<td>FA2</td>
<td>PATTERSON</td>
<td>RENEE</td>
</tr>
<tr>
<td>9</td>
<td>1413</td>
<td>FA2</td>
<td>PETERS</td>
<td>RANDALL</td>
</tr>
<tr>
<td>10</td>
<td>1555</td>
<td>FA2</td>
<td>RODRIGUEZ</td>
<td>JULIA</td>
</tr>
</tbody>
</table>
a proc sql nonumber outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

b proc sql number;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   reset nonumber outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

c proc sql nonumber;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   reset number outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

d proc sql;
   select *
      from sasuser.flightattendants
      where jobcode='FA1';
   reset outobs=10;
   select *
      from sasuser.flightattendants
      where jobcode='FA2';

Correct answer: c

After you specify an option, it remains in effect until you change it or you re-invoke PROC SQL. You can use the RESET statement to add, drop, or change PROC SQL options without re-invoking the SQL procedure. In the correct answer, the RESET statement adds the NUMBER option and the OUTOBS= option. The resulting output lists the first 10 rows in the table Sasuser.Flightattendants where the value of Jobcode equals FA2 and includes a column named Row.

4 Which of the following options does not affect the appearance of HTML, PDF, or RTF output?

a NUMBER | NONUMBER
b DOUBLE | NODOUBLE
c FLOW | NOFLOW | FLOW=n | FLOW=n m

Correct answer: d

The DOUBLE | NODOUBLE option specifies whether PROC SQL output is double-spaced in listing output. The FLOW | NOFLOW | FLOW=n | FLOW=n m option controls the appearance of wide character columns in listing output. Neither option affects the appearance of HTML output.

5 Which of the following statements is true regarding the STIMER option in PROC SQL?

a The STIMER option in PROC SQL writes timing information for each statement to the SAS log.
b The STIMER option in PROC SQL writes only cumulative timing information for the entire procedure to the SAS log.
c When using the STIMER option in PROC SQL, the SAS system option STIMER must also be in effect.
d a and c
Correct answer: d

The STIMER | NOSTIMER option in PROC SQL specifies whether PROC SQL writes timing information for each statement to the SAS log, instead of as a cumulative value for the entire procedure. NOSTIMER is the default. In order to use the STIMER option in PROC SQL, the SAS system option STIMER (the default) must also be in effect. If you use the system option alone, you will receive timing information for the entire procedure, not on a statement-by-statement basis.

6 A Dictionary table contains

a information about SAS data libraries.
b information about SAS data sets.
c information about SAS macros.
d all of the above
Correct answer: d

A Dictionary table is a special, read-only SAS data view that contains information about SAS data libraries, SAS data sets, SAS macros, and external files that are in use or available in the current SAS session. A Dictionary table also contains the settings for SAS system options that are currently in effect.

7 Dictionary tables are

a created each time they are referenced in a SAS program.
b updated automatically.
c limited to read-only access.
d all of the above
Correct answer: d

Dictionary tables are created each time they are referenced in a SAS program, updated automatically, and limited to read-only access. Accessing a Dictionary table causes SAS to determine the current state of the SAS session and return the information that you want.

8 Dictionary tables can be accessed

a by running a PROC SQL query against the table, using the Dictionary libref.
b by referring to the PROC SQL view of the table that is stored in the Sashelp library.
c by referring to the PROC SQL view of the table that is stored in the Sasuser library.
d a and b
Correct answer: d

Dictionary tables can be accessed by running a PROC SQL query against the table, using the Dictionary libref. Though SAS librefs are usually limited to eight characters, Dictionary is an automatically assigned, reserved word. You can also access a Dictionary table by referring to the PROC SQL view of the table that is stored in the Sashelp library.

9 Which of the following PROC SQL steps displays information about the Dictionary table Dictionary.Titles?

a proc sql;
   describe dictionary.titles;
b proc sql;
   describe table dictionary.titles;

c proc sql describe table dictionary.titles;

d proc sql describe dictionary titles;

Correct answer: b

To see how a Dictionary table is defined, submit a DESCRIBE TABLE statement. The DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the table specified in the DESCRIBE TABLE statement.

10 Which of the following PROC SQL steps displays the name (Memname), modification date (Modate), number of variables (Nvar), and the number of observations (Nobs) for each table in the Sasuser library?

a proc sql;
   select memname, modate, nvar, nobs
   from dictionary.tables
   where libname='SASUSER';

b proc sql;
   select memname, modate, nvar, nobs
   from dictionary.tables
   where libname='Sasuser';

c proc sql;
   select memname, modate, nvar, nobs
   from 'SASUSER'
   where table=dictionary.tables;

d proc sql;
   select SASUSER
   from dictionary.tables
   where cols= 'memname, modate, nvar, nobs';

Correct answer: a

To display information about the files in a specific library, specify the column names in a SELECT statement and the Dictionary table name in the FROM clause. The library name in the WHERE clause must be specified in uppercase letters because that is how it is stored in SAS and it must be enclosed in quotation marks.

Chapter 9: Introducing Macro Variables

1 Which of the following statements is false?

   a A macro variable can be defined and referenced anywhere in a SAS program except within data lines.
   b Macro variables are always user-defined, and their values remain constant until they are changed by the user.
   c Macro variables are text strings that are independent of SAS data sets.
   d The values of macro variables can be up to 65,534 characters long.

Correct answer: b

Macro variables are always text strings that are independent of SAS data sets. The value of a macro variable can be up to 65,534 characters long, and the name of a macro variable can be up to 32 characters long. A macro variable can be
defined or referenced anywhere in a SAS program except within data lines. There are two types of macro variables: automatic and user-defined.

2 Which of the following TITLE statements correctly references the macro variable month?
   a title "Total Sales for '&month' ";
   b title "Total Sales for 'month'";
   c title "Total Sales for &month";
   d title Total Sales for "&month";
   Correct answer: c
   To reference a macro variable, you precede the name with an ampersand. You do not need to enclose the macro variable reference in quotation marks.

3 Which of the following statements does not correctly display the value of the macro variable month in the SAS log?
   a options &month;
   b %PUT &month;
   c options symbolgen;
   d %PUT the macro variable MONTH has the value &month.;
   Correct answer: a
   There are two ways to display the value of a macro variable in the SAS log: you can turn on the SYMBOLGEN system option to list the values of all macro variables that are used, or you can use the %PUT statement to write specific text, including macro variable values, to the log.

4 Which statement will create a macro variable named location that has the value storage?
   a &let location = storage;
   b let &location = storage;
   c %let location = "storage";
   d %let location = storage;
   Correct answer: d
   You use the %LET statement to define a macro variable. You do not need to enclose the value in quotation marks. If you do include quotation marks in the assigned value for a macro variable, the quotation marks will be stored as part of the value.

5 What value will these statements assign to the macro variable reptitle:
   %let area = "Southeast";
   %let reptitle = * Sales Report for &area Area *
   a Sales Report for Southeast Area
   b Sales Report for "Southeast" Area
   c *Sales Report for "Southeast" Area*
   d * Sales Report for "Southeast" Area *
   Correct answer: d
   Macro variables are stored as character strings. Quotation marks and most special characters are stored exactly as they are assigned, but leading blanks are stripped from assigned values. You can also include references to other macro variables within %LET statements.
6 Assuming that you began your SAS session today, which of the following statements correctly sets the macro variable currdate to today’s date:

- a \%let currdate = %sysfunc(today(), worddate.);
- b \%let currdate = &sysdate9;
- c \%let currdate = %sysfunc(date());
- d all of the above

Correct answer: d

SYSDATE9 is an automatic macro variable that stores the date that your SAS session began in ddmmyyyy format. You can use the %SYSFUNC function along with any DATA step function, so both the TODAY() function and the DATE() function will result in the current date.

7 Macro character functions

- a can be used to manipulate character strings in macro variable values.
- b have the same basic syntax as the corresponding DATA step functions and yield similar results.
- c all of the above
- d none of the above

Correct answer: c

Macro character functions such as %UPCASE and %SUBSTR enable you to perform character manipulations on your macro variable values.

8 The four types of tokens that SAS recognizes are

- a expressions, literals, names, and special characters.
- b literals, names, numbers, and special characters.
- c expressions, names, numbers, and special characters.
- d expressions, literals, numbers, and special characters.

Correct answer: b

The word scanner recognizes four types of tokens. Expressions are not a type of token.

9 What are the resulting values for the macro variables that are defined here?

\%

- a month1 June
  month2 July
  period1 June July
  period2 May June
  period3 July Aug
- b month1 June
  month2 July
  period1 June July
  period2 May June
  period3 July Aug
- c month1 June
  month2 July
  period1 June July
  period2 May June
  period3 July Aug
Correct answer: c

You can combine macro variable references with text to create new text strings. If you precede a macro variable with text, the ampersand at the beginning of the macro variable name signals the end of the text and the beginning of a macro variable name. If you want text to follow the macro variable value, you must signal the end of the macro variable name with a period.

10 Which of the following correctly produces a title in which the current date is left justified in order to remove extra blanks?

a title "Report for %sysfunc(left(%sysfunc(today(),worddate.)))";
b title "Report for %sysfunc(left(today(), worddate.))";
c title "Report for %sysfunc(left(%qsysfunc(today(), worddate.)))";
d title "Report for %left(today(), worddate.)";

Correct answer: c

You use the %QSYSFUNC function in this case, in order to mask the comma that results from the worddate. format. You must mask this comma since the LEFT() function expects only one argument.
call symput('region', 'West');
end;
run;

b  data new;
   set sasuser.all;
   if location='Boston' then do;
      %let region=East;
   end;
   else
      %let region=West;
   end;
run;

c  data new;
   set sasuser.all;
   if location='Boston' then do;
      call symput(region, "East");
   end;
   else
      call symput(region, "West");
   end;
run;

d  data new;
   set sasuser.all;
   if location='Boston' then do;
      symput(region, East);
   end;
   else
      symput(region, West);
   end;
run;

Correct answer: a

To create a macro variable and assign to it a value that is based on the value of a DATA step variable, you use the SYMPUT routine. In the SYMPUT routine, to assign a literal string as a macro variable name, you enclose the literal in quotation marks. To assign a literal string as a value of the macro variable, you enclose the literal in quotation marks.

The SYMPUT routine cannot
a  be used to assign a data set variable as a value to a macro variable.
b  create a series of macro variables in one DATA step.
c  automatically convert a numeric value to a character value when used to assign a value to a macro variable in a DATA step.
d  be used to assign a numeric value to a macro variable in an SCL program.

Correct answer: d

The SYMPUT routine enables you to assign a data set variable as the value of a macro variable. You can also use the SYMPUT routine to create a series of related macro variables. Because all macro variable values are character strings, SYMPUT automatically converts any numeric value that you attempt to assign as a value for a macro variable. In an SCL program, you must use SYMPUTN rather than SYMPUT if you are attempting to assign a numeric value to a macro variable.

Which of the following programs correctly creates a series of macro variables whose names are values of the data set variable Course_code, then indirectly references one of those macro variables in a later step?
a data _null_;  
set sasuser.courses;  
call symput(course_code, trim(course_title));  
%let crsid=C005;  
proc print data=sasuser.schedule noobs label;  
where course_code="&crsid";  
var location begin_date teacher;  
title1 "Schedule for &c005";  
run;

b data _null_;  
set sasuser.courses;  
call symput(course_code, trim(course_title));  
run;  
%let crsid=C005;  
proc print data=sasuser.schedule noobs label;  
where course_code="&crsid";  
var location begin_date teacher;  
title1 "Schedule for &&&crsid";  
run;

c data _null_;  
set sasuser.courses;  
call symput(‘course_code’, trim(course_title));  
run;  
%let crsid=C005;  
proc print data=sasuser.schedule noobs label;  
where course_code="&crsid";  
var location begin_date teacher;  
title1 "Schedule for &&&crsid";  
run;

d data _null_;  
set sasuser.courses;  
call symget(course_code, trim(course_title));  
run;  
%let crsid=C005;  
proc print data=sasuser.schedule noobs label;  
where course_code="&crsid";  
var location begin_date teacher;  
title1 "Schedule for &&&crsid";  
run;

Correct answer: b

You can use multiple ampersands to create an indirect reference when the value of one macro variable is the name of another. If you enclose the DATA step variable name in quotation marks in the SYMPUT routine, the new macro variable will have the same name as the DATA step variable rather than having the DATA step variable’s value as a name. Use the SYMGET function to obtain the value of a macro variable during the execution of a DATA step.

5 Which of the following statements about the resolution of macro variable references is false?

a Two ampersands resolve to one ampersand.

b If more than four consecutive ampersands precede a name token, the macro processor generates an error message.
c  Re-scanning continues until there are no remaining macro triggers that the macro processor can resolve.

d  The macro processor always re-scans a name token that is preceded by multiple ampersands or by multiple percent signs.

Correct answer: b

If more than four consecutive ampersands precede a name token, rescanning continues from left to right until no more triggers can be resolved. The Forward Re-scan rule describes how the macro processor resolves macro variable references that start with multiple ampersands or with multiple percent signs.

6  In which of the following situations would you use SYMGET rather than a macro variable reference (&macvar)?

   a  to create a DATA step variable from a macro variable value during the execution of the DATA step

   b  to include a macro variable reference in a PROC SQL view

   c  to access the value of a macro variable during the execution of an SCL program

   d  all of the above

Correct answer: d

A macro variable reference (&macvar) is resolved before any SAS language statements are sent to the compiler. The SYMGET function enables you to obtain the value of a macro variable during the execution of a DATA step or a PROC SQL step. The SYMGET function can also be used to obtain the value of a macro variable during the execution of an SCL program.

7  Which of the following correctly creates a macro variable in a PROC SQL step?

   a  call symput(daily_fee, put(fee/days, dollar8.));

   b  %let daily_fee=put(fee/days, dollar8.)

   c  select fee/days format=dollar8.
      into :daily_fee from sasuser.all;

   d  select fee/days format=dollar8.
      into daily_fee from sasuser.all;

Correct answer: c

To create a macro variable during the execution of a PROC SQL step, use the INTO clause of the SELECT statement. In the INTO clause, you precede the name of the macro variable with a colon.

8  According to the global symbol table shown here, what value will a reference to &TEACH&crs resolve to?

<table>
<thead>
<tr>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEACH1</td>
</tr>
<tr>
<td>TEACH2</td>
</tr>
<tr>
<td>TEACH3</td>
</tr>
<tr>
<td>CRS</td>
</tr>
</tbody>
</table>

   a  &TEACH3

   b  TEACH3
c Forest, Mr. Peter  
d none of the above  
Correct answer: c

You can use multiple ampersands to delay the resolution of a macro variable reference. You can also combine macro variable references in order to create new tokens. In this example, the reference &teach&crs resolves to &teach3 on the first scan. On the next scan, &teach3 resolves to Forest, Mr. Peter.

9 Which of the following statements correctly creates a DATA step variable named Price and assigns to it the value of the macro variable daily_fee during DATA step execution?

a price=&daily_fee;
b price=symget(daily_fee);
c price=symget(&daily_fee);
d price=symget("daily_fee");

Correct answer: d

You can use the SYMGET function in an assignment statement to obtain the current value of a macro variable and to assign that value to a DATA step variable. The SYMGET function enables you to obtain the value of a macro variable during execution of a DATA step, a PROC SQL step, or an SCL program.

10 Which of the following is false?

a The SYMPUT routine can be used to create a macro variable during execution of the DATA step or during execution of an SCL program.
b In the DATA step, the SYMPUT routine automatically converts to a character value any numeric value that you attempt to assign as the value of a macro variable.
c PROC SQL automatically converts to a numeric value any macro variable value that you attempt to compare to a numeric value.
d In an SCL program, the SYMPUTN routine can be used to assign a numeric value to a macro variable.

Correct answer: c

The SYMPUT routine can be used in either the DATA step or in an SCL program. In the DATA step, the SYMPUT routine will perform automatic conversion on numeric values that you attempt to assign as values for macro variables, using the BEST12. format. In an SCL program, you should use the SYMPUTN routine if you want to assign a numeric value as a value for a macro variable. In a PROC SQL step, you need to use the INPUT function in order to convert macro variable values to numeric before you compare them to other numeric values.

---

Chapter 11: Creating and Using Macro Programs

1 Which of the following is false?

a A %MACRO statement must always be paired with a %MEND statement.
b A macro definition can include macro variable references, but it cannot include SAS language statements.
c Only macro language statements are checked for syntax errors when the macro is compiled.
d  Compiled macros are stored in a temporary SAS catalog by default.
Correct answer: b

A macro definition must begin with a `%MACRO` statement and must end with a
`%MEND` statement. The macro definition can include macro language statements
as well as SAS language statements. When the macro is compiled, macro language
statements are checked for syntax errors. The compiled macro is stored in a
temporary SAS catalog by default.

2 Which of the following examples correctly defines a macro named `Print` that
includes parameters named `vars` and `total`?

a  `%macro print(vars, total);
   proc print data=classes;
   var vars;
   sum total;
   run;
%end
%end print;

b  `%macro print('vars', 'total');
   proc print data=classes;
   var &vars;
   sum &total;
   run;
%end print;

c  `%macro print(vars, total);
   proc print data=classes;
   var &vars;
   sum &total;
   run;
%end print;

d  `%macro print(vars, total);
   proc print data=classes;
   var :vars;
   sum :total;
   run;
%end print;

Correct answer: c

To include positional parameters in a macro definition, you list the parameters in
parentheses and separate them with commas. When the macro is executed, macro
variables will be created in the local symbol table and will have the same names
as the parameters. You can then use these macro variables within the macro.

3 Which of the following correctly references the macro named `Printdsn` as shown
here:

```sas
%macro printdsn(dsn,vars);
   %if &vars= %then %do;
      proc print data=&dsn;
      title "Full Listing of %upcase(&dsn) data set";
      run;
   %end;
   %else %do;
      proc print data=&dsn;
      var &vars;
      title "Listing of %upcase(&dsn) data set";
      run;
```
Quiz Answer Keys  Chapter 11: Creating and Using Macro Programs  925

%end;
%mend;
a %printdsn(sasuser.courses, course_title days);
b %printdsn(dsn=sasuser.courses, vars=course_title days)
c %printdsn(sasuser.courses, course_title days)
d %printdsn(sasuser.courses, course_title, days)
Correct answer: c

To call a macro that includes positional parameters, you precede the macro name with a percent sign. You list the values for the macro variables that are defined by the parameters in parentheses. List values in the same order in which the parameters are listed, and separate them with commas. Remember that a macro call is not a SAS language statement and does not require a semicolon.

4 If you use a mixed parameter list in your macro program definition, which of the following is false?
a You must list positional parameters before any keyword parameters.
b Values for both positional and keyword parameters are stored in a local symbol table.
c Default values for keyword parameters are the values that are assigned in the macro definition, whereas positional parameters have a default value of null.
d You can assign a null value to a keyword parameter in a call to the macro by omitting the parameter from the call.
Correct answer: d

In a mixed parameter list, positional parameters must be listed before any keyword parameters. Both positional and keyword parameters create macro variables in the local symbol table. To assign a null value to a keyword parameter, you list the parameter without a value in the macro call.

5 Which of the following is false?
a A macro program is compiled when you submit the macro definition.
b A macro program is executed when you call it ($macro-name).
c A macro program is stored in a SAS catalog entry only after it is executed.
d A macro program is available for execution throughout the SAS session in which it is compiled.
Correct answer: c

When you submit a macro definition, the macro is compiled and is stored in a SAS catalog. Then when you call the macro, the macro is executed. The macro is available for execution anytime throughout the current SAS session.

6 When you use an %IF-%THEN statement in your macro program,
a you must place %DO and %END statements around code that describes the conditional action, if that code contains multiple statements.
b the %ELSE statement is optional.
c you cannot refer to DATA step variables in the logical expression of the %IF statement.
d all of the above.
Correct answer: d

You can use %IF-%THEN statements to conditionally process code. Within a %IF-%THEN statement, you must use %DO and %END statements to enclose multiple statements. %IF-%THEN statements are similar to IF THEN statements in the DATA step, but they are part of the macro language.
7 Which of the following can be placed onto the input stack?
   a only whole steps.
   b only whole steps or whole statements.
   c only whole statements or pieces of text within a statement.
   d whole steps, whole statements, or pieces of text within statements.
Correct answer: d

   By using %IF-%THEN statements, you can place whole steps, individual statements, or parts of statements onto the input stack.

8 Which of the following will create a macro variable named class in a local symbol table?
   a data _null_;  
       set sasuser.courses;
       %let class=course_title;
       run;
   b data _null_;  
       set sasuser.courses;
       call symput('class', course_title);
       run;
   c %macro sample(dsn);
       %local class;
       %let class=course_title;
       data _null_;  
       set &dsn;
       run;
       %mend;
   d %global class;
       %macro sample(dsn);
       %let class=course_title;
       data _null_;  
       set &dsn;
       run;
       %mend;
Correct answer: c

   There are several ways to create macro variables in the local symbol table. Macro variables that are created by parameters in a macro definition or by a %LOCAL statement are always created in the local table. Macro variables that are created by a %LET statement or by the SYMPUT routine inside a macro definition might be created in the local table as well.

9 Which of the following examples correctly defines the macro program Hex?
   a %macro hex(start=1, stop=10, incr=1);
      %local i;
      data _null_;  
      %do i=start to stop by incr;
         value=&i;
         put "Hexadecimal form of &i is " value hex6.;
      %end;
      run;
      %mend hex;
   b %macro hex(start=1, stop=10, incr=1);
      %local i;
data _null_;  
%do i=&start %to &stop %by &incr;  
  value=&i;  
  put "Hexadecimal form of &i is " value hex6.;  
%end;  
run;  
%mend hex;  

c %macro hex(start=1, stop=10, incr=1);  
%local i;  
data _null_;  
%do i=&start to &stop by &incr;  
  value=&i;  
  put "Hexadecimal form of &i is " value hex6.;  
%end  
run;  
%mend hex;  
d %macro hex(start=1, stop=10, incr=1);  
%local i;  
data _null_;  
%do i=&start to &stop by &incr;  
  value=&i;  
  put "Hexadecimal form of &i is " value hex6.;  
%end  
run;  
%mend hex;  

Correct answer: b

To define macros with %DO loops you use a %DO statement and a %END statement. Be sure to precede all keywords in the statements with percent signs since the %DO and %END statements are macro language statements. Also, be sure to end these statements with semicolons.

10 When you submit a call to a compiled macro, what happens?

a First, the macro processor checks all macro programming statements in the macro for syntax errors.
   Then the macro processor executes all statements in the macro.

b The macro processor executes compiled macro programming statements.
   Then any SAS programming language statements are executed by the macro processor.

c First, all compiled macro programming statements are executed by the macro processor.
   After all macro statements have been processed, any SAS language statements are passed back to the input stack in order to be passed to the compiler and then executed.

d The macro processor executes compiled macro statements.
   If any SAS language statements are encountered, they are passed back to the input stack.
   The macro processor pauses while those statements are passed to the compiler and then executed.
   Then the macro processor continues to repeat these steps until it reaches the %MEND statement.

Correct answer: d

When you submit a call to a compiled macro, the macro is executed. Specifically, the macro processor executes compiled macro language statements first. When any SAS language statements are encountered, the macro processor places these
statements onto the input stack and pauses while they are passed to the compiler and then executed. Then the macro processor continues to repeat these steps until the %MEND statement is reached.

Chapter 12: Storing Macro Programs

1 The %INCLUDE statement
   a can be used to insert the contents of an external file into a program.
   b will cause a macro definition that is stored in an external file to be compiled when the contents of that file are inserted into a program and submitted.
   c can be specified with the SOURCE2 option in order to write the contents of the external file that is inserted into a program to the SAS log.
   d all of the above
   Correct answer: d

   The %INCLUDE statement can be used to insert the contents of an external file into a SAS program. If a macro definition is stored in an external file, the %INCLUDE statement causes the macro definition to be compiled when it is inserted into the SAS program. The contents of the macro definition will be written to the SAS log only if the SOURCE2 option is specified.

2 If you store a macro definition in a SAS catalog SOURCE entry
   a the macro definition can be submitted for compilation by using the FILENAME and %INCLUDE statements.
   b you can use the PROC CATALOG statement to compile the macro.
   c the SOURCE entry will be deleted at the end of the session.
   d you do not need to compile the macro before you invoke it in a program.
   Correct answer: a

   When a macro definition is stored as a catalog SOURCE entry, you must compile it before you can call it from a SAS program. You compile a macro that is stored as a catalog SOURCE entry by using the CATALOG access method. This creates a session-compiled macro that will be deleted at the end of the SAS session. The PROC CATALOG statement enables you to view a list of the contents of a SAS catalog.

3 Which of the following programs correctly sets the appropriate system options and calls the macro Prtlast? Assume that Prtlast is stored in an autocall library and that it has not been compiled during the current SAS session.
   a options mautosource;
      %prtlast
   b libname mylib 'c:\mylib';
      filename macsbc catalog 'mylib.macsbc';
      %prtlast
   c libname mylib 'c:\mylib';
      filename macsbc 'mylib.macsbc';
      options mautosource sasautos=(macsrc, sasautos);
      %prtlast
   d libname mylib 'c:\mylib';
      options mautosource sasautos=mylib;
      %prtlast
Correct answer: c

To call a macro that is stored in an autocall library, you must specify both the MAUTOSOURCE system option and the SASAUTOS system option. The SASAUTOS system option can be set to include multiple pathnames or filerefs. Once these two system options are set, you call the macro by preceding the macro name with a percent sign.

4 If you use the Stored Compiled Macro Facility,
   a the macro processor does not compile a macro every time it is used.
   b the only compiled macros that the Stored Compiled Macro Facility can access are those that are stored in the Sasmacr catalog.
   c you need to specify the MSTORED and SASMSTORE system options.
   d all of the above
Correct answer: d

The Stored Compiled Macro Facility enables you to store compiled macros permanently so that you can reuse them in later SAS sessions without compiling them again. Compiled macros must be stored in a catalog named Sasmacr, and both the MSTORED system option and the SASMSTORE system option must be specified.

5 Which of the following correctly creates a permanently stored compiled macro?
   a libname macrolib 'c:\mylib';
      options sasmstore;
      %macro prtlast / store;
      proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
      run;
      %mend;
   b libname macrolib 'c:\mylib';
      options mstored sasmstore=macrolib;
      %macro prtlast / store;
      proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
      run;
      %mend;
   c libname macrolib 'c:\mylib';
      options mstored sasmstore=macrolib;
      %macro prtlast;
      proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
      run;
      %mend;
   d libname macrolib 'c:\mylib';
      %macro prtlast / store;
      proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
      run;
      %mend;
Correct answer: b

In order to create a permanently stored compiled macro, you must specify the MSTORED system option. The SASMSTORE system option must be specified to point to the library in which you want your macros to be stored. You must also use the STORE option in the %MACRO statement.
6 When you submit the following code, what happens?

```sas
%macro prtlast;
   proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
   run;
%mend;
```

a A session-compiled macro named `Prtlast` is stored in `Work.Sasmacr`.
b A macro named `Prtlast` is stored in the autocall library.
c The `Prtlast` macro is stored as a stored compiled macro.
d The `Prtlast` macro is stored as a SOURCE entry in a permanent SAS catalog.
Correct answer: a

When you submit a macro definition, SAS creates a session-compiled macro and stores it in the temporary SAS catalog `Work.Sasmacr`. This macro will be deleted at the end of the SAS session.

7 Why would you want to store your macros in external files?

a You could easily share your macros with others.
b You could edit your macros with any text editor.
c Your macros would be available for use in later SAS sessions.
d all of the above
Correct answer: d

If you store your macro definitions in external files, you can easily share these files with others. Also, you can edit a macro definition that is stored in an external file with any text editor, and you can reuse the macro in other SAS sessions.

8 What will the following PROC CATALOG step do?

```sas
proc catalog cat=mylib.sasmacr;
   contents;
quit;
```

a Copy the contents of the `Sasmacr` catalog to a temporary data set.
b List the contents of the `Sasmacr` catalog as output.
c Copy the contents of the output window to the `Sasmacr` catalog.
d none of the above
Correct answer: b

The PROC CATALOG step enables you to view a list of the contents of a SAS catalog. This might be especially useful if you store your macro definitions as SOURCE entries in permanent SAS catalogs. You might also use the PROC CATALOG step to see a list of the session-compiled macros that are stored in `Work.Sasmacr`.

9 Which of the following is not true about stored compiled macros?

a Because these stored macros are compiled, you should save and maintain the source for the macro definitions in a different location.
b The Stored Compiled Macro Facility compiles and saves compiled macros in a permanent catalog, in a library that you specify.
c You do not need to specify any system options in order to use the Stored Compiled Macro Facility.
d You cannot move a stored compiled macro to another operating system.
Correct answer: c

In order to use the Stored Compiled Macro Facility, you need to specify the MSTORED and SASMSTORE system options. The Stored Compiled Macro Facility saves the compiled macro in a permanent SAS catalog, but it does not
save the macro definition. You cannot move a compiled macro across operating systems. Since you cannot re-create the macro definition from a compiled macro, it is a good idea to save your source program permanently as well.

10 Which of the following is not true?

a The autocall macro facility stores compiled SAS macros in a collection of external files called an autocall library.

b Autocall libraries can be concatenated together.

c One disadvantage of the autocall facility is that the first time you call an autocall macro in a SAS session, the macro processor must use system resources to compile it.

d The autocall facility can be used in conjunction with the Stored Compiled Macro Facility.

Correct answer: a

The autocall macro facility stores macro definitions — not compiled macros — permanently. The first time an autocall macro is called during a SAS session, the macro is compiled and a session-compiled macro is created in Work.Sasmacr. You can have multiple autocall libraries that are concatenated, and you can use the autocall facility in conjunction with the Stored Compiled Macro Facility.

Chapter 13: Creating Samples and Indexes

1 The variable that is created by the POINT= option is assigned a value

a automatically during compilation of the DATA step.

b automatically during execution of the DATA step.

c during compilation of the DATA step, by program statements.

d during execution of the DATA step, by program statements.

Correct answer: d

The POINT= option in the SET statement names a variable. You must use program statements to assign a value to this variable during execution of the DATA step, before execution of the SET statement. Also, the value of the POINT= variable should be a number that corresponds to an observation number in the input data set, and it should be different each time the SET statement executes.

2 Which of the following programs correctly creates a systematic sample from a data set with an unknown number of observations and outputs these sample observations to a data set named Sample?

a data sample;
   set sasuser.sale2000 point=thisone nobs=totnum;
   output;
   stop;
run;

b data sample;
   do thisone=100 to totnum by 100;
      set sasuser.sale2000 point=thisone nobs=totnum;
      output;
   end;
   stop;
run;
c data sample;
   do thisone=100 to 1000 by 100;
   set sasuser.sale2000 point=thisone;
   output;
   end;
   stop;
run;

d data sample;
   do thisone=100 to totnum by 100;
   set sasuser.sale2000 point=thisone nobs=totnum;
   end;
run;

Correct answer: b

To create a systematic sample from a data set that has an unknown number of observations, you use the NOBS= option in conjunction with the POINT= option in the SET statement. The NOBS= variable is automatically assigned a value of the total number of observations in the input data set, and you must assign a value to the POINT= variable before the SET statement executes.

3 Which of the following expressions will generate a random integer between 1 and 50?
   a ceil(ranuni(50))
   b ranuni(50)
   c ceil(ranuni(0)*50)
   d ceil(ranuni(0))*50

Correct answer: c

In order to create a random sample of a data set, you need to generate a random integer. You can use the RANUNI function in conjunction with the CEIL function to create a random integer. You can use a multiplier with the RANUNI function to increase the range from which the random number is chosen to include as many numbers as you need.

4 An index
   a is an optional file that is associated with a data set.
   b provides direct access to specific observations of a data set, based on the value of one or more key variables.
   c can be classified as simple or composite, either of which can consist of unique values.
   d all of the above

Correct answer: d

An index is a separate file from a data set that contains information about observations within the data set. Specifically, an index contains value/identifier pairs that indicate the location of observations within the data set and the value of one or more key variables in that observation.

5 Which of the following correctly creates a data set named Flights from the Sasuser.Revenue data set, creates a composite index named Fromto that is based on the values of Origin and Dest, and prints informational messages about the index to the SAS log?
   a options msglevel=i;
      data flights index=(Fromto=origin dest);
      set sasuser.revenue;
b options msglevel=n;
    data flights (index=(Fromto=origin dest));
    set sasuser.revenue;
    run;

c options msglevel=i;
    data flights (index=(Fromto=(origin dest)));
    set sasuser.revenue;
    run;

d options msglevel=n;
    data flights (index=Fromto);
    set sasuser.revenue;
    run;

Correct answer: c

To create an index at the same time that you create a data set, you use the INDEX= option in the DATA statement. You must assign a unique name to a composite index, while a simple index is automatically assigned the name of the key variable as its name. You can set the value of the MSGLEVEL= system option to I in order to see messages about indexes in the SAS log.

Which of the following is true?

a When you add observations to a data set, the index(es) are automatically updated with additional value/identifier pairs.
b When you rename a variable that is used as the key variable in a simple index, you must re-create the index.
c When you delete a data set, the index file remains until you delete it as well.
d When you copy a data set with the COPY statement, you must also copy the index file in another step.

Correct answer: a

For many maintenance tasks that you perform on a data set, SAS automatically performs corresponding tasks to the index file. For example, if you delete a data set, the index file is deleted as well. If you rename a data set with the CHANGE statement in the DATASETS procedure, SAS automatically renames the index file. If you copy a data set to a new location with the COPY statement in the DATASETS procedure, SAS automatically reconstructs the index file in the new location.

To create an index on an existing data set, you use

a PROC DATASETS.
b PROC SQL.
c the DATA step with the INDEX= option, to rebuild the data set.
d any of the above

Correct answer: d

You can use the DATASETS procedure or the SQL procedure to create an index on or delete an index from an existing data set. You can also rebuild the index with a DATA step and use the INDEX= option to create an index on the rebuilt data set. However, rebuilding a data set uses more system resources than adding an index to an existing data set with either the DATASETS or the SQL procedure.

Which of the following correctly creates a simple index named Origin on the Revenue data set?

a proc sql;
    create index origin on revenue(origin);
Correct answer: a

You use the CREATE INDEX statement of the SQL procedure to create an index on an existing data set. In the SQL procedure, you must name the index in the CREATE INDEX statement; for a simple index, the index name must match the name of the key variable.

9 To view a list of the indexes that are associated with a data set, you use

a PROC COPY or the COPY statement in PROC DATASETS.  
b PROC CONTENTS or the CONTENTS statement in PROC DATASETS.  
c the MSGLEVEL= system option and a PROC PRINT step.  
d any of the above

Correct answer: b

You can use either the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure to generate a list of information about a data set, including a list of existing indexes. All indexes for a data set are stored in a single file that is separate from but has the same name as the data set.

10 Suppose that the Sasuser.Revenue data set has a simple index named FlightID. For which of the following programs will the index be used?

a proc print data=sasuser.revenue;  
   where flightid ne 'IA11200';  
   run;

b data someflights;  
   set sasuser.revenue;  
   where flightid > 'IA11200';  
   run;

c data someflights;  
   set sasuser.revenue;  
   if flightid > 'IA11200';  
   run;

d proc print data=sasuser.revenue;  
   where origin='RDU' or flightid='IA03400';  
   run;

Correct answer: b

An index can improve the efficiency with which SAS is able to access certain observations in a data set. However, an index is not always useful. SAS will not use an index to process subsetting IF statements, or other statements that SAS determines might be more efficiently processed without an index.
1 Which of the following statements associates the fileref OnSale with the raw data files London.dat, Paris.dat, and Zurich.dat? The files are stored in the C:\Routes\New directory in the Windows operating environment.

a filename onsale (c:\routes\new\london.dat, c:\routes\new\paris.dat, c:\routes\new\zurich.dat);

b filename onsale ‘c:\routes\new\london.dat’ ‘c:\routes\new\paris.dat’ ‘c:\routes\new\zurich.dat’;

c filename onsale (‘c:\routes\new\london.dat’ ‘c:\routes\new\paris.dat’ ‘c:\routes\new\zurich.dat’);

d filename onsale ‘c:\routes\new\london.dat c:\routes\new\paris.dat c:\routes\new\zurich.dat’;

Correct answer: c

When a FILENAME statement is used to assign a fileref to multiple raw data files, the list of files must be enclosed in a single set of parentheses. Each filename specified must be enclosed in quotation marks.

2 Which of the following statements is true?

a The FILEVAR= option can be used to dynamically change the currently opened input file to a new physical file.

b The FILEVAR= variable is not written to the data set.

c The FILEVAR= variable must contain a character string that is a physical filename.

d all of the above

Correct answer: d

The FILEVAR= option enables you to dynamically change the currently opened input file to a new input file. The FILEVAR= variable must contain a character string that is a physical filename. Like automatic variables, the FILEVAR= variable is not written to the data set.

3 Given the following program, which table correctly shows the corresponding values of the variables x and readfile?

```r
data work.revenue;
  do x = 8, 9, 10;
    readfile=compress("c:\data\month" !!put(x,2.)!!".dat"," ‘ ‘");
    do until (lastobs);
      infile temp filevar=nextfile
      end=lastobs;
      input Date : date7. Location $ Sales : dollar10.2;
      output;
    end;
  end;
stop;
```

run;

a

<table>
<thead>
<tr>
<th>When x=</th>
<th>readfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>month8.dat</td>
</tr>
<tr>
<td>9</td>
<td>month9.dat</td>
</tr>
<tr>
<td>10</td>
<td>month10.dat</td>
</tr>
</tbody>
</table>

b

<table>
<thead>
<tr>
<th>When x=</th>
<th>readfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>c:\data\month8.dat</td>
</tr>
<tr>
<td>9</td>
<td>c:\data\month9.dat</td>
</tr>
<tr>
<td>10</td>
<td>c:\data\month10.dat</td>
</tr>
</tbody>
</table>

c

<table>
<thead>
<tr>
<th>When x=</th>
<th>readfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>c:\data\month 8.dat</td>
</tr>
<tr>
<td>9</td>
<td>c:\data\month 9.dat</td>
</tr>
<tr>
<td>10</td>
<td>c:\data\month10.dat</td>
</tr>
</tbody>
</table>

d

<table>
<thead>
<tr>
<th>When x=</th>
<th>readfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>month8</td>
</tr>
<tr>
<td>9</td>
<td>month9</td>
</tr>
<tr>
<td>10</td>
<td>month10</td>
</tr>
</tbody>
</table>

Correct answer: b

The DO statement creates the index variable \( x \) and assigns it the values of 8, 9, and 10. The assignment statement assigns the name of a raw data file to \texttt{readfile} using the current value of \( x \) and the PUT function, which concatenates the values of \( x \) with the text strings \texttt{c:\data\month and .dat}. The COMPRESS function removes blank spaces from the values of \texttt{readfile}.

4 If the current date is March 30, 2003, which table correctly shows the corresponding values of the variables \( y_1, y_2, y_3 \), and \texttt{nextfile}?

```plaintext
data work.quarter (drop=monthnum midmon lastmon);
y3=year(today());
y2=y3-1;
y1=y3-2;
do i = y3, y2, y1;
  nextfile="c:\data\Y"!!put(i,4.)!!".dat";
```
do until (lastobs);
  infile temp filevar=nextfile  
    end=lastobs;
  input Flight $ Origin $ Dest $  
    Date : date9.;
  output;
  end;
end;
stop;
run;

Correct answer: a

The TODAY function returns the current date from the system clock as a SAS date value. The year number is then extracted from the current date using the YEAR function. The value of the current year, 2003, is assigned to \(y_3\). The year values 2002 and 2001 are assigned to \(y_2\) and \(y_1\), respectively. The PUT function

<table>
<thead>
<tr>
<th>When i=</th>
<th>nextfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_1)</td>
<td>c:\data\Y2001.dat</td>
</tr>
<tr>
<td>(y_2)</td>
<td>c:\data\Y2002.dat</td>
</tr>
<tr>
<td>(y_3)</td>
<td>c:\data\Y2003.dat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When i=</th>
<th>nextfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_1)</td>
<td>Y2001.dat</td>
</tr>
<tr>
<td>(y_2)</td>
<td>Y2002.dat</td>
</tr>
<tr>
<td>(y_3)</td>
<td>Y2003.dat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When i=</th>
<th>nextfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_1)</td>
<td>c:\data\Y2003.dat</td>
</tr>
<tr>
<td>(y_2)</td>
<td>c:\data\Y2002.dat</td>
</tr>
<tr>
<td>(y_3)</td>
<td>c:\data\Y2001.dat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When i=</th>
<th>nextfile=</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_1)</td>
<td>c:\data\Y3.dat</td>
</tr>
<tr>
<td>(y_2)</td>
<td>c:\data\Y2.dat</td>
</tr>
<tr>
<td>(y_3)</td>
<td>c:\data\Y1.dat</td>
</tr>
</tbody>
</table>
concatenates the text string `c:\data\Y` with the year values and the text string `.dat`.

5 Which of the following statements is false?

a The END= variable is set to 0 when SAS processes the last data record in the input file.
b The END= variable is set to 1 when SAS processes the last data record in the input file.
c The END= variable is not written to the data set.
d a and c

Correct answer: a

The END= option enables you to name a variable whose value is controlled by SAS. The value of the variable is 0 when you are not reading the last record in an input file and 1 when you are reading the last record in an input file. You can test the value of the END= variable to determine if the DATA step should continue processing. Like automatic variables, the END= variable is not written to the SAS data set.

6 Which program appends `Work.London` to `Work.Flights`?

Data Set Description for Work.London

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Destination</td>
<td>char</td>
<td>3</td>
</tr>
<tr>
<td>Departure</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Gate</td>
<td>char</td>
<td>2</td>
</tr>
</tbody>
</table>

Data Set Description for Work.Flights

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Destination</td>
<td>char</td>
<td>3</td>
</tr>
<tr>
<td>Departure</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Gate</td>
<td>char</td>
<td>2</td>
</tr>
</tbody>
</table>

a proc append base=work.london
data=work.flights;
run;
b proc append data=work.london
base=work.flights;
run;
c proc append data=work.london work.flights;
run;
d proc append data=work.flights work.london;
run;

Correct answer: b

PROC APPEND uses the BASE= and DATA= arguments. BASE=SAS-data-set names the data set to which you want to add observations. DATA=SAS-data-set names the SAS data set containing observations that you want to append to the end of the SAS data set specified in the BASE= argument.

7 What happens when the following program is submitted?

```sas
proc append base=staff.marketing
data=staff.sales force;
```
run;

Data Set Description for Staff.Marketing

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>char</td>
<td>12</td>
</tr>
<tr>
<td>FirstName</td>
<td>char</td>
<td>10</td>
</tr>
<tr>
<td>EmpID</td>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>Office</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Phone</td>
<td>char</td>
<td>12</td>
</tr>
</tbody>
</table>

Data Set Description for Staff.Sales

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>char</td>
<td>20</td>
</tr>
<tr>
<td>FirstName</td>
<td>char</td>
<td>10</td>
</tr>
<tr>
<td>EmpID</td>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>Office</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>Phone</td>
<td>char</td>
<td>12</td>
</tr>
</tbody>
</table>

a The length of **LastName** is converted to 20 in **Staff.Marketing**.
b **LastName** is dropped from **Staff.Marketing**.
c Missing values are assigned to **LastName** observations that are read in from **Staff.Sales**.
d Some of the values of **LastName** may be truncated in the observations that are read in from **Staff.Sales**.

Correct answer: d

If a DATA= data set contains variables that are longer than the corresponding variables in the BASE= data set, the FORCE option must be used with PROC APPEND. Using the FORCE option enables you to append the data sets. However, some of the variable values may be truncated in the observations that are read in from the DATA= data set.

8 Which program appends **Work.April** to **Work.Y2003**?

Data Set Description for **Work.Y2003**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>FirstClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>BusinessClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Coach</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

Data Set Description for **Work.April**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightNum</td>
<td>char</td>
<td>4</td>
</tr>
<tr>
<td>FirstClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>BusinessClass</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Coach</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

a proc append base=work.y2003
data=work.april;
run;
b proc append base=work.april
data=work.y2003 force;
run;
c proc append base=work.y2003
data=work.april force;
run;
d) proc append base=work.april
   data=work.y2003;
run;

Correct answer: c

You must use the FORCE option with PROC APPEND when the DATA= data set contains a variable that does not have the same type as the corresponding variable in the BASE= data set.

9 What happens when the SAS data set Work.NewHires is appended to the SAS data set Work.Employees using PROC APPEND?

Data Set Description for Work.Employees

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>EmpID</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Name</td>
<td>char</td>
<td>20</td>
</tr>
<tr>
<td>Room</td>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>Extension</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

Data Set Description for Work.NewHires

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>EmpID</td>
<td>num</td>
<td>8</td>
</tr>
<tr>
<td>Name</td>
<td>char</td>
<td>20</td>
</tr>
<tr>
<td>Extension</td>
<td>num</td>
<td>8</td>
</tr>
</tbody>
</table>

a) Missing values are assigned to Room for the observations that are read in from Work.NewHires.
b) Missing values are assigned to Room for all of the observations in Work.Employees.
c) Room is dropped from Work.Employees.
d) The values of Name are truncated in the observations that are read in from Work.NewHires.

Correct answer: a

PROC APPEND reads only the data in the DATA= SAS data set, not the BASE= SAS data set. When the BASE= data set contains more variables than the DATA= data set, missing values for the additional variables are assigned to the observations that are read in from the DATA= data set.

10 You do not need to use the FORCE option with PROC APPEND when

a) the DATA= data set contains variables that are not in the BASE= data set.
b) the BASE= data set contains variables that are not in the DATA= data set.
c) the variables in the DATA= data set are longer than the corresponding variables in the BASE= data set.
d) the variables in the DATA= data set have a different type than the corresponding variables in the BASE= data set.

Correct answer: b

The FORCE option does not need to be used if the BASE= data set contains variables that are not in the DATA= data set. The FORCE option must be used if

- the DATA= data set contains variables that are not in the BASE= data set
- the variables in the DATA= data set are longer than the corresponding variables in the BASE= data set
- the variables in the DATA= data set have a different type than the corresponding variables in the BASE= data set.
Chapter 15: Combining Data Horizontally

1. According to the data set descriptions below, which of the variables listed qualify as BY variables for a DATA step match-merge?

<table>
<thead>
<tr>
<th>Variable Type Length Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code char 5 Department code</td>
</tr>
<tr>
<td>Totemps num 3 Total number of employees</td>
</tr>
<tr>
<td>Region char 4 Location of the department</td>
</tr>
<tr>
<td>Manager num 5 Employee ID number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable Type Length Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDnum num 5 Employee ID number</td>
</tr>
<tr>
<td>Name char 20 Employee name</td>
</tr>
<tr>
<td>Division char 3 Division abbreviation</td>
</tr>
<tr>
<td>Hiredate num 8 Date of hire</td>
</tr>
<tr>
<td>Supervisor char 20 Name of supervisor</td>
</tr>
</tbody>
</table>

   a Code and IDnum
   b Manager and Supervisor
   c Manager and IDnum
   d There are no variables that are common to both of these data sets.

Correct answer: c

Remember that common variables might not have the same names. Manager and IDnum are the only two variables listed that match according to type and description. You can use the RENAME= option to rename one of these variables so that they can be used as BY variables in the MERGE statement of the DATA step.

2. Suppose you want to merge Dataset1, Dataset2, and Dataset3. Also suppose that Dataset1 and Dataset2 have the common variable Startdate, Dataset2 and Dataset3 have the common variable Instructor, and that these data sets have no other common variables. How can you use a DATA step to merge these three data sets into one new data set?

   a You use a MERGE statement in one DATA step to merge Dataset1, Dataset2, and Dataset3 by Startdate and Instructor.
   b You sort Dataset1 and Dataset2 by Startdate and merge them into a temporary data set in a DATA step. Then you sort the temporary data set and Dataset3 by Instructor and merge them into a new data set in a DATA step.
   c You can merge these data sets only with a PROC SQL step.
   d You cannot merge these three data sets at all because they do not have a common variable.

Correct answer: b

In order to merge multiple data sets in a DATA step, the data sets must have a common variable. However, if there are variables that are common to at least two of the input data sets, and if each input data set contains at least one of these
variables, then you can use subsequent DATA steps to merge the data sets. You can also use a PROC SQL step to merge data sets that do not have common variables.

3 Which of the following programs correctly creates a table with the results of a PROC SQL inner join matched on the values of `empcode`?

   a proc sql;
      select newsals.empcode allemps.lastname
       newsals.salary contrib.amount
      from sasuser.allemps, sasuser.contrib,
       sasuser.newsals
      where empcode=allemps.empid
      and empcode=contrib.empid;
      quit;

   b proc sql;
      create table usesql as
      select newsals.empcode allemps.lastname
       newsals.salsry contrib.amount
      from sasuser.allemps, sasuser.contrib,
       sasuser.newsals
      quit;

   c proc sql;
      create table usesql as;
      select newsals.empcode, allemps.lastname,
       newsals.salary, contrib.amount;
      from sasuser.allemps, sasuser.contrib,
       sasuser.newsals;
      where empcode=allemps.empid
      and empcode=contrib.empid;
      quit;

   d proc sql;
      create table usesql as
      select newsals.empcode, allemps.lastname,
       newsals.salary, contrib.amount
      from sasuser.allemps, sasuser.contrib,
       sasuser.newsals
      where empcode=allemps.empid
      and empcode=contrib.empid;
      quit;

   Correct answer: d

4 You can use PROC SQL to join data from data sets that do not have a single common variable among them. If you create a new table with the result of an inner join in a PROC SQL step, the resulting data set can be similar or identical to the result of a DATA step match-merge.

To process a default DATA step match-merge, SAS first reads the descriptor portion of each data set and sets up the PDV and the descriptor portion of the new data set. Which of the following accurately describes the rest of this process?

   a Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. When the BY value changes in all the input data sets, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are written to the new data set.
b Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. After each DATA step iteration, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are omitted from the new data set.

c Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. Then SAS goes through the new data set and eliminates all observations that do not have matching values of the BY variable.

d Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. The new data set is then ordered by values of the BY variable.

Correct answer: a

In a DATA step match-merge, SAS reads observations from the input data sets sequentially and match-merges them with observations from other input data sets. Combined observations are created when SAS writes values from all input data sets to the variables in the PDV. These observations, as well as any observations that contain missing or nonmatched values, are then written to the new data set. A PROC SQL join creates a Cartesian product of matches and then eliminates nonmatching data.

5 Which of the following statements is false about using multiple SET statements in one DATA step?

a You can use multiple SET statements to combine observations from several SAS data sets.

b Processing stops when SAS encounters the end-of-file (EOF) marker on either data set (even if there is more data in the other data set).

c You can use multiple SET statements in one DATA step only if the data sets in each SET statement have a common variable.

d The variables in the PDV are not reinitialized when a second SET statement is executed.

Correct answer: c

You can use multiple SET statements in one DATA step to combine observations from several data sets, and the data sets do not need to have a common variable. When you use multiple SET statements, you need to keep in mind the process that SAS uses to combine data from the input data sets. Otherwise, you might achieve unexpected results.

6 Select the program that correctly creates a new data set named Sasuser.Summary that contains one observation with summary data created from the Salary variable of the Sasuser.Empdata data set.

a proc sum data=sasuser.empdata noprint;
   output out=sasuser.summary sum=Salarysum;
run;

b proc means data=sasuser.empdata noprint;
   var salary;
   output out=sasuser.summary sum=Salarysum;
run;

c proc sum data=sasuser.empdata noprint;
   var salary;
   output out=sasuser.summary sum=Salarysum;
run;

d proc means data=sasuser.empdata noprint;
   output=sasuser.summary sum=Salarysum;
Correct answer: b

You can use the MEANS procedure to create a new data set that contains a summary statistic. You use the NOPRINT option to suppress the default report and the OUTPUT statement to route the results from the MEANS procedure to a new data set. You use the VAR statement to focus the procedure on one or more specific variables from the input data set.

7 If the value of **Cargosum** is $1000 at the end of the first iteration of the DATA step shown below, what is the value of **Cargosum** in the PDV when the DATA step is in its third iteration?

```
data sasuser.percent1;
  if _n_=1 then set sasuser.summary (keep=cargosum);
  set sasuser.monthsum (keep=salemon revcargo);
PctRev=revcargo/cargosum;
run;
```

- a $1000
- b $3000
- c The value is missing.
- d The value cannot be determined without seeing the data that is in Sasuser.Summary.

Correct answer: a

The **_N_** variable records how many times the DATA step has iterated. In the example shown above, **_N_** is used to ensure that only the first observation is read from Sasuser.Summary. Since the values in the PDV are not reinitialized after each DATA step iteration, this value will be retained as long as the DATA step continues to iterate. Therefore, if the value of **Cargosum** is $1000 in the first iteration, it will be $1000 in each subsequent iteration as well.

8 According to the data set shown, what is the value of **Totalrev** in the PDV at the end of the fourth iteration of the DATA step?

```
data sasuser.percent2(drop=totalrev);
  if _n_=1 then do until(lastobs);
    set sasuser.monthsum2(keep=revcargo) end=lastobs;
    totalrev+revcargo;
  end;
  set sasuser.monthsum2 (keep=salemon revcargo);
PctRev=revcargo/totalrev;
run;
```

- a The value is missing.
- b $350.00
- c $520.00
- d $1100.00

Correct answer: d

**Totalrev** is the accumulator variable of the sum statement, which is automatically initialized with a value of 0. If the expression in a sum statement produces a missing value, SAS replaces the missing value with a value of 0. As the DATA step iterates, the sum statement retains the accumulator variable so that it will accumulate a total.

9 Which of the following programs correctly uses an index to combine data from two input data sets?
a data work.profit;
   set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo)
      key=flightdate;
   set sasuser.dnunder;
   Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;

b data work.profit;
   set sasuser.dnunder;
   set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo)
      key=flightdate;
   where routeid='0000103';
   Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;

c data work.profit;
   set sasuser.dnunder;
   set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo);
      key=flightdate;
   Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;

d data work.profit;
   set sasuser.dnunder;
   set sasuser.sale2000(keep=routeid flightid date rev1st revbusiness revecon revcargo)
      key=flightdate;
   Profit=sum(rev1st, revbusiness, revecon, revcargo, -expenses);
run;

Correct answer: d

You use the KEY= option in a SET statement to cause SAS to use an index to combine data from multiple data sets. When the SET statement with the KEY= option executes, the program data vector must already contain a value for the indexed variable. You cannot use WHERE processing on a data set that has been read with the KEY= option within the same DATA step.

10 Which of the following statements about the _IORC_ variable is false?

a It is automatically created when you use either a SET statement with the KEY= option or the MODIFY statement with the KEY= option in a DATA step.

b A value of zero for _IORC_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) did not execute successfully.

c A value of zero for _IORC_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) executed successfully.

d You can use the _IORC_ variable to prevent nonmatching data from being included when you use an index to combine data from multiple data sets.

Correct answer: b
When you use the KEY= option, SAS creates an automatic variable named _IORC_, which stands for INPUT/OUTPUT Return Code. If the value of _IORC_ is zero, the index search was successful. The _IORC_ variable is also created automatically when you use a MODIFY statement in a DATA step.

Chapter 16: Using Lookup Tables to Match Data

1 Which SAS statement correctly specifies the array Sales as illustrated in the following table?

Table Representation of Sales Array

<table>
<thead>
<tr>
<th></th>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
</tr>
</thead>
<tbody>
<tr>
<td>m5</td>
<td>m6</td>
<td>m7</td>
<td>m8</td>
<td></td>
</tr>
<tr>
<td>m9</td>
<td>m10</td>
<td>m11</td>
<td>m12</td>
<td></td>
</tr>
</tbody>
</table>

- a array Sales{3,4} m1-m12;
- b array Sales{4,3} m1-m12;
- c array {3,4} Sales m1-m12;
- d array {4,12} Sales m1-m12;

Correct answer: a

An array is specified using the keyword ARRAY followed by the name of the array and the dimensions of the array. In a two-dimensional array, the two dimensions can be thought of as a table of rows and columns. The first dimension in the ARRAY statement specifies the number of rows. The second dimension specifies the number of columns.

2 Which of the following statements creates temporary array elements?

- a array new (*) _temporary_;
- b array new {6} _temporary_;
- c array new (*) _temporary_ Jan Feb Mar Apr May Jun;
- d array _temporary_ new {6} Jan Feb Mar Apr May Jun;

Correct answer: b

To create temporary array elements, specify the keyword _TEMPORARY_ after the array name and dimension. Remember that if you use an asterisk to count the array elements, you must list the array elements. You cannot use the asterisk and the _TEMPORARY_ keyword together in an ARRAY statement.

3 Which DO statement processes all of the elements in the Yearx array?

- a array Yearx(12) Jan--Dec;
a do i=1 to dim(yearx);
b do i=1 to 12;
c do i=Jan to Dec;
d a and b
Correct answer: d

To process all of the elements in an array, you can use either the DIM function with the array name as the argument or specify the array dimension.

4 Given the following program, what is the value of Points in the fifth observation in the data set Work.Results?

```sas
data work.results;
array score{2,4} _temporary_;
  (40,50,60,70,40,50,60,70);
set work.contest;
Points=score{week,finish};
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Week</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Triffle</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Gomez</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Chapman</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Venter</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Vandeusen</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Jittle</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Venter</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Gomez</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

a 40
b 50
c 60
d 70
Correct answer: a

The ARRAY statement creates the two-dimensional array score and specifies the dimensions of the array: two rows and four columns. The value of Points for each observation is determined by referencing the array based on the values of Week and Finish in the Work.Contest data set. The row number for the array reference is determined by the value of Week. The column number for the array reference is determined by the value of Finish.

5 Array values should be stored in a SAS data set when
a there are too many values to initialize easily in an array.
b the values change frequently.
c the same values are used in many programs.
d all of the above
Correct answer: d

Array values should be stored in a SAS data set when there are too many values to initialize easily in an array, the values change frequently, or the same values are used in many programs.

6 Given the following program, which statement is not true?

```sas
data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.ctargets;
```

array Mnth{*} Jan-Dec;
do j=1 to dim(mnth);
    targets(year,j)=mnth{j};
end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
run;

a The IF-THEN statement specifies that the Targets array is loaded once.
b During the first iteration of the DATA step, the outer DO loop executes three times.
c After the first iteration of the DO loop, the pointer drops down to the second SET statement.
d During the second iteration of the DATA step, the condition _N_=1 is false. So, the DO loop doesn’t execute.

Correct answer: c

The IF-THEN statement specifies that the Targets array is loaded only once, during the first iteration of the DATA step. During the first iteration of the DATA step, the condition _N_=1 is true, so the outer DO loop executes three times; once for each observation in Sasuser.Ctargrt. After the third iteration of the DO loop, the pointer drops down to the second SET statement and the values from the first observation in Sasuser.Monthum are read into the program data vector. During the second iteration of the DATA step, the condition _N_=1 is false. So, the DO loop doesn’t execute.

7 Given the following program, which variable names will appear in the data set Work.New?

SAS Data Set Work.Revenue

<table>
<thead>
<tr>
<th>Obs</th>
<th>Year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>1.003561</td>
<td>922.080</td>
<td>836.068</td>
<td>973.016</td>
</tr>
<tr>
<td>2</td>
<td>2001</td>
<td>1.018.552</td>
<td>1.013.908</td>
<td>1.047.812</td>
<td>1.005.578</td>
</tr>
<tr>
<td>3</td>
<td>2002</td>
<td>1.182.442</td>
<td>1.657.323</td>
<td>1.079.866</td>
<td>1.466.640</td>
</tr>
</tbody>
</table>

a Year, Jan, Feb, Mar, Apr
b Year, 2000, 2001, 2002
c _NAME_, Col1, Col2, Col3
d _NAME_, Jan, Feb, Mar, Apr

Correct answer: c

The TRANSPOSE procedure creates an output data set by restructuring the values in a SAS data set. When the data set is restructured, selected variables are transposed into observations. The procedure creates several variable names by default. _NAME_ is the default name of the variable that PROC TRANSPOSE creates to identify the source of the values in each observation in the output data set. The remaining transposed variables are named COL1…COLn by default.
8 Which program creates the output data set `Work.Temp2`?

SAS Data Set Work.Temp

<table>
<thead>
<tr>
<th>Obs</th>
<th>Month1</th>
<th>Month2</th>
<th>Month3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13604250</td>
<td>24901752</td>
<td>18166825</td>
</tr>
<tr>
<td>2</td>
<td>72551855</td>
<td>23048835</td>
<td>17804326</td>
</tr>
<tr>
<td>3</td>
<td>12540156</td>
<td>27572140</td>
<td>14152851</td>
</tr>
<tr>
<td>4</td>
<td>11241374</td>
<td>28684655</td>
<td>15156575</td>
</tr>
</tbody>
</table>

SAS Data Set Work.Temp2

<table>
<thead>
<tr>
<th>Obs</th>
<th>Month</th>
<th>Quarter1</th>
<th>Quarter2</th>
<th>Quarter3</th>
<th>Quarter4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13604250</td>
<td>72551855</td>
<td>12540156</td>
<td>11241374</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>24901752</td>
<td>23048835</td>
<td>27572140</td>
<td>28684655</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18166825</td>
<td>17804326</td>
<td>14152851</td>
<td>15156575</td>
<td></td>
</tr>
</tbody>
</table>

- a proc transpose data=work.temp
  
  `out=work.temp2`  
  `prefix=Quarter;`
  
  `run;`

- b proc transpose data=work.temp
  
  `out=work.temp2`  
  `name=Month`  
  `prefix=Quarter;`
  
  `run;`

- c proc transpose data=work.temp
  
  `out=work.temp2`  
  `prefix=Month`  
  `name=Quarter;`
  
  `run`

- d proc transpose data=work.temp
  
  `out=work.temp2`  
  `prefix=Month`  
  `index=Quarter;`
  
  `run;`

Correct answer: b

You can use several options with PROC TRANSPOSE to give the variables in the output data set descriptive names. The NAME= option specifies a name for `_NAME_` variable. The PREFIX= option specifies a prefix to use in constructing names for the other variables in the output data set.

9 Which version of the data set `Work.Sales2` is created by the following program?

SAS Data Set Work.Sales

```
proc transpose data=work.sales
  out=work.sales2
  name=Week;
  by employee;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Employee</th>
<th>Week1</th>
<th>Week2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Almers</td>
<td>3393.50</td>
<td>2192.25</td>
</tr>
<tr>
<td>2</td>
<td>Bonaventure</td>
<td>5093.75</td>
<td>2247.50</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>1813.30</td>
<td>2082.75</td>
</tr>
<tr>
<td>4</td>
<td>LAmance</td>
<td>1572.50</td>
<td>2960.00</td>
</tr>
</tbody>
</table>
A BY statement can be used with PROC TRANSPOSE. For each BY group, PROC TRANSPOSE creates one observation for each variable that it transposes. The BY variable itself is not transposed. The original data set must be sorted or indexed prior to using a BY statement with PROC TRANSPOSE.
10 Which program creates the data set Work.Fishsize?

SAS Data Set Work.Fishdata

<table>
<thead>
<tr>
<th>Obs</th>
<th>Location</th>
<th>Date</th>
<th>Length1</th>
<th>Weight1</th>
<th>Length2</th>
<th>Weight2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>31</td>
<td>0.25</td>
<td>32</td>
<td>0.30</td>
</tr>
<tr>
<td>2</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>29</td>
<td>0.23</td>
<td>30</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>32</td>
<td>0.35</td>
<td>32</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>33</td>
<td>0.30</td>
<td>33</td>
<td>0.28</td>
</tr>
</tbody>
</table>

SAS Data Set Work.Fishsize

<table>
<thead>
<tr>
<th>Obs</th>
<th>Location</th>
<th>Date</th>
<th>NAME</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Length1</td>
<td>31.00</td>
</tr>
<tr>
<td>2</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Weight1</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Length2</td>
<td>32.00</td>
</tr>
<tr>
<td>4</td>
<td>Cole Pond</td>
<td>02JUN95</td>
<td>Weight2</td>
<td>0.30</td>
</tr>
<tr>
<td>5</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Length1</td>
<td>29.00</td>
</tr>
<tr>
<td>6</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Weight1</td>
<td>0.23</td>
</tr>
<tr>
<td>7</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Length2</td>
<td>30.00</td>
</tr>
<tr>
<td>8</td>
<td>Cole Pond</td>
<td>04AUG95</td>
<td>Weight2</td>
<td>0.25</td>
</tr>
<tr>
<td>9</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Length1</td>
<td>32.00</td>
</tr>
<tr>
<td>10</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Weight1</td>
<td>0.35</td>
</tr>
<tr>
<td>11</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Length2</td>
<td>32.00</td>
</tr>
<tr>
<td>12</td>
<td>Eagle Lake</td>
<td>02JUN95</td>
<td>Weight2</td>
<td>0.25</td>
</tr>
<tr>
<td>13</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Length1</td>
<td>33.00</td>
</tr>
<tr>
<td>14</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Weight1</td>
<td>0.30</td>
</tr>
<tr>
<td>15</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Length2</td>
<td>33.00</td>
</tr>
<tr>
<td>16</td>
<td>Eagle Lake</td>
<td>04AUG95</td>
<td>Weight2</td>
<td>0.28</td>
</tr>
</tbody>
</table>

a proc transpose data=work.fishdata
   out=work.fishsize
   prefix=Measurement;
run;

b proc transpose data=work.fishdata
   out=work.fishsize
   prefix=Measurement;
   by location;
run;

c proc transpose data=work.fishdata
   out=work.fishsize
   prefix=Measurement;
   by date;
run;

d proc transpose data=work.fishdata
   out=work.fishsize
   prefix=Measurement;
   by location date;
run;
Correct answer: d

The observations in Work.Fishsize are grouped by Location and Date. For each BY group, PROC TRANSPOSE creates four observations, one for each variable (Length1, Weight1, Length2, and Weight2) that it is transposing.

Chapter 17: Formatting Data

1 Which SAS system option is used to identify format catalogs to SAS?
   a FMTERR
   b FMTLIB
   c NOFMTERR
   d FMTSEARCH=
   Correct answer: d

   By default, SAS searches for custom formats in the Work and Library libraries. The FMTSEARCH= system option specifies other catalogs to search when a format is referenced.

2 Given the following PROC FORMAT step, how is the value 70 displayed when the AGEGRP. format is applied?
   proc format;
     picture agegrp
       1-<13='00 Youth'
       13-<20='00 Teen'
       20-<70='00 Adult'
       70-high='000 Senior';
   run;
   a 000 Senior
   b 70 Adult
   c 70 Senior
   d 070 Senior
   Correct answer: c

   A non-inclusive range is used such that the age at the high end of the range is not included. To create the picture format, three zeros are used to create a position for a three-digit numeric value. Because zero is used as a digit selector rather than a nonzero value, leading zeros are not included in the formatted value.

3 When the NOFMTERR system option is in effect, what happens when SAS encounters a format it cannot locate?
   a Creates the format in the default Work.Formats directory and continues processing.
   b Substitutes the $w. or w. format and continues processing.
   c Stops processing and writes an error message to the log.
   d Skips processing at that step and continues with the next step and writes a note to the log.
   Correct answer: b

   By default, FMTERR is in effect and SAS stops processing if it cannot find a format that is referenced. When NOFMTERR is in effect, SAS substitutes the $w. or w. format and continues processing.

4 Which of the following variables must be in the data set that is specified on the CNTLIN= option?
Quiz Answer Keys △ Chapter 17: Formatting Data 953

a  End
b  FmtName
c  Value
d  Description

Correct answer: b

A data set that is used to create a format with the CNTLIN= option must have the variables **FmtName**, **Start**, and **Label**. If a range is specified, it must also include the variable **End**.

5 Given the following code, what option is missing?

```
proc format;
  value times (?)
    '00:00't-'04:59't = 'Red Eye'
    '05:00't-'11:59't = 'Morning'
    '12:00't-'17:59't = 'Afternoon'
    '18:00't-'23:59't = 'Evening'
    '00:00't-'11:59't = 'AM'
    '12:00't-'23:59't = 'PM';
run;
```

a  MULTILABEL
b  MULTIRANGE
c  MLF
d  MULTIFORMAT

Correct answer: a

The format created by this value statement has overlapping ranges, so the MULTILABEL option must be used. A multilabel format can be used by any procedure that supports the MLF option.

6 Which PROC FORMAT option is used to create a SAS data set from a format?

a  CNTLIN=
b  LIB=
c  CNTLOUT=
d  FMTLIB

Correct answer: c

The CNTLOUT= option is used to create a SAS data set from a format.

7 Given the following OPTIONS statement, in what order will SAS search to find a user-defined format?

```
options fmtsearch=(work abc.newfmt sasuser);
```

c  Work.Formats ▶ Abc.Newfmt ▶ Sasuser.Format
d  the default search order

Correct answer: b

SAS will search in the order specified on the FMTSEARCH= option. By default, SAS searches in the Work and Library libraries first unless they are specified on the option. Because Library is not specified here, it is searched after Work.

8 What option is used with PROC FORMAT to document the formats in a particular format catalog?

a  FMTSEARCH
b FMTERR
C CATALOG
d FMTLIB
Correct answer: d

The FMTLIB keyword is used to document the formats in a catalog. You can use the SELECT and EXCLUDE statements to process specific formats rather than the entire catalog.

9 Which set of statements would you add to the PROC CATALOG code to copy the LEVELS. and $PICKS. formats from the Sasuser.Formats catalog to the Work.Forms catalog?

```plaintext
proc catalog cat=sasuser.formats;
   copy out=work.formats;
   select levels.format $picks.format;
run;
```

Correct answer: c

In the COPY statement, OUT= specifies the catalog to which you want to copy the format catalog entry. In the SELECT statement you specify the catalog entries by their entire name. Remember that numeric formats are stored with the extension .FORMAT and character formats are stored with the extension .FORMATC.

10 Given the following PROC FORMAT step, how is the value 6.1 displayed when the SKICOND format is applied?

```plaintext
proc format;
   value skicond
      0-3='Poor'
      3<-6='Fair'
      6<-9='Good'
      9<-high='Excellent';
run;
```

a 6.1
b Fair
c Good
d 
Correct answer: c

The value 6.1 falls in the range 6<-9, which is labeled ‘Good.’ The non-inclusive range does not include the value 6, but it does include everything above 6.
1. Which type of integrity constraint would you place on the variable **StoreID** to ensure that there are no missing values and that there are no duplicate values?
   
   a. UNIQUE  
   b. CHECK  
   c. PRIMARY KEY  
   d. NOT NULL  
   Correct answer: c  
   The PRIMARY KEY integrity constraint includes both the NOT NULL and UNIQUE constraints.

2. Which code creates an audit trail on the SAS data set *Reports.Quarter1*?
   
   a. proc datasets nolist;  
      audit quarter1;  
      initiate;  
      quit;  
   b. proc datasets lib=reports nolist;  
      audit initiate reports.quarter1;  
      quit;  
   c. proc datasets lib=reports nolist;  
      initiate audit quarter1;  
      quit;  
   d. proc datasets lib=reports nolist;  
      audit quarter1;  
      initiate;  
      quit;  
   Correct answer: d  
   To initiate an audit on an existing SAS data set with the DATASETS procedure, you specify the data set in the AUDIT statement, and then you specify the INITIATE statement. You specify the library with the LIB= option.

3. Which DATA step uses the transaction data set *Records.Overnight* to update the master data set *Records.Snowfall* by **accumAmt**?
   
   a. data records.snowfall;  
      modify records.snowfall records.overnight  
      key=accumAmt;  
      run;  
   b. data records.snowfall;  
      modify records.overnight records.snowfall;  
      by accumAmt;  
      run;  
   c. data records.snowfall;  
      modify records.snowfall records.overnight;  
      by accumAmt;  
      run;  
   d. data records.snowfall;  
      modify records.snowfall records.overnight;  
      update accumAmt;  
      run;  
   Correct answer: c  
   In the MODIFY statement, you specify the master data set followed by the transaction data set. Then you specify the variable in the BY statement.
4 The automatic variable _IORC_ is created when you use the MODIFY statement with a BY statement or the KEY= option. How can you use the value of _IORC_?
   a to determine whether the index specified on the KEY= option is a valid index
   b to determine the number of observations that were updated in the master data set
   c to determine the status of the I/O operation
   d to determine the number of observations that could not be updated in the master data set
   Correct answer: c
   
   The value of _IORC_ is a numeric return code that indicates the status of the most recently executed I/O operation. Checking the value of this variable allows you to detect abnormal I/O conditions and direct execution in particular ways.

5 Which PROC DATASETS step creates an integrity constraint named val_age on the data set Survey to ensure that values of the variable age are greater than or equal to 18?
   a proc datasets nolist;
      modify age;
      ic create val_age=check(where=(age>=18));
      quit;
   b proc datasets nolist;
      modify Survey;
      ic create val_age=check(age>=18);
      quit;
   c proc datasets nolist;
      modify survey;
      integrity constraint
         val_age=check(where=(age>=18));
      quit;
   d proc datasets nolist;
      modify survey;
      ic create val_age=check(where=(age>=18));
      quit;
   Correct answer: d
   
   In the MODIFY statement, you list the SAS data set that you want to modify. Then you use the IC CREATE statement to create the integrity constraint. This integrity constraint is a CHECK constraint and you use a WHERE clause to specify the condition that the variable values must meet.

6 Which statement about using the MODIFY statement in a DATA step is true?
   a MODIFY creates a second copy of the data while variables in the data are being matched with a WHERE clause and then deletes the second copy.
   b You cannot modify the descriptor portion of the data set using the MODIFY statement.
   c You can use the MODIFY statement to change the name of a variable.
   d If the system terminates abnormally while a DATA step that is using the WHERE statement is processing, SAS automatically saves a copy of the unaltered data set.
   Correct answer: b
   
   The MODIFY statement in a DATA step can only be used to modify the values in a data set. It cannot be used to modify the descriptor portion of the data set.

7 Which of the following statements about audit trails is true?
a. They create historical versions of data so that a copy of the data set is saved each time the data is replaced.
b. They record information about changes to observations in a data set each time the data set is replaced.
c. They record information about changes to observations in a data set each time the data is modified in place.
d. The audit trail file has the same name as the SAS data file it is monitoring, but has #AUDIT at the end of the data set name.

Correct answer: c

Audit trails are used to track changes that are made to a data set in place.

8. Which code initiates generation data sets on the existing SAS data set *Sasuser.Amounts* and specifies that five historical versions are saved in addition to the base version?

a. `proc datasets lib=sasuser nolist;
modify Amounts (genmax=6);
quit;`
b. `proc datasets lib=sasuser nolist;
modify Amounts (genmax=5);
quit;`
c. `proc datasets lib=sasuser nolist;
modify Amounts (gennum=6);
quit;`
d. `proc datasets lib=sasuser nolist;
modify Amounts (gennum=5);
quit;`

Correct answer: a

You use the DATASETS procedure and the MODIFY statement to specify a number of generation data sets for a data set. The GENMAX= option is used to specify the number of versions to save. The number you specify includes the base version.

9. Which statement about using the KEY= option in the MODIFY statement is true?

a. SAS locates the variables to update using the index specified in the KEY= option and then automatically overlays nonmissing transaction values as it does when you use the MODIFY/BY statements.
b. When you use the KEY= option, you must explicitly state the update that you want to make. SAS does not automatically overlay nonmissing transaction values.
c. The KEY= option is used to specify a variable to match for updating observations.
d. The index named in the KEY= option must be a simple index.

Correct answer: b

When you use the KEY= option, you must specify the update that you want to make to the data set.

10. Which code deletes all generations of the data set *Sasuser.Amounts* including the base data set?

a. `proc datasets lib=sasuser nolist;
delete amounts (gennum=ALL);
quit;`
b proc datasets lib=sasuser nolist;
   delete amounts (gennum=HIST);
quit;

c proc datasets lib=sasuser nolist;
   delete amounts (gennum=0);
quit;
d proc datasets lib=sasuser nolist;
   delete amounts;
quit;

Correct answer: a

The keyword ALL is used to indicate that you want to delete all generations of the specified data set including the base version. The keyword HIST deletes the generation data sets, but saves the base version.

---

**Chapter 19: Introduction to Efficient SAS Programming**

This chapter has no quiz.

**Chapter 20: Controlling Memory Usage**

1 Which of the following statements is true regarding the BUFNO= option?
   a The BUFNO= option specifies the size of each buffer that is used for reading or writing a SAS data set.
   b The BUFNO= option can improve execution time by limiting the number of input/output operations that are required.
   c Using the BUFNO= option results in permanent changes to the data set.
   d Using the BUFNO= option to increase the number of buffers results in decreased memory consumption.

Correct answer: b

You can use the BUFNO= system option or data set option to control how many buffers are available for reading or writing a SAS data set. Using BUFNO= can improve execution time by limiting the number of input/output operations that are required for a particular SAS data set. However, the improvement in I/O comes at the cost of increased memory consumption. The buffer number is not a permanent attribute of the data set and is valid only for the current step or SAS session.

2 Which of the following statements is not true regarding a page?
   a A page is the unit of data transfer between the engine and the operating environment.
   b A page includes the number of bytes used by the descriptor portion, the data values, and the overhead.
   c The size of a page is analogous to buffer size.
   d The size of a page can be changed at any time.

Correct answer: d

A page is fixed in size when the data set is created, either to a default value or a specified value. You can use the BUFSIZE= option to control the page size of an output SAS data set. BUFSIZE= specifies not only the page size (in bytes), but
also the size of each buffer that is used for reading or writing the SAS data set. The new buffer size is permanent; after it is specified, it is used whenever the data set is processed.

3 The total number of bytes occupied by a data set equals...?
   a the page size multiplied by the number of pages.
   b the page size multiplied by the number of observations.
   c the sum of the page size and the number of pages.
   d the number of pages multiplied by the number of variables.

   Correct answer: a
   The total number of bytes occupied by a data set equals the page size multiplied by the number of pages. You can use the CONTENTS procedure to report the page size and the number of pages.

4 Which statement opens the file Work.Quarter1, allocates enough buffers to hold the entire file in memory, and reads the data into memory?
   a sasfile work.quarter1 open;
   b sasfile work.quarter1 load;
   c sasfile work.quarter1 bufno=max;
   d sasfile work.quarter1 bufsize=max;

   Correct answer: b
   The SASFILE LOAD statement opens the file, allocates the buffers, and reads the data into memory.

5 Which of the following statements is true regarding a file that is opened with the SASFILE statement?
   a The file is available to subsequent DATA and PROC steps or applications until a SASFILE CLOSE statement is executed or until the program ends.
   b The file is available to subsequent DATA and PROC steps or applications until a SASFILE END statement is executed.
   c The file is available for subsequent utility or output processing until the program ends.
   d If the file increases in size during processing, the number of buffers remains the same.

   Correct answer: a
   When a SAS data file is opened using the SASFILE statement, the data is held in memory, and is available to subsequent DATA and PROC steps or applications, until either a SASFILE CLOSE statement is executed or the program ends. Though a file that is opened with the SASFILE statement can be used for subsequent input or update processing, it cannot be used for subsequent utility or output processing. If the file in memory increases in size during processing, the number of buffers also increases.

---

Chapter 21: Controlling Data Storage Space

1 Which of the following statements about uncompressed SAS data files is true?
   a The descriptor portion is stored on whatever page has enough room for it.
   b New observations are always added in the first sufficient available space.
   c Deleted observation space is tracked.
d  New observations are always added at the end of the data set.
Correct answer: d

The descriptor portion of an uncompressed data file is always stored at the end of the first data set page. New observations are always added to the end of the data set, and deleted observation space is neither tracked nor reused.

2  Which of the following statements about compressed SAS data files is true?
   a  The descriptor portion is stored on whatever data set page has enough room for it.
   b  Deleted observation spaced can be reused.
   c  Compressed SAS data files have a smaller overhead than uncompressed SAS data files.
   d  In a compressed SAS data set, each observation must be the same size.
Correct answer: b

The descriptor portion of a compressed data file is always stored at the end of the first data set page. If you specify REUSE=YES, SAS tracks and reuses deleted observation space within a compressed data file. Therefore, every observation in a compressed data file can be a different size. Compressed data files do have a larger overhead than uncompressed data files.

3  Which of the following programs correctly creates reduced-length numeric variables?
   a  data temp;
      infile file1;
      input x 4.
         y 3.
         z 2.;
      run;
   b  data temp;
      format x 4.
         y 3.
         z 2.;
      infile file1;
      input x 4.
         y 3.
         z 2.;
      run;
   c  data temp;
      length x 4
         y 3
         z 2;
      infile file1;
      input x 4.
         y 3.
         z 2.;
      run;
   d  data temp;
      informat x 4.
         y 3.
         z 2.;
      infile file1;
      input x 4.
         y 3.
You use the LENGTH statement to assign a reduced length to a numeric variable. If you do not use the LENGTH statement to define a reduced length for numeric variables, their default length is 8 bytes. The FORMAT statement associates a format with a variable, and the INFORMAT statement associates an informat with a variable.

4 Which of the following statements about SAS data views is true?
   a) SAS data views use less disk space but more CPU resources than SAS data files.
   b) SAS data views can be created only in permanent SAS data libraries.
   c) SAS data views use less CPU resources but more disk space than SAS data files.
   d) SAS data views can be created only in temporary SAS data libraries.
   Correct answer: a

   SAS data views use significantly less disk space than SAS data files. However, SAS data views typically need more CPU resources than SAS data files. You can create a SAS data view in either the temporary SAS data library or in a permanent SAS data library.

5 Which of the following programs should you use to detect any loss of precision between the default-length numeric variables in Company.Regular and the reduced-length numeric variables in the data set Company.Reduced?
   a) proc contents data=company.regular;
      compare data=company.reduced;
      run;
   b) proc compare base=company.regular
      compare=company.reduced;
      run;
   c) proc print data=company.regular;
      run;
      proc print data=company.reduced;
      run;
   d) proc datasets library=company;
      contents data=regular compare=reduced;
      run;
   Correct answer: b

   You use the COMPARE procedure to detect any differences in the values of two data sets. The COMPARE statement is not valid syntax in either the CONTENTS procedure or the DATASETS procedure. Printing both data sets might not reveal differences in the precise values of the shortened variables, depending on the formats that are used.
1 Placing the subsetting IF statement at the top rather than near the bottom of a DATA step results in a savings in CPU usage. What happens if the subset is large rather than small?

   a The savings in CPU usage increases as the subset grows larger because the I/O increases.
   b The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement at the top of a DATA step always uses less resources than placing it at the bottom.
   c The savings in CPU usage remains constant as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set is preferable.
   d The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set increases the I/O.

Correct answer: b

As SAS processes a larger subset of the data, more CPU resources are required. However, positioning of the subsetting IF statement in a DATA step does affect performance and efficiency.

2 Which of the following statements is true about techniques that are used for modifying data and attributes?

   a You can use PROC DATASETS to modify both data values and variable attributes.
   b You can use PROC DATASETS to modify only data values.
   c You can use the DATA step to modify both data values and variable attributes.
   d You can use the DATA step to modify only variable attributes.

Correct answer: c

The DATA step is the only technique that can be used to modify both data values and variable attributes. The DATASETS procedure enables you to modify only variable attributes.

3 For selecting observations, is a subsetting IF statement or a WHERE statement more efficient? Why?

   a A subsetting IF statement is more efficient because it loads all observations sequentially into the program data vector.
   b A subsetting IF statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.
   c A WHERE statement is more efficient because it loads all observations sequentially into the program data vector.
   d A WHERE statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.

Correct answer: d

For selecting observations, a WHERE statement is more efficient than a subsetting IF statement because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.

4 When is it more advantageous to create a temporary SAS data set rather than a permanent SAS data set?

   a When the external file on which the data set is based might change between SAS sessions.
b When the external file on which the data set is based does not change between SAS sessions.
c When the data set is needed for more than one SAS session.
d When you are converting raw numeric values to SAS data values.

Correct answer: a

It is more advantageous to create a temporary SAS data set rather than a permanent SAS data set when the external file on which the data set is based is frequently updated between SAS sessions.

5 When you compare the technique of using multiple DATASETS procedures to using a single DATASETS procedure to modify the descriptor portion of a data set, which is true?

a A one-step DATASETS procedure results in an increase in I/O operations.
b Multiple DATASETS procedures result in a decrease in I/O operations.
c A one-step DATASETS procedure results in a decrease in CPU usage.
d Multiple DATASETS procedures result in a decrease in CPU usage.

Correct answer: c

A one-step DATASETS procedure results in a savings of CPU usage and I/O operations. PROC DATASETS supports RUN-group processing, which enables you to process multiple SAS data sets from the same library with one invocation of the procedure.

Chapter 23: Selecting Efficient Sorting Strategies

1 When the following program is submitted, what is the value of FIRST.Product_Line for the third observation in the data set Work.Products?

```sas
data new.products;
  set work.products
  by product_line notsorted;
run;
```

SAS Data Set Work.Products

<table>
<thead>
<tr>
<th>Obs</th>
<th>Product_Line</th>
<th>Product_Name</th>
<th>Supplier_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Children</td>
<td>Kids Children's Fleece Hat</td>
<td>3Top Sports</td>
</tr>
<tr>
<td>2</td>
<td>Children</td>
<td>Ski Jacket w/Removable Fleece</td>
<td>Scandinavian Clothing A/S</td>
</tr>
<tr>
<td>3</td>
<td>Clothes &amp; Shoes</td>
<td>Typhoon Linen Pants</td>
<td>Typhoon Clothing</td>
</tr>
<tr>
<td>4</td>
<td>Clothes &amp; Shoes</td>
<td>Watchit 120 Sterling/Reflective</td>
<td>Eclipse Inc</td>
</tr>
<tr>
<td>5</td>
<td>Clothes &amp; Shoes</td>
<td>Money Belt, Black</td>
<td>Top Sports</td>
</tr>
</tbody>
</table>

a 1
b 3
c 0
d Clothes & Shoes

Correct answer: a
First. is a temporary automatic variable that identifies the first observation in each BY group. When an observation is the first in a BY group, SAS sets the value of the FIRST. variable to 1. For all other observations in the BY group, the value of the FIRST. variable is 0.

2. Which option is used with the SORT procedure to store only the BY variables and the observation numbers in temporary files?
   a. NOTSORTED
   b. GROUPFORMAT
   c. TAGSORT
   d. SORTEDBY=
   Correct answer: c

   The TAGSORT option stores only the BY variables and the observation numbers in temporary files. The BY variables and the observation numbers are called tags. At the completion of the sorting process, PROC SORT uses the tags to retrieve records from the input data set in sorted order.

3. Which of the following is not an advantage of BY-group processing with an index that is based on the BY variables?
   a. The data can be sequenced without using the SORT procedure.
   b. There is no need to re-sort a data set when observations are modified or added.
   c. It is generally more efficient than reading a sorted data set sequentially.
   d. The data can be sequenced by different variables if multiple indexes are used.
   Correct answer: c

   When BY-group processing is used with an index that is based on the BY variables, the data can be sequenced without using the SORT procedure. The data can be sequenced by different variables if multiple indexes are used. Because indexes are updated automatically, there is no need to re-sort a data set when observations are modified or added. However, BY-group processing with an index is generally less efficient than reading a sorted data set sequentially.

4. Which SORT procedure option compares all of the variable values for each observation to those for the previous observation that was written to the output data set?
   a. NODUPKEY
   b. NODUPRECS
   c. EQUALS
   d. NOEQUALS
   Correct answer: b

   The NODUPRECS option compares all of the variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

5. What happens if the workspace that is required for completing a sort is less than or equal to the value that is specified in the SORTSIZE= system option or procedure option?
   a. CPU time is increased.
   b. I/O is increased.
   c. The entire sort can take place in memory.
   d. A temporary utility file is created in the Work directory or in a mainframe temporary area.
   Correct answer: c
The SORTSIZE= system option or procedure option specifies how much memory is available to the SORT procedure. If the required workspace is less than or equal to the value specified in the SORTSIZE= system option or procedure option, then the entire sort can take place in memory, which reduces processing time.

Chapter 24: Querying Data Efficiently

1 Why can using an index reduce the number of I/O operations that are required for accessing a small subset?
   a Using an index requires larger input buffers, which can hold more pages.
   b The index does not have to be loaded into an input buffer.
   c The number of observations that SAS has to load into the program data vector (PDV) is decreased.
   d The number of pages that SAS has to load into input buffers is decreased.
   Correct answer: d

   When using an index to select a subset, SAS loads only the pages that contain a qualified observation into input buffers. When accessing observations sequentially, SAS must load all observations into input buffers. Loading more pages requires more I/O operations.

2 You want to select a subset of observations in the data set Company.Products, and you have defined a simple index on the variable Rating. SAS cannot use the index to process which of the following WHERE statements?
   a where rating is missing;
   b where rating=int(rating);
   c where rating between 3.5 and 7.5;
   d where rating=5.5;
   Correct answer: b

   SAS considers using an index to process a WHERE condition that contains one of a specific group of operators and functions. However, SAS will not consider using an index for a WHERE condition that contains other elements, such as a function other than TRIM or SUBSTR.

3 In which of the following situations is sequential access likely to be more efficient than direct access for WHERE processing?
   a The subset contains over 75% of the observations in the data set.
   b The WHERE expression specifies both key variables in a single composite index.
   c The data is sorted on the key variable.
   d The data set is very large.
   Correct answer: a

   The size of the subset relative to the size of the data set is an important factor in determining which access method is most efficient. If a subset is large (more than 33% of the data set), it is likely to be more efficient to use sequential access than direct access. Direct access is usually more efficient when you are selecting a small subset (less than 33% of the data set), especially if the data set is large (has a high page count). However, if the data set is very small (less than three pages), using an index is not efficient. The number of key variables specified in a WHERE
expression does not determine which access method is most efficient. If the two key variables that are specified are the first two variables in the same index, the WHERE expression is a candidate for compound optimization. Sorting the data also does not determine which access method is most efficient. However, sorting the data before subsetting improves the efficiency of WHERE processing regardless of the access method.

4 You want to summarize data for one class variable, and you are trying to decide whether to use PROC MEANS (or PROC SUMMARY), PROC REPORT, PROC TABULATE, PROC SQL, or the DATA step with PROC SORT. Which of the following statements about the efficiency of these summarization tools is not true?

   a PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE have similar resource usage.
   b The efficiency of all these tools is affected by the shape of the data.
   c The SQL procedure is always the least efficient because it is a general-purpose tool.
   d PROC SQL and the DATA step with PROC SORT have similar resource usage.

Correct answer: c

When you are summarizing data for one or more class variables, the tools in each of the following groups are similar in resource usage:

- PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE
- PROC SQL and the DATA step with PROC SORT.

However, the relative efficiency of the two groups of tools varies according to the shape of the data.

5 Which of the following techniques is most efficient for summarizing data for specific combinations of class variables?

   a the NWAY option in multiple PROC MEANS steps
   b the TYPES statement in a PROC MEANS step
   c the WHERE= option in a PROC MEANS step
   d a basic PROC MEANS step

Correct answer: b

The TYPES statement in a PROC MEANS step is the most efficient technique for summarizing data for one or more specific combinations of class variables. A program that contains the NWAY option in multiple PROC MEANS steps is usually least efficient because SAS must read the entire data set in each step. A PROC MEANS step that contains the WHERE= option in the OUTPUT statement is also less efficient than the TYPES statement in PROC MEANS because SAS must calculate all possible combinations of class variables, and subsetting does not occur until the results are written to output. A basic PROC MEANS step summarizes data for all possible combinations of class variables instead of for specific combinations.
Index

& (ampersand) 288
ä conditional operator description 33 general form 59
WHERE conditions and 842
? conditional operator % (percent sign)
macro programs and specifying directives tokens and 302, 374
. (period) 315
; (semicolon) 374
A
%A directive 608
%A directive 608
access methods, selecting accumulator variables 537
ADD clause, ALTER TABLE statement (SQL) 203
ADD method 583
aliases column 230, 231, 232, 233 in-line views table 234
ALL conditional operator 33
ALL keyword
DELETE statement (DATASETS) 666
EXCEPT set operator and 132, 133
INTERSECT set operator and 137, 138
set operations and 129
UNION set operator and 141, 142
ALTER= data set option 654
ALTER TABLE statement, SQL procedure
ADD clause 203
DROP clause 204
functionality 69
general form 202
MODIFY clause 205
ampersand (&) 288
ANY conditional operator comparison operator and description 33
APPEND procedure
BASE= data set option 492
DATA= data set option 492
FORCE option 492, 496, 498
general form 492
handling large data set sorts arguments, summary functions and arithmetic expressions in macro programs 411
ARRAY statement combining data with creating arrays general form loading array elements 559, 563
lookup values and stored array values and arrays combining data in creating defined loading elements multidimensional reading values stored values 562, 563, 564, 567, 568
AS keyword, CREATE TABLE statement (SQL) 170
_ATDATETIME_ audit trail variable 656
_ATMESSAGE_ audit trail variable 656
_ATOBNO_ audit trail variable 656
_ATOPCODE_ audit trail variable 656
_ATRETCODE_ audit trail variable 656
ATTRIB statement 518
attributes 579, 585
_ATUSERID_ audit trail variable 656
AUDIT statement, DATASETS procedure 659
audit trails controlling controlling data in initiating overview reading files USER_VAR statement variables 656, 657
autocall libraries accessing macros creating defined default defined automatic macro variables defined 430
431
430
429
AVG function 53
B
%B directive 608
%B directive 608
BASE= data set option base table 513
BEFORE_IMAGE option, LOG statement 657
benchmark guidelines best practices conditional logic creating multiple output data sets DO groups eliminating data passes executing only necessary statements reading/writing essential data 761, 762, 763
subsetting variables 767, 768, 769
BETWEEN-AND conditional operator description general form 34
identifying conditions to optimize 840
binary search 518
%BQUOTE function 303
buffers controlling number of controlling page size 688, 691, 692
BUFNO= data set option 692
BUFNO= system option 692
BUFFSIZE= system option 691
BY statement, DATA step GROUPFORMAT option handling duplicate values handling large data set sorts indexes and 786
MODIFY statement and NOTSORTED option 787, 790
TRANSPOSE procedure 785
BY variable
DATA step match-merge joining tables 97, 98, 99
C
CALCULATED keyword 40
CALL MISSING routine 582
Cartesian product 81, 530
CASE expression general form 195
INDEX 967
INSET statement (SQL) 300
INDEX 969

Compression process
Number of pages
POINTER= data set option
Reasons for compressing
REUSE= data set option
REUSE= system option
Uncompressed structure
Data sets
Accessing observations directly
Appending
Audit trail overview
Audit trails, controlling data in audit trails, initiating
Audit trails, reading files
Controlling update process
Copying
Creating formats from creating multiple output
Duplicate values in generation
Handling large sorts
Hash objects and integrity constraint overview
Integrity constraints, documenting integrity constraints, placing in integrity constraints, removing integrity constraints, removing listing variables
Lookup values outside
Modifying
Modifying observations located by indexes
Renaming
Storing data
Storing raw file names
Transaction
Transposed
Data step
ATTRIB statement
Best practices
BY statement
Comparing joins and conserving storage space
CREATING indexes in creating macro variables in creating multiple macro variables in
DESCRIBE statement
DROP statement
FIRST, automatic variable
FORMAT procedure
Hash objects and hash objects and IF-THEN/ELSE statement
INDEX= option
KEEP statement
LAST, automatic variable
LENGTH statement
Match-merges
MERGE statement
Obtaining macro variable values
OUTPUT statement
PUT function
REMOVE statement
RENAME statement
REPLACE statement
REPLACE= data set option
SET statement
SUBSETTING IF statement
SUM statement
SYMMET function
SYMPUT routine and SYMPUTX routine and table lookups and tools for summarizing data
UNIQUE option
UPDATE statement
WHERE statement
Data types
Appending variables with different character creating tables by defining columns
Creating tables by defining columns numeric
DATA, IMAGE option, LOG statement
DATALINES statement
DATASETS procedure
AUDIT statement
Best practices
CHANGE statement
CONTENTS statement
COPY statement
creating tables with integrity constraints
DELETE statement
displaying index specifications
general form
IC CREATE statement
IC DELETE statement
INDEX CREATE statement
INDEX DELETE statement
INITIATE statement
LIBRARY= option
MANAGING indexes
MODIFY statement
managing indexes
MODIFY statement
manually assigning formats
RENAME statement
%DATATYP statement
DATATYPE= option, PICTURE statement (FOR-MAT)
DATE function
debugging
FEEDBACK option, SQL procedure
Macro programs
DECLARE statement
DEFAULT= option, LENGTH statement
(Data)
DEFINEDATA method
DEFINEDONE method
DEFINEKEY method
DELETE statement, DATASETS procedure
delete statement, SQL procedure
Audit trails
Functionality
General form
Updating views
Deleting
Columns from tables
Indexes
Rows in tables
Views
definements
In macro programs
In macro variable names
dense match
DESCRIBE statement
Data step
SQL procedure
DESCRIBE TABLE CONSTRAINTS statement, SQL procedure
DESCRIBE TABLE statement, SQL procedure
Displaying Dictionary table definitions
displaying index specifications
displaying indexes
displaying table structure
general form
DESCRIBE VIEW statement, SQL procedure
detail reports
Dictionary tables
Functionality
querying
digit selectors
direct access
directives, specifying pictures
DISTINCT keyword
DO loops
DO UNTIL loops
do WHILE loops
dot notation method
DOUBLE option, SQL procedure
DROP clause, ALTER TABLE statement
(SQL)
DROP= data set option
general form
DROP INDEX statement, SQL procedure
SUBSETTING variables
DROP statement
Data step
SQL procedure
DROP statement, SQL procedure
DROP TABLE statement, SQL procedure
DROP VIEW statement, SQL procedure
 DezeMTR mnemonic
 DezeNMR mnemonic
 DezeNOM mnemonic
duplicate columns, eliminating duplicate rows
Eliminating from output
Processing unique vs.
Removing efficiently
duplicate values in data sets
Efficiency
Assessing needs
Benchmark guidelines
Comparing resource usage
Computer resources
Detail reports
Estimating observations
Identifying available indexes
Index 971

benefits 223, 224, 836
BY statement and combining data with composite controlling usage 222, 223, 227, 450 copying data sets costs of using creating 226 creating in DATA step 460 creating multiple creation guidelines defined 220, 450, 458 displaying specifications documenting 465 dropping 233 identifying available maintaining 465 managing usage 229, 231 managing with DATASETS procedure managing with SQL procedure modifying observations located by querying 228, 334, 335, 830 reasons for not using 464 renaming data sets renaming variables simple 222, 227, 450, 459 subsetting data subsets of types of unique 222, 227

INFILE statement assigning names of files to be read 485 COMPRESS function date functions 489 END= option 488 FILEVAR= option 484 general form 484 INTNX function 491 INFORMAT option, CREATE TABLE statement (SQL) 165 INITIATE statement, DATASETS procedure 653 inner joins combining data horizontally 524, 529, 530 defined 13, 83 general form with outer join 97, 262 INOBS= option, SQL procedure input data sources 515 INPUT function 312, 567 INPUTC function 312 INPUTFN function 312 INSERT statement, SQL procedure audit trails 653 CASE expression 200 controlling UNDO processing FROM clause 178 functionality general form handling errors in row insertions inserting rows from query results inserting rows of data in tables SELECT clause 178 SET clause 172, 173 updating views 249 VALUES clause 172, 173, 188 instantiating hash objects integrity constraints creating outside of column specifications 184 creating tables with displaying for tables documenting 651 enforcing functionality 649 general 181, 646 general form with specifications in column specifications placing in data sets 182 referential 181, 646 removing 651 INTERSECT set operator ALL keyword and CORR keyword and functionality 137, 138 IS MISSING conditional operator 815 INTO clause, SELECT statement (SQL) 352, 355 _JOCR_ automatic variable 543, 644 MISSING conditional operator 33, 36, 840 INTOX function 491 J %j directive 608 joining data sets combining summary/detail data DATA step match-merge defined 512 indexes and lookup values and 517, 518 multiple SET statements relationships between input data sources SQL procedure support terminology 513 transactional data sets transposed data sets joining tables advantages 101 COALESCE function comparing with DATA step match-merges 98, 99 defined 13, 81 EXCEPT set operator for rows with matching values in-line views and 102, 103 indexes and 223 inner joins 13, 83 inner joins with outer-join general form INTERSECT set operator outer joins 91 OUTER UNION set operator processes defined 85 set operations 125 UNION set operator with views 106 K KEYP= option data set option general form 108 subsetting variables 767, 768, 769, 769 KEEP statement, DATA step key columns 220, 223 KEY= option MODIFY statement (DATA) 556, 639, 642 SET statement (DATA) 539 key values defined 513, 558 return codes with FIND method key variables 513 keyword parameters 583, 584 keywords, modifying set operations 129 L LABEL= option CREATE TABLE statement (SQL) 165 SELECT statement, SQL procedure labels, column 43 LAST, automatic variable 790, 820 LEFT function 313 left outer join 91, 93 %LEFT statement 430 %LENGTH function 307 LENGTH statement, DATA step 708, 710 %LET statement macro parameter support processing 327 user-defined macro variables and LIB= option, FORMAT procedure LIBRARY= option, DATASETS procedure librefs, views and 466 LIKE clause, CREATE TABLE statement (SQL) 167 LIKE conditional operator description 33 general form 37 identifying conditions to optimize specifying patterns 840 %LOCAL statement 295 local symbol table 587, 588, 391 LOG statement BEFORE_IMAGE option 657 controlling data in audit trails DATA_IMAGE option 657 ERROR_IMAGE option 659 logical expressions in macro programs lookup tables defined 513, 558 hash objects as 577 multidimensional arrays and multiple 520 stored array values 562, 563, 564, 567 TRANSPOSE function 568, 571 transposed data sets 572, 573, 574, 575, 576 LOOP= option, SQL procedure %LOWCASE statement 430 M %m directive 508 %M directive 508 macro character functions 508 %INDEX function 506
CASE expression 195 197
controlling UNDO processing functionality 189

general form 193
SET clause 193
updating table row values 192
updating views 249
WHERE clause 193 195 249
UPDATEMODE= option, MODIFY statement (DATA) 639
user-defined macro variables defined 288
global symbol table and 386
%LET statement and user variables 658
USER_VAR statement 658
USING clause, CREATE VIEW statement (SQL) 247

V
VALIDATE keyword 68
value/identifier pairs 459
VALUE statement, FORMAT procedure combining data 519
creating formats 518 603
creating formats with overlapping ranges MULTILABEL option 604
VALUES clause, INSERT statement (SQL) functionality 172 175
handling errors for row insertions 184
VAR statement, TRANSPOSE function variables 569
accumulator 537
adding descriptive names 571
appending with different lengths 496
appending with different types 498
associating formats with audit trail 518
automatic best practices 737
character 708
class 858 859 861 863
hash objects and key 585
listing in data sets 818
nenumeric renaming 469
subsetting summary statistics and 858 859

W
%w directive 608
WAYS statement, MEANS procedure 870
WHERE clause 842
SELECT statement (SQL) 10 14 32 40
UPDATE statement (SQL) 193 195 249
WHERE condition 842
compound optimization and controlling index usage 842
not optimized 842
optimizing 840
printing centile information 844
%d directive 608
WHERE= data set option 867
WHERE statement 867
DATA step 762 763 834 835
PRINT procedure 764
SORT procedure 756 757 803
word scanner
macro triggers and tokenization and 296

Y
%y directive 608
Your Turn

We welcome your feedback.

☐ If you have comments about this book, please send them to yourturn@sas.com. Include the full title and page numbers (if applicable).

☐ If you have comments about the software, please send them to suggest@sas.com.
SAS® Publishing delivers!

Whether you are new to the workforce or an experienced professional, you need to distinguish yourself in this rapidly changing and competitive job market. SAS® Publishing provides you with a wide range of resources to help you set yourself apart.

**SAS® Press Series**
Need to learn the basics? Struggling with a programming problem? You’ll find the expert answers that you need in example-rich books from the SAS Press Series. Written by experienced SAS professionals from around the world, these books deliver real-world insights on a broad range of topics for all skill levels. [support.sas.com/saspress](http://support.sas.com/saspress)

**SAS® Documentation**
To successfully implement applications using SAS software, companies in every industry and on every continent all turn to the one source for accurate, timely, and reliable information—SAS documentation. We currently produce the following types of reference documentation: online help that is built into the software, tutorials that are integrated into the product, reference documentation delivered in HTML and PDF—free on the Web, and hard-copy books. [support.sas.com/publishing](http://support.sas.com/publishing)

**SAS® Learning Edition 4.1**
Get a workplace advantage, perform analytics in less time, and prepare for the SAS Base Programming exam and SAS Advanced Programming exam with SAS® Learning Edition 4.1. This inexpensive, intuitive personal learning version of SAS includes Base SAS® 9.1.3, SAS/STAT®, SAS/GRAPH®, SAS/QC®, SAS/ETS®, and SAS® Enterprise Guide® 4.1. Whether you are a professor, student, or business professional, this is a great way to learn SAS. [support.sas.com/LE](http://support.sas.com/LE)