



PROC SQL

Beyond the Basics Using SAS®

Third Edition



Kirk Paul Lafler

The correct bibliographic citation for this manual is as follows: Lafler, Kirk Paul. 2019. *PROC SQL: Beyond the Basics Using SAS®, Third Edition*. Cary, NC: SAS Institute Inc.

PROC SQL: Beyond the Basics Using SAS®, Third Edition

Copyright © 2019, SAS Institute Inc., Cary, NC, USA

978-1-64295-192-9(Hard cover)

978-1-63526-684-9 (Hardcopy)

978-1-63526-683-2 (Web PDF)

978-1-63526-681-8 (epub)

978-1-63526-682-5 (mobi)

All Rights Reserved. Produced in the United States of America.

For a hard copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

March 2019

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

SAS software may be provided with certain third-party software, including but not limited to open-source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to

<http://support.sas.com/thirdpartylicenses>.

Contents

About This Book	vii
Chapter 1: Designing Database Tables	1
Introduction.....	1
Database Design.....	1
Column Names and Reserved Words	7
Data Integrity	8
Database Tables Used in This Book	8
Table Contents	11
Summary	20
Chapter 2: Working with Data in PROC SQL.....	21
Introduction.....	21
The SELECT Statement and Clauses	21
Overview of Data Types	23
SQL Operators, Functions, and Keywords	31
Dictionary Tables.....	66
Summary	82
Chapter 3: Formatting Output.....	83
Introduction.....	83
Formatting Output	83
Formatting Output with the Output Delivery System.....	102
Summary	108
Chapter 4: Coding PROC SQL Logic.....	109
Introduction.....	109
Conditional Logic.....	109
CASE Expressions	114
Interfacing PROC SQL with the Macro Language	139
Summary	152
Chapter 5: Creating, Populating, and Deleting Tables	153
Introduction.....	153
Creating Tables	154
Populating Tables.....	160
Integrity Constraints	177
Deleting Rows in a Table.....	189

Deleting Tables	190
Summary	193
Chapter 6: Modifying and Updating Tables and Indexes.....	195
Introduction	195
Modifying Tables	195
Indexes.....	207
Updating Data in a Table.....	218
Summary.....	219
Chapter 7: Coding Complex Queries	221
Introduction	222
Introducing Complex Queries.....	222
Joins.....	222
Why Joins Are Important.....	223
Cartesian Product Joins	229
Inner Joins.....	230
Outer Joins.....	240
Subqueries	246
Set Operations	256
Data Structure Transformations	265
Chapter 8: Working with Views.....	289
Introduction	289
Views—Windows to Your Data	289
Eliminating Redundancy.....	299
Restricting Data Access—Security.....	299
Hiding Logic Complexities	300
Nesting Views	302
Updatable Views	304
Deleting Views	311
Summary.....	312
Chapter 9: Fuzzy Matching Programming	313
Introduction	313
Data Sets Used in Examples.....	314
6-Step Fuzzy Matching Process.....	316
Summary.....	342
Chapter 10: Data-driven Programming	343
Introduction	343
Programming Paradigms.....	343
SAS Metadata Sources	344
DICTIONARY Tables	350
CALL EXECUTE Routine	354

Custom-defined Formats	357
Macro Language	360
Summary	364
Chapter 11: Troubleshooting and Debugging.....	365
Introduction.....	365
The World of Bugs.....	365
The Debugging Process.....	366
Types of Problems	367
Troubleshooting and Debugging Techniques	368
Undocumented PROC SQL Options	382
Summary	389
Chapter 12: Tuning for Performance and Efficiency	391
Introduction.....	391
Understanding Performance Tuning.....	391
Sorting and Performance.....	392
User-Specified Sorting (SORTPGM= System Options).....	392
Grouping and Performance	393
Splitting Tables.....	393
Indexes and Performance	394
Reviewing CONTENTS Output and System Messages	395
Optimizing WHERE Clause Processing with Indexes	398
Summary	404
References.....	405

About This Book

What Does This Book Cover?

PROC SQL: Beyond the Basics Using SAS, Third Edition, is a step-by-step, example-driven guide that helps readers master the language of PROC SQL. Packed with analysis and examples illustrating an assortment of PROC SQL options, statements, and clauses, this book covers all the basics, but also offers extensive guidance on complex topics such as set operators and correlated subqueries.

The third edition explores new and powerful features in SAS® 9.4, including topics such as IFN and IFN functions, nearest neighbor processing, the HAVING clause, and indexes. It also features two completely new chapters on fuzzy matching and data-driven programming. Delving into the workings of PROC SQL with greater analysis and discussion, *PROC SQL: Beyond the Basic Using SAS, Third Edition*, examines a broad range of topics and provides greater detail about this powerful database language using discussion and numerous real-world examples.

Is This Book for You?

The intended audience for this book includes SAS users, programmers, business analysts, application and software developers, database analysts and administrators, help desk support staff, statisticians, IT support staff, and other professionals who want or need application-oriented information to extend their knowledge of PROC SQL beyond the basics.

This book offers readers under-the-hood, behind-the-scenes knowledge on how PROC SQL works and is the perfect tool for students pursuing an undergraduate or graduate information science, computer science, or cognitive science degree.

What Should You Know about the Examples?

This book includes tutorials for you to follow to gain hands-on experience with SAS.

Software Used to Develop the Book's Content

SAS® 9.4 was used to develop all content for this book.

Example Code and Data

You can access the example code and data for this book by linking to its author page at <https://support.sas.com/lafier>. Then, look for the cover thumbnail of this book, and select Example Code and Data to display the SAS programs that are included in this book.

SAS University Edition



This book is compatible with SAS University Edition. If you are using SAS University Edition, then begin here: <https://support.sas.com/ue-data>.

We Want to Hear from You

Do you have questions about a SAS Press book that you are reading? Contact us at saspress@sas.com.

SAS Press books are written *by* SAS Users *for* SAS Users. Please visit sas.com/books to sign up to request information on how to become a SAS Press author.

We welcome your participation in the development of new books and your feedback on SAS Press books that you are using. Please visit sas.com/books to sign up to review a book

Learn about new books and exclusive discounts. Sign up for our new books mailing list today at <https://support.sas.com/en/books/subscribe-books.html>.

Learn more about this author by visiting his author page at <http://support.sas.com/lafler>. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

Acknowledgments

This book was made possible because of the support and encouragement of many people. I would like to extend my sincerest thanks to each person who encouraged me to write this book. For, as G. B. Stern once said, “Silent gratitude isn’t very much use to anyone.” So from me to you, I want to thank everyone who had a hand in helping to make the first, second, and third editions possible.

To Sian Roberts for supporting my desire to introduce and develop new topics for the third edition. It was a distinct pleasure working with Sian where her encouragement, support, and coordination of the entire development process made this project possible.

To Suzanne Morgen, Development Editor, SAS Press for her guidance, feedback, and support, and for bringing clarity throughout the writing process.

To the technical reviewers who provided valuable feedback, suggestions, and technical accuracy on the new and revised chapters including the SAS code. Special thanks to Stephen Sloan, Vince DelGobbo, Leonid Batkhan, and Lewis Church for performing technical reviews of the third edition.

To the copy editors who made a difficult job look easy. A heart-felt thank you to Catherine Connolly who performed her magic identifying typos, and turning words, run-on sentences and, sometimes, long-winded paragraphs into coherent prose.

To Julie Palmieri, former Editor-in-Chief of SAS Press, for supporting my desire to develop a more comprehensive second edition. Her encouragement and support gave me the inspiration to see this project to completion.

To Stacey Hamilton at SAS Institute Inc. for her editorial assistance and coordinating the technical review process while I developed the second edition.

To Paul Kent at SAS Institute Inc. for his contagious enthusiasm, great examples, clear explanations, and mentorship with some of the fine points of PROC SQL and its capabilities over the years.

To David Baggett at SAS Institute Inc. for encouraging the idea of a book on PROC SQL and accepting the original first edition manuscript many years ago. His encouragement and support over the years has meant a great deal to me.

To Stephenie Joyner at SAS Institute Inc. for her support and encouragement during the development of the first edition and for coordinating the technical review process.

To Charles Edwin Shipp, Michael Raithel, Mary Rosenbloom, Richard La Valley, Clark Roberts, Stephen Sloan, Josh Horstman, and Sunil Kumar Gupta for their friendship, support, and encouragement through the years.

To Art Carpenter of California Occidental Consultants for believing that a “beyond the basics” type of book on PROC SQL would be useful to SAS users.

To the many people at SAS Institute Inc. with whom I have developed so many friendships over the years. I’d like to express my thanks to each of you as well as all the knowledgeable people in SAS Technical Support. Thank you for developing and supporting a great product and enabling me to have a rewarding and enjoyable career for all these years.

To the SAS user group community around the world: You are the greatest group of professionals anywhere.

To the many teachers I have had in my life. Special thanks go to Lawrence Delk (6th grade); Mr. Almeida (12th grade); Professor Carl Kromp (Industrial Engineering); Joseph J. Moder, Ph.D. (Management Science); Charles N. Kurucz, Ph.D. (Management Science); John F. Stewart, Ph.D. (Computer Information Systems); Earl Wiener, Ph.D. (Management Science); Howard Seth Gitlow, Ph.D. (Management Science); Dean Paul K. Sugrue, Ph.D. (University of Miami School of Business); Edward K. Baker III, Ph.D. (Management Science); Robert T. Grauer, Ph.D. (Computer Information Systems); and Ulu (Rydacom) for sharing your knowledge and enthusiasm.

To the countless people I have worked with and the companies I have worked for – the experiences and memories have been invaluable.

To my mother, father, and brother for sharing life’s many lessons, experiences, and memories. Your love and encouragement through the years fueled my desire to learn, work hard, and experience life to the fullest.

Finally, to my wife, Darlynn, and son, Ryan, for your love, support, and sense of balance between family and work. I love you both so very much.

Thank you!

Chapter 1: Designing Database Tables

Introduction	1
Database Design.....	1
Conceptual View	2
Table Definitions	2
Redundant Information	3
Normalization	3
Normalization Strategies.....	4
Column Names and Reserved Words	7
ANSI SQL Reserved Words.....	7
SQL Code	7
Data Integrity.....	8
Referential Integrity	8
Database Tables Used in This Book.....	8
CUSTOMERS Table	8
INVENTORY Table	9
INVOICE Table.....	9
MANUFACTURERS Table	9
PRODUCTS Table	10
PURCHASES Table	10
Table Contents	11
The Database Structure	13
Sample Database Tables	14
Summary	20

Introduction

The area of database design is very important in relational processes. Much has been written on this subject, including entire textbooks and thousands of technical papers. No pretenses are made about the thoroughness of this very important subject in these pages. Rather, an attempt is made to provide a quick-start introduction for those readers who are unfamiliar with the issues and techniques of basic design principles. Readers needing more information are referred to the references listed in the back of this book. As you read this chapter, the following points should be kept in mind.

Database Design

Activities related to good database design require the identification of end-user requirements and involve defining the structure of data values on a physical level. Database design begins with a *conceptual view* of what is needed. The next step, called *logical design*, consists of developing a formal description of database entities and relationships to satisfy user requirements. Seldom does a database consist of a single table. Consequently, tables of

interrelated information are created to enable more complex and powerful operations on data. The final step, referred to as *physical design*, represents the process of achieving optimal performance and storage requirements of the logical database.

Conceptual View

The health and well-being of a database depends on its database design. A database must be in balance with all of its components (or optimized) to avoid performance and operation bottlenecks. Database design doesn't just happen and is not a process that occurs by chance. It involves planning, modeling, creating, monitoring, and adjusting to satisfy the endless assortment of user requirements without impeding resource requirements. Of central importance to database design is the process of planning. Planning is a valuable component that, when absent, causes a database to fall prey to a host of problems including poor performance and difficulty in operation. Database design consists of three distinct phases, as illustrated in Figure 1.1.

Figure 1.1: Three Distinct Phases of Database Design

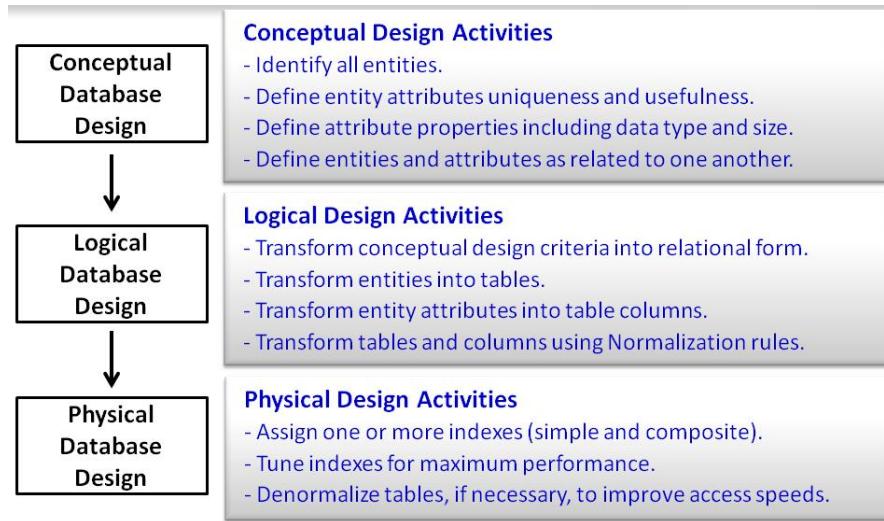


Table Definitions

PROC SQL uses a model of data that is conceptually stored as multisets rather than as physical files. A physical file consists of one or more records ordered sequentially or some other way. Programming languages such as COBOL and FORTRAN evolved to process files of this type by performing operations one record at a time. These languages were generally designed and used to mimic the way people process paper forms.

PROC SQL was designed to work with multisets of data. Multisets have no order, and members of a multiset are of the same type using a data structure known as a table. For classification purposes, a table is a base table consisting of zero or more rows and one or more columns, or a table is a virtual table (called a *view*), which can be used the same way that a table can be used (see Chapter 8, "Working with Views").

Redundant Information

One of the rules of good database design requires that data not be redundant or duplicated in the same database. The rationale for this conclusion originates from the belief that if data appears more than once in a database, then there is reason to believe that one of the pieces of data is likely to be in error. Furthermore, redundancy often leads to the following:

- Inconsistencies, because errors are more likely to result when facts are repeated.
- Update anomalies where the insertion, modification, or deletion of data may result in inconsistencies.

Another thing to watch for is the appearance of too many columns containing NULL values. When this occurs, the database is probably not designed properly. To alleviate potential table design issues, a process referred to as *normalizing* is performed. When properly done, this ensures the complete absence of redundant information in a table.

Normalization

The development of an optimal database design is an important element in the life cycle of a database. Not only is it critical for achieving maximum performance and flexibility while working with tables and data, it is essential to the organization of data by reducing or minimizing redundancy in one or more database tables. The process of table design is frequently referred to by database developers and administrators as *normalization*.

The normalization process is used for reducing redundancy in a database by converting complex data structures into simple data structures. It is carried out for the following reasons:

- To organize the data to save space and to eliminate any duplication or repetition of data.
- To enable simple retrieval of data to satisfy query and report requests.
- To simplify data manipulation requests such as data insertions, updates, and deletions.
- To reduce the impact associated with reorganizing or restructuring data as new application requirements arise.

The normalization process attempts to simplify the relationship between columns in a database by splitting larger multicolumn tables into two or more smaller tables containing fewer columns. The rationale for doing this is contained in a set of data design guidelines called *normal forms*. The guidelines provide designers with a set of rules for converting one or two large database tables containing numerous columns into a normalized database consisting of multiple tables and only those columns that should be included in each table. The normalization process consists of multiple steps with each succeeding step subscribing to the rules of the previous steps.

Normalization helps to ensure that a database does not contain redundant information in two or more of its tables. In an application, normalization prevents the destruction of data or the creation of incorrect data in a database. What this means is that information of fact is represented only once in a database, and any possibility of it appearing more than once is not, or should not be, allowed.

As database designers and analysts proceed through the normalization process, many are not satisfied unless a database design is carried out to at least third normal form (3NF). Joe Celko in his popular book *SQL for Smarties: Advanced SQL Programming* (Morgan Kaufman, 1999), describes 3NF this way: “Databases are considered to be in 3NF when a column is dependent on the key, the whole key, and nothing but the key.”

While the normalization guidelines are extremely useful, some database purists actually go to great lengths to remove any and all table redundancies even at the expense of performance. This is in direct contrast to other database experts who follow the guidelines less rigidly in an attempt to improve the performance of a database by only going as far as third normal form (or 3NF). Whatever your preference, you should keep this thought in mind as you normalize database tables. A fully normalized database often requires a greater number of joins and can adversely affect the speed of queries. Celko mentions that the process of joining multiple tables in a fully normalized database is costly, specifically affecting processing time and computer resources.

Normalization Strategies

After transforming entities and attributes from the conceptual design into a logical design, the tables and columns are created. This is when a process known as *normalization* occurs. Normalization refers to the process of making your database tables subscribe to certain rules. Many, if not most, database designers are satisfied when third normal form (3NF) is achieved and, for the objectives of this book, I will stop at 3NF, too. To help explain the various normalization steps, an example scenario follows.

First Normal Form (1NF)

First normal form (1NF) involves the elimination of data redundancy or repeating information from a table. A table is considered to be in first normal form when all of its columns describe the table completely and when each column in a row has only one value. A table satisfies 1NF when each column in a row has a single value and no repeating group information. Essentially, every table meets 1NF as long as an array, list, or other structure has not been defined. The following table illustrates a table satisfying the 1NF rule because it has only one value at each row-and-column intersection. The table is in ascending order by CUSTNUM and consists of customers and the purchases they made at an office supply store.

Table 1.1: First Normal Form (1NF) Table

CUSTNUM	CUSTNAME	CUSTCITY	ITEM	UNITS	UNITCOST	MANUCITY
1	Smith	San Diego	Chair	1	\$179.00	San Diego
1	Smith	San Diego	Pens	12	\$0.89	Los Angeles
1	Smith	San Diego	Paper	4	\$6.95	Washington
1	Smithe	San Diego	Stapler	1	\$8.95	Los Angeles
7	Lafler	Spring Valley	Mouse Pad	1	\$11.79	San Diego
7	Loffler	Spring Valley	Pens	24	\$1.59	Los Angeles
13	Thompson	Miami	Markers	.	\$0.99	Los Angeles

Second Normal Form (2NF)

Second normal form (2NF) addresses the relationships between sets of data. A table is said to be in second normal form when all the requirements of 1NF are met and a foreign key is used to link any data in one table which has relevance to another table. The very nature of leaving

a table in first normal form (1NF) may present problems due to the repetition of some information in the table. One noticeable problem is that Table 1.1 has repetitive information in it. Another problem is that there are misspellings in the customer name. Although repeating information may be permissible with hierarchical file structures and other legacy type file structures, it does pose a potential data consistency problem as it relates to relational data.

To describe how data consistency problems can occur, let's say that a customer takes a new job and moves to a new city. In changing the customer's city to the new location, it would be very easy to miss one or more occurrences of the customer's city resulting in a customer residing incorrectly in two different cities. Assuming that our table is only meant to track one unique customer per city, this would definitely be a data consistency problem. Essentially, second normal form (2NF) is important because it says that every non-key column must depend on the entire primary key.

Tables that subscribe to 2NF prevent the need to make changes in more than one place. What this means in normalization terms is that tables in 2NF have no partial key dependencies. As a result, our database that consists of a single table that satisfies 1NF will need to be split into two separate tables in order to subscribe to the 2NF rule. Each table would contain the CUSTNUM column to connect the two tables. Unlike the single table in 1NF, the tables in 2NF allow a customer's city to be easily changed whenever they move to another city because the CUSTCITY column only appears once. The tables in 2NF would be constructed as follows.

Table 1.2: CUSTOMERS Table

CUSTNUM	CUSTNAME	CUSTCITY
1	Smith	San Diego
1	Smithe	San Diego
7	Lafler	Spring Valley
13	Thompson	Miami

Table 1.3: PURCHASES Table

CUSTNUM	ITEM	UNITS	UNITCOST	MANUCITY
1	Chair	1	\$179.00	San Diego
1	Pens	12	\$0.89	Los Angeles
1	Paper	4	\$6.95	Washington
1	Stapler	1	\$8.95	Los Angeles
7	Mouse Pad	1	\$11.79	San Diego
7	Pens	24	\$1.59	Los Angeles
13	Markers	.	\$0.99	Los Angeles

Third Normal Form (3NF)

Referring to the two tables constructed according to the rules of 2NF, you may have noticed that the PURCHASES table contains a column called MANUCITY. The MANUCITY column stores the city where the product manufacturer is headquartered. Keeping this column in the PURCHASES table violates the third normal form (3NF) because MANUCITY does not provide factual information about the primary key column (CUSTNUM) in the PURCHASES table. Consequently, tables are considered to be in third normal form (3NF) when each column is dependent on the key, the whole key, and nothing but the key. The

tables in 3NF are constructed so the MANUCITY column would be in a table of its own as follows.

Table 1.4: CUSTOMERS Table

CUSTNUM	CUSTNAME	CUSTCITY
1	Smith	San Diego
1	Smithe	San Diego
7	Lafler	Spring Valley
13	Thompson	Miami

Table 1.5: PURCHASES Table

CUSTNUM	ITEM	UNITS	UNITCOST
1	Chair	1	\$179.00
1	Pens	12	\$0.89
1	Paper	4	\$6.95
1	Stapler	1	\$8.95
7	Mouse Pad	1	\$11.79
7	Pens	24	\$1.59
13	Markers	.	\$0.99

Table 1.6: MANUFACTURERS Table

MANUNUM	MANUCITY
101	San Diego
112	San Diego
210	Los Angeles
212	Los Angeles
213	Los Angeles
214	Los Angeles
401	Washington

Beyond Third Normal Form

In general, database designers are satisfied when their database tables subscribe to the rules in 3NF. But, it is not uncommon for others to normalize their database tables to fourth normal form (4NF) where independent one-to-many relationships between primary key and non-key columns are forbidden. Some database purists will even normalize to fifth normal form (5NF) where tables are split into the smallest pieces of information in an attempt to eliminate any and all table redundancies. Although constructing tables in 5NF may provide the greatest level of database integrity, it is neither practical nor desired by most database practitioners.

There is no absolute right or wrong reason for database designers to normalize beyond 3NF as long as they have considered all the performance issues that may arise by doing so. A common problem that occurs when database tables are normalized beyond 3NF is that a large number of small tables are generated. In these cases, an increase in time and computer resources frequently occurs because small tables must first be joined before a query, report, or statistic can be produced.

Column Names and Reserved Words

According to the American National Standards Institute (ANSI), SQL is the standard language used with relational database management systems. The ANSI Standard reserves a number of SQL keywords from being used as column names. The SAS SQL implementation is not as rigid, but users should be aware of what reserved words exist to prevent unexpected and unintended results during SQL processing. Column names should conform to proper SAS naming conventions (as described in the *SAS Language Reference*), and they should not conflict with certain reserved words found in the SQL language. The following list identifies the reserved words found in the ANSI SQL standard.

ANSI SQL Reserved Words

AS	INNER	OUTER
CASE	INTERSECT	RIGHT
EXCEPT	JOIN	UNION
FROM	LEFT	UPPER
FULL	LOWER	USER
GROUP	ON	WHEN
HAVING	ORDER	WHERE

You probably will not encounter too many conflicts between a column name and an SQL reserved word, but when you do you will need to follow a few simple rules to prevent processing errors from occurring. As was stated earlier, although PROC SQL's naming conventions are not as rigid as other vendor's implementations, care should still be exercised, in particular when PROC SQL code is transferred to other database environments expecting it to run error free. If a column name in an existing table conflicts with a reserved word, you have three options at your disposal:

1. Physically rename the column name in the table, as well as any references to the column.
2. Use the RENAME= data set option to rename the desired column in the current query.
3. Specify the PROC SQL option DQUOTE=ANSI, and surround the column name (reserved word) in double quotes, as illustrated below.

SQL Code

```
PROC SQL DQUOTE=ANSI;
  SELECT *
  FROM RESERVED_WORDS
  WHERE "WHERE"='EXAMPLE';
QUIT;
```

Data Integrity

Webster's New World Dictionary defines *integrity* as “the quality or state of being complete; perfect condition; reliable; soundness.” Data integrity is a critical element that every organization must promote and strive for. It is imperative that the data tables in a database environment be reliable, free of errors, and sound in every conceivable way. The existence of data errors, missing information, broken links, and other related problems in one or more tables can impact decision-making and information reporting activities resulting in a loss of confidence among users.

Applying a set of rules to the database structure and content can ensure the integrity of data resources. These rules consist of table and column constraints, and will be discussed in detail in Chapter 5, “Creating, Populating, and Deleting Tables.”

Referential Integrity

Referential integrity refers to the way in which database tables handle update and delete requests. Database tables frequently have a *primary key* where one or more columns have a unique value by which rows in a table can be identified and selected. Other tables may have one or more columns called a *foreign key* that are used to connect to some other table through its value. Database designers frequently apply rules to database tables to control what happens when a primary key value changes and its effect on one or more foreign key values in other tables. These referential integrity rules apply restrictions on the data that may be updated or deleted in tables.

Referential integrity ensures that rows in one table have corresponding rows in another table. This prevents lost linkages between data elements in one table and those of another enabling the integrity of data to always be maintained. Using the 3NF tables defined earlier, a foreign key called CUSTNUM can be defined in the PURCHASES table that corresponds to the primary key CUSTNUM column in the CUSTOMERS table. Users are referred to Chapter 5, “Creating, Populating, and Deleting Tables” for more details on assigning referential integrity constraints.

Database Tables Used in This Book

This section describes a database or library of tables that is used by an imaginary computer hardware and software wholesaler. The library consists of six tables: Customers, Inventory, Invoice, Manufacturers, Products, and Purchases. The examples used throughout this book are based on this library (database) of tables and are described and displayed below. An alphabetical description of each table used throughout this book appears below.

CUSTOMERS Table

The CUSTOMERS table contains customers that have purchased computer hardware and software products from a manufacturer. Each customer is uniquely identified with a customer number. A description of each column in the Customers table follows.

Table 1.7: Description of Columns in the Customers Table

CUSTNUM	Unique number identifying the customer.
CUSTNAME	Name of customer.
CUSTCITY	City where customer is located.

INVENTORY Table

The INVENTORY table contains customer inventory information consisting of computer hardware and software products. The Inventory table contains no historical data. As inventories are replenished, the old quantity is overwritten with the new quantity. A description of each column in the Inventory table follows.

Table 1.8: Description of Columns in the Inventory Table

PRODNUM	Unique number identifying product.
MANUNUM	Unique number identifying the manufacturer.
INVENQTY	Number of units of product in stock.
ORDDATE	Date product was last ordered.
INVENCST	Cost of inventory in customer's stock room.

INVOICE Table

The INVOICE table contains information about customers who purchased products. Each invoice is uniquely identified with an invoice number. A description of each column in the Invoice table follows.

Table 1.9: Description of Columns in the Invoice Table

INVNUM	Unique number identifying the invoice.
MANUNUM	Unique number identifying the manufacturer.
CUSTNUM	Customer number.
PRODNUM	Product number.
INVQTY	Number of units sold.
INVPRICE	Unit price.

MANUFACTURERS Table

The MANUFACTURERS table contains companies who make computer hardware and software products. Two companies cannot have the same name. No historical data is kept in this table. If a company is sold or stops making computer hardware or software, then the

manufacturer is dropped from the table. In the event that a manufacturer has an address change, the old address is overwritten with the new address. A description of each column in the Manufacturers table follows.

Table 1.10: Description of Columns in the Manufacturers Table

MANUNUM	Unique number identifying the manufacturer.
MANUNAME	Name of manufacturer.
MANUCITY	City where manufacturer is located.
MANUSTAT	State where manufacturer is located.

PRODUCTS Table

The PRODUCTS table contains computer hardware and software products offered for sale by the manufacturer. Each product is uniquely identified with a product number. A description of each column in the Products table follows.

Table 1.11: Description of Columns in the Products Table

PRODNUM	Unique number identifying the product.
PRODNAME	Name of product.
MANUNUM	Unique number identifying the manufacturer.
PRODTYPE	Type of product.
PRODCOST	Cost of product.

PURCHASES Table

The PURCHASES table contains computer hardware and software products purchased by customers. Each product is uniquely identified with a product number. A description of each column in the Purchases table follows.

Table 1.12: Description of Columns in the Purchases Table

CUSTNUM	Unique number identifying the customer.
ITEM	Name of product.
UNITS	Number of items purchased by customer.
UNITCOST	Cost of product.

Table Contents

An alphabetical list of tables, variables, and attributes for each table is displayed below.

Output 1.1: Customers CONTENTS Output

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	custcity	Char	20	Customer's Home City
2	custname	Char	25	Customer Name
1	custnum	Num	3	Customer Number

Output 1.2: Inventory CONTENTS Output

Alphabetic List of Variables and Attributes						
#	Variable	Type	Len	Format	Informat	Label
4	invencst	Num	6	DOLLAR10.2		Inventory Cost
2	invenqty	Num	3			Inventory Quantity
5	manunum	Num	3			Manufacturer Number
3	orddate	Num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
1	prodnum	Num	3			Product Number

Output 1.3: Invoice CONTENTS Output

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
3	custnum	Num	3		Customer Number
1	invnum	Num	3		Invoice Number
5	invprice	Num	5	DOLLAR12.2	Invoice Unit Price
4	invqty	Num	3		Invoice Quantity - Units Sold
2	manunum	Num	3		Manufacturer Number
6	prodnum	Num	3		Product Number

Output 1.4: Manufacturers CONTENTS Output

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	manucity	Char	20	Manufacturer City
2	manuname	Char	25	Manufacturer Name
1	manunum	Num	3	Manufacturer Number
4	manustat	Char	2	Manufacturer State

Output 1.5: Products CONTENTS Output

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
3	manunum	Num	3		Manufacturer Number
5	prodcost	Num	5	DOLLAR9.2	Product Cost
2	prodname	Char	25		Product Name
1	prodnum	Num	3		Product Number
4	prodtype	Char	15		Product Type

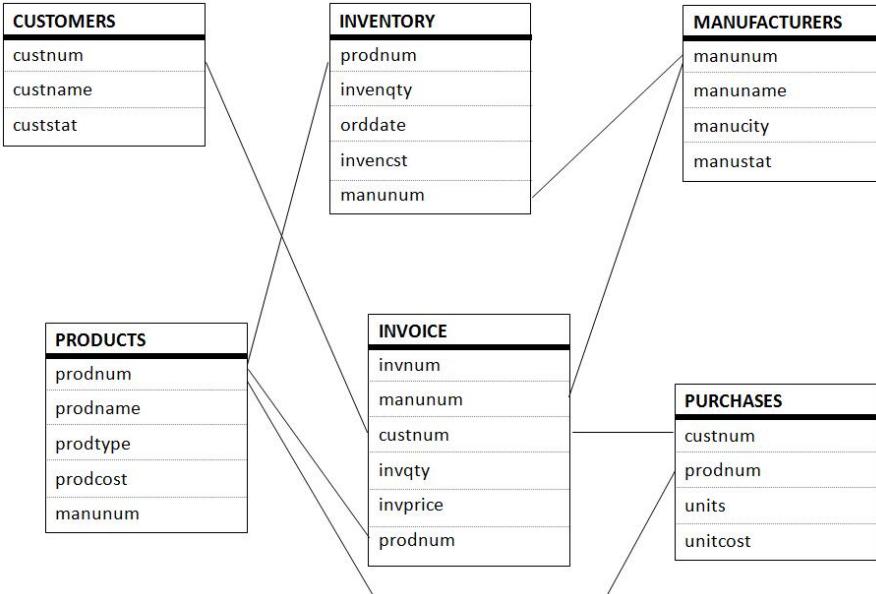
Output 1.6: Purchases CONTENTS Output

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
1	custnum	Num	4		Custnum
2	prodnum	Num	3		Prodnum
4	unitcost	Num	4	DOLLAR12.2	Unitcost
3	units	Num	3		Units

The Database Structure

The logical relationship between each table, and the columns common to each, appear below.

Figure 1.2. Logical Database Structure



Sample Database Tables

The following tables: Customers, Inventory, Manufacturers, Products, Invoice, and Purchases represent a relational database that will be illustrated in the examples in this book. These tables are small enough to follow easily, but complex enough to illustrate the power of SQL. The data contained in each table appears below.

Table 1.13: CUSTOMERS Table

Obs	custnum	custname	custcity
1	101	La Mesa Computer Land	La Mesa
2	201	Vista Tech Center	Vista
3	301	Coronado Internet Zone	Coronado
4	401	La Jolla Computing	La Jolla
5	501	Alpine Technical Center	Alpine
6	601	Oceanside Computer Land	Oceanside
7	701	San Diego Byte Store	San Diego
8	801	Jamul Hardware & Software	Jamul
9	901	Del Mar Tech Center	Del Mar
10	1001	Lakeside Software Center	Lakeside
11	1101	Bonsall Network Store	Bonsall
12	1201	Rancho Santa Fe Tech	Rancho Santa Fe
13	1301	Spring Valley Byte Center	Spring Valley
14	1401	Poway Central	Poway
15	1501	Valley Center Tech Center	Valley Center
16	1601	Fairbanks Tech USA	Fairbanks Ranch
17	1701	Blossom Valley Tech	Blossom Valley
18	1801	Chula Vista Networks	
N = 18			

Table 1.14: INVENTORY Table

Obs	prodnum	invenqty	orddate	invencst	manunum
1	1110	20	09/01/2000	\$45,000.00	111
2	1700	10	08/15/2000	\$28,000.00	170
3	5001	5	08/15/2000	\$1,000.00	500
4	5002	3	08/15/2000	\$900.00	500
5	5003	10	08/15/2000	\$2,000.00	500
6	5004	20	09/01/2000	\$1,400.00	500
7	5001	2	09/01/2000	\$1,200.00	600

Table 1.15: INVOICE Table

Obs	invnum	manunum	custnum	invqty	invprice	prodnum
1	1001	500	201	5	\$1,495.00	5001
2	1002	600	1301	2	\$1,598.00	6001
3	1003	210	101	7	\$245.00	2101
4	1004	111	501	3	\$9,600.00	1110
5	1005	500	801	2	\$798.00	5002
6	1006	500	901	4	\$396.00	6000
7	1007	500	401	7	\$23,100.00	1200

Table 1.16: MANUFACTURERS Table

Obs	manunum	manuname	manucity	manustat
1	111	Cupid Computer	Houston	TX
2	210	Global Comm Corp	San Diego	CA
3	600	World Internet Corp	Miami	FL
4	120	Storage Devices Inc	San Mateo	CA
5	500	KPL Enterprises	San Diego	CA
6	700	San Diego PC Planet	San Diego	CA
N = 6				

Table 1.17: PRODUCTS Table

Obs	prodnum	prodname	manunum	prodtype	prodcost
1	1110	Dream Machine	111	Workstation	\$3,200.00
2	1200	Business Machine	120	Workstation	\$3,300.00
3	1700	Travel Laptop	170	Laptop	\$3,400.00
4	2101	Analog Cell Phone	210	Phone	\$35.00
5	2102	Digital Cell Phone	210	Phone	\$175.00
6	2200	Office Phone	220	Phone	\$130.00
7	5001	Spreadsheet Software	500	Software	\$299.00
8	5002	Database Software	500	Software	\$399.00
9	5003	Wordprocessor Software	500	Software	\$299.00
11	5004	Graphics Software	500	Software	\$299.00
N = 10					

Table 1.18: PURCHASES Table

Obs	custnum	prodnum	units	unitcost
1	1701	1110	1	\$3,200.00
2	101	5001	7	\$299.00
3	701	5001	11	\$299.00
4	701	5003	8	\$299.00
5	701	5002	4	\$399.00
6	701	5004	3	\$299.00
7	701	1700	2	\$3,400.00
8	701	1200	3	\$3,300.00
9	701	1110	2	\$3,200.00
10	1301	5001	3	\$299.00
11	1301	5003	5	\$299.00
12	1301	5002	2	\$399.00
13	901	1700	2	\$3,400.00
14	901	1200	3	\$3,300.00
15	901	1110	5	\$3,200.00
16	901	5001	9	\$299.00
17	901	5002	5	\$399.00
18	901	5003	8	\$299.00
19	901	5004	2	\$299.00
20	401	5001	11	\$299.00

21	401	5002	5	\$399.00
22	401	5003	7	\$299.00
23	401	5004	3	\$299.00
24	401	1700	3	\$3,400.00
25	401	1200	6	\$3,300.00
26	201	5001	6	\$299.00
27	201	5001	6	\$299.00
28	201	5003	9	\$299.00
29	201	5002	4	\$399.00
30	201	1700	3	\$3,400.00
31	901	5001	2	\$299.00
32	201	5001	2	\$299.00
33	201	2102	5	\$175.00
34	1101	2102	9	\$175.00
35	1301	2102	11	\$175.00
36	1401	2102	7	\$175.00
37	801	2102	5	\$175.00
38	501	2102	12	\$175.00
39	301	2102	8	\$175.00
40	1101	2200	3	\$130.00
41	101	2102	9	\$175.00

42	101	5003	3	\$299.00
43	101	5004	2	\$299.00
44	101	1200	3	\$3,300.00
45	101	1700	5	\$3,400.00
46	1301	1700	3	\$3,400.00
47	1601	1700	7	\$3,400.00
48	1801	1700	4	\$3,400.00
49	1001	1700	5	\$3,400.00
50	1101	1700	2	\$3,400.00
51	1201	1200	8	\$3,300.00
52	501	5001	3	\$299.00
53	501	5003	5	\$299.00
54	501	5004	1	\$299.00
55	501	1700	4	\$3,400.00
56	301	5001	6	\$299.00
57	501	2102	9	\$175.00
N = 57				

Summary

1. Good database design often improves the relative ease by which tables can be created and populated in a relational database and can be implemented into any database (see the “Conceptual View” section).
2. SQL was designed to work with sets of data and accesses a data structure known as a table or a “virtual” table, known as a view (see the “Table Definitions” section).
3. Achieving optimal design of a database means that the database contains little or no redundant information in two or more of its tables. This means that good database design calls for little or no replication of data (see the “Redundant Information” section).
4. Good database design avoids data redundancy, update anomalies, costly or inefficient processing, coding complexities, complex logical relationships, long application development times, and/or excessive storage requirements (see the “Normalization” section).
5. Design decisions made in one phase may involve making one or more tradeoffs in another phase (see the “Normalization” section).
6. A database in third normal form (3NF) is defined as a column that is dependent on the key, the whole key, and nothing but the key (see the “Normalization” section).

Chapter 2: Working with Data in PROC SQL

Introduction	21
The SELECT Statement and Clauses	21
Overview of Data Types.....	23
Numeric Data	24
Date and Time Column Definitions	26
Character Data.....	27
Missing Values and NULL	27
Arithmetic and Missing Data	28
SQL Keywords	30
SQL Operators, Functions, and Keywords	31
Comparison Operators.....	32
Logical Operators.....	33
Arithmetic Operators.....	34
Character String Operators and Functions	36
Summarizing Data	51
Predicates	55
CALCULATED Keyword	63
Dictionary Tables	66
Dictionary Tables and Metadata	67
Displaying Dictionary Table Definitions	68
Dictionary Table Column Names.....	69
Accessing a Dictionary Table's Contents.....	72
Summary	82

Introduction

PROC SQL is essentially a database language as opposed to a procedural or computational language. This chapter’s focus is on working with data in PROC SQL using the SELECT statement. Often referred to as an SQL query, the SELECT statement is the most versatile statement in SQL and is used to read data from one or more database tables (or data sets). It also supports numerous extensions including keywords, operators, functions, and predicates, and returns the data in a table-like structure called a result-set.

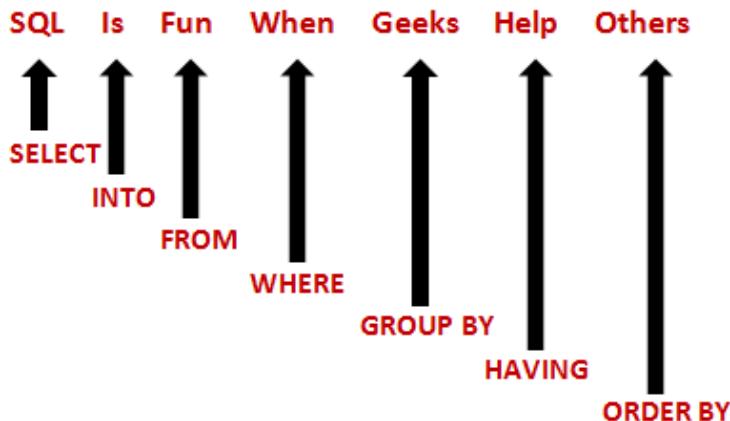
The SELECT Statement and Clauses

The SELECT statement’s purpose is to retrieve (or read) data from the underlying tables (or views). Although it supports multiple clauses, the SELECT statement has only one clause that is required to be specified – the FROM clause. All the remaining clauses, described below, are optional and only used when needed. Note: Not every query needs to have all the clauses specified, but SQL provides

developers and data analysts with a powerful and flexible language to access, manipulate, and display data without the need to write large amounts of code.

During execution, SAS carries out the tasks associated with planning, optimizing, and performing the operations specified in the SELECT statement and its clauses to produce the desired results. To prevent syntax errors from occurring when using the SELECT statement, the clauses must be specified in the correct order. To help you remember the order of the SELECT statement's clauses recite, “**SQL is fun when geeks help others.**” The first letter in each word corresponds to the name of the SELECT statement’s clause as shown in Figure 2.1.

Figure 2.1: Order of the SELECT Statement Clauses



When constructed correctly, the SELECT statement and its clauses declares the database table (or data set) to find the data in, what data to retrieve, and whether any special transformations or processing is needed before the data is returned. The next example shows the correct syntax of a query’s SELECT statement and its clauses.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
    , PRODTYPE
    , PRODCOST
  INTO :M_PRODNAME
    , :M_PRODTYPE
    , :M_PRODCOST
  FROM PRODUCTS
  WHERE PRODNAME CONTAINS "Software"
    GROUP BY PRODTYPE
    HAVING COUNT(PRODTYPE) > 3
    ORDER BY PRODNAME;
QUIT;
```

Results

Product Name	Product Type	Product Cost
Database Software	Software	\$399.00
Graphics Software	Software	\$299.00
Spreadsheet Software	Software	\$299.00
Wordprocessor Software	Software	\$299.00

Now that we've explored the order that each clause is specified in an SQL query, let's examine the order of execution of each clause in an SQL query. Table 2.1 illustrates and describes the execution order of each SELECT statement clause.

Table 2.1: Clause Execution Order

Clause Execution Order	Description
1. FROM Clause	The first clause executed in a query is the FROM clause. It is a required clause with the purpose of determining the working set of data that is being queried (i.e., variable names, variable type, number of rows, and other important information).
2. INTO Clause	The INTO clause is used to create one or more macro variables where the values can be used to manipulate data in DATA and PROC steps.
3. WHERE Clause	The WHERE clause is used to subset rows of data based on the condition(s) specified, and rows that aren't satisfied by the condition(s) are discarded.
4. GROUP BY Clause	The GROUP BY clause takes the rows that were subset with the WHERE clause and grouped based on common values in the column specified in the GROUP BY clause.
5. HAVING Clause	The HAVING clause applies the condition(s) to the grouped rows specified in the GROUP BY clause, and any grouped rows that aren't satisfied by the condition(s) are discarded.
6. SELECT Statement	Expressions specified in the SELECT statement are processed.
7. ORDER BY Clause	The ORDER BY clause sorts the rows of data in either ascending (default) or descending order.

Overview of Data Types

The purpose of a database is to store data. A database contains one or more tables (and other components). Tables consist of columns and rows of data. In the SAS implementation of SQL, the available data types are limited to only two possibilities:

- numeric
- character

Numeric Data

The SAS implementation of SQL provides programmers with numerous arithmetic, statistical, and summary functions. It offers one numeric data type to represent numeric data. Columns defined as a numeric data type with the NUMERIC or NUM column definition are assigned a default length of 8 bytes, even if the column is created with a numeric length of less than 8 bytes. This provides the greatest degree of precision allowed by SAS software. In the example, a table called PURCHASES is created consisting of four numeric columns. The resulting table contains no rows of data, as illustrated by the SAS log results. For more information about the CREATE TABLE statement, see Chapter 5, “Creating, Populating, and Deleting Tables.”

SQL Code

```
PROC SQL;
  CREATE TABLE PURCHASES
    (CUSTNUM  NUM,
     PRODNUM  NUM,
     UNITS    NUM,
     UNITCOST NUM(8,2));
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE PURCHASES
    (CUSTNUM  NUM,
     PRODNUM  NUM,
     UNITS    NUM,
     UNITCOST NUM(8,2));
NOTE: Table PURCHASES created, with 0 rows and 4 columns.
      QUIT;
```

Results

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	CUSTNUM	Num	8
2	PRODNUM	Num	8
4	UNITCOST	Num	8
3	UNITS	Num	8

Creating a numeric column that is less than 8 bytes requires the use of the DATA step LENGTH statement. Although it is not necessary to assign smaller lengths to numeric columns because it can result in precision issues, doing so can make for more efficient use of data storage system resources. The example illustrates a DATA step that assigns smaller lengths to the four numeric variables in the PURCHASES table: CUSTNUM, PRODNUM, UNITS, and UNITCOST. In contrast to the SAS log results produced by the previous PROC SQL code, the CONTENTS output illustrates the creation of a data set with one record and four user-defined, and shorter length, numeric variables.

DATA Step Code

```

DATA PURCHASES;
  LENGTH CUSTNUM    4.
        PRODNUM     3.
        UNITS       3.
        UNITCOST   4.;

  LABEL CUSTNUM = 'Customer Number'
        PRODNUM = 'Product Purchased'
        UNITS = '# Units Purchased'
        UNITCOST = 'Unit Cost';

  FORMAT UNITCOST DOLLAR12.2;

RUN;
PROC CONTENTS DATA=PURCHASES;
RUN;

```

SAS Log Results

```

DATA PURCHASES;
  LENGTH CUSTNUM    4.
        PRODNUM     3.
        UNITS       3.
        UNITCOST   4.;

  LABEL CUSTNUM = 'Customer Number'
        PRODNUM = 'Product Purchased'
        UNITS = '# Units Purchased'
        UNITCOST = 'Unit Cost';

  FORMAT UNITCOST DOLLAR12.2;

RUN;

NOTE: Variable CUSTNUM is uninitialized.
NOTE: Variable PRODNUM is uninitialized.
NOTE: Variable UNITS is uninitialized.
NOTE: Variable UNITCOST is uninitialized.
NOTE: The data set WORK.PURCHASES has 1 observations and 4 variables.
NOTE: DATA statement used:
      real time           2.80 seconds

PROC CONTENTS DATA=PURCHASES;
RUN;

NOTE: PROCEDURE CONTENTS used:
      real time           1.82 seconds

```

CONTENTS Results

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
1	CUSTNUM	Num	4		Customer Number
2	PRODNUM	Num	3		Product Purchased
4	UNITCOST	Num	4	DOLLAR12.2	Unit Cost
3	UNITS	Num	3		# Units Purchased

In the next example, a LENGTH= modifier is illustrated in the PROC SQL code to assign smaller lengths to the four numeric variables (columns) in the PURCHASES table: CUSTNUM, PRODNUM, UNITS, and UNITCOST.

PROC SQL Code

```
PROC SQL;
  CREATE TABLE PURCHASES
    (CUSTNUM NUM LENGTH=4
     LABEL='Customer Number',
     PRODNUM NUM LENGTH=3
     LABEL='Product Purchased',
     UNITS   NUM LENGTH=3
     LABEL='# Units Purchased',
     UNITCOST NUM LENGTH=4
     LABEL='Unit Cost');
QUIT;
```

As shown in the CONTENTS results below, four numeric columns are defined using a LENGTH= column modifier that is smaller than the default length of 8 bytes.

CONTENTS Results

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
1	CUSTNUM	Num	4		Customer Number
2	PRODNUM	Num	3		Product Purchased
4	UNITCOST	Num	4	DOLLAR12.2	Unit Cost
3	UNITS	Num	3		# Units Purchased

Date and Time Column Definitions

Database application processing stores date and time information in the form of a numeric data type. Date and time values are represented internally as an offset where a SAS date value is stored as the number of days from the fixed date value of 01/01/1960 (January 1, 1960). The SAS date value for January 1, 1960, is represented as 0 (zero). A date earlier than this date is represented as a negative number and a date later than this date is represented as a positive number. This makes performing date calculations much easier.

SAS has integrated the vast array of date and time informats and formats with PROC SQL. The various informats and formats act as input and output templates and describe how date and time information is to be read or rendered on output. See the *SAS Language Reference: Dictionary* for detailed descriptions of the various informats and formats and their use. Numeric date and time columns, when combined with informats and/or formats, automatically validate values according to the following rules:

- **Date**—Date informats and formats enable PROC SQL and SAS to determine the month, day, and year values of a date. The month value handles values from 1 through 12. The day value handles values from 1 through 31 and applies additional validations to a maximum of 28, 29, or 30 depending on the month in question. The year value handles values from 1 through

9999. Dates go back to 1582 and ahead to 20,000. When you enter a year value of 0001 and specify a format and a Yearcutoff value of 1920, the returned value would be 2001.

- **Time**—Time informats and formats enable PROC SQL to determine the hour, minute, and second values of a time. The hour portion handles values from 00 through 23. The minute portion handles values from 00 through 59. The second portion handles values from 00 through 59.
- **DATETIME**—Date and time stamps enable the SQL procedure to determine the month, day, and year of a date as well the hour, minute, and second of a time.

See Chapter 5, “Creating, Populating, and Deleting Tables” and Chapter 6, “Modifying and Updating Tables and Indexes” for more information about date and time informats and formats.

Character Data

PROC SQL provides tools to manipulate and store character data including words, text, and codes using the CHARACTER or CHAR data type. The characters allowed by this data type include the ASCII or EBCDIC character sets. The CHARACTER or CHAR data type stores fixed-length character strings consisting of a maximum of 32K characters. If a length is not specified, a CHAR column stores a default of 8 characters.

The SQL programmer has a vast array of SQL and Base SAS functions that can make the task of working with character data considerably easier. In this chapter, you’ll learn how columns based on the character data type are defined, and how string functions, pattern matching, phonetic matching techniques, and a variety of other techniques are used with character data.

Missing Values and NULL

Missing values are an important aspect when dealing with data. The concept of missing values is familiar to programmers, statisticians, researchers, and other SAS users. This chapter describes what NULL values are, what they aren’t, and how they are used.

Missing or unknown information is supported by PROC SQL in a form known as a *null value*. A null value is not the same as a zero value. In SAS, null values are treated as a separate category from known values. A value consisting of zero has a known value. In contrast, a value of null has an unknown quantity and will never be known. For example, a patient who is given an eye exam does not have zero eyesight just because the results from the exam haven’t been received. The correct value to assign in a case like this is a missing or a null value.

In another example, say a person declines to provide their age on a survey. This person’s age is null, not zero. Essentially, this person has an age, but it is unknown. Whenever an unknown value occurs, you have no choice but to assign an unknown value—null.

Because the value of null is unknown, any arithmetic calculation using a null will return a null. This makes a lot of sense because the results of a calculation using a null are not determinable. This is sometimes referred to as the *propagation of nulls* because when a null value is used in a calculation or an expression, it propagates a null value. For example, if a null is added to a known value, then the result is a null value.

Arithmetic and Missing Data

In SAS, a numeric data type containing a null value (absence of any value) is represented with a period (.). This representation indicates that the column has not been assigned a value. A null value has no value and is not the same as zero. A value consisting of zero has a known quantity as opposed to a null value that is not known and never will be known.

If a null value is multiplied with a known value, the result is a null value represented with a period (.). In the next example, when UNITS and UNITCOST both have known values, their product will generate a known value, as illustrated below.

SQL Code

```
PROC SQL;
  SELECT CUSTNUM,
         PRODNUM,
         UNITS,
         UNITCOST,
         UNITS * UNITCOST AS Total FORMAT=DOLLAR12.2
    FROM PURCHASES
   ORDER BY Total;
QUIT;
```

Results

Custnum	Prodnum	Units	Unitcost	Total
501	5004	1	\$299.00	\$299.00
1101	2200	3	\$130.00	\$390.00
901	5004	2	\$299.00	\$598.00
101	5004	2	\$299.00	\$598.00
901	5001	2	\$299.00	\$598.00
201	5001	2	\$299.00	\$598.00
1301	5002	2	\$399.00	\$798.00
201	2102	5	\$175.00	\$875.00
801	2102	5	\$175.00	\$875.00
501	5001	3	\$299.00	\$897.00
701	5004	3	\$299.00	\$897.00
401	5004	3	\$299.00	\$897.00
1301	5001	3	\$299.00	\$897.00
101	5003	3	\$299.00	\$897.00
1401	2102	7	\$175.00	\$1,225.00
301	2102	8	\$175.00	\$1,400.00
501	5003	5	\$299.00	\$1,495.00
1301	5003	5	\$299.00	\$1,495.00
501	2102	9	\$175.00	\$1,575.00
101	2102	9	\$175.00	\$1,575.00

1101	2102	9	\$175.00	\$1,575.00
201	5002	4	\$399.00	\$1,596.00
701	5002	4	\$399.00	\$1,596.00
301	5001	6	\$299.00	\$1,794.00
201	5001	6	\$299.00	\$1,794.00
201	5001	6	\$299.00	\$1,794.00
1301	2102	11	\$175.00	\$1,925.00
401	5002	5	\$399.00	\$1,995.00
901	5002	5	\$399.00	\$1,995.00
101	5001	7	\$299.00	\$2,093.00
401	5003	7	\$299.00	\$2,093.00
501	2102	12	\$175.00	\$2,100.00
901	5003	8	\$299.00	\$2,392.00
701	5003	8	\$299.00	\$2,392.00
201	5003	9	\$299.00	\$2,691.00
901	5001	9	\$299.00	\$2,691.00
1701	1110	1	\$3,200.00	\$3,200.00
701	5001	11	\$299.00	\$3,289.00
401	5001	11	\$299.00	\$3,289.00
701	1110	2	\$3,200.00	\$6,400.00
701	1700	2	\$3,400.00	\$6,800.00
901	1700	2	\$3,400.00	\$6,800.00
1101	1700	2	\$3,400.00	\$6,800.00
901	1200	3	\$3,300.00	\$9,900.00
101	1200	3	\$3,300.00	\$9,900.00
701	1200	3	\$3,300.00	\$9,900.00
201	1700	3	\$3,400.00	\$10,200.00
401	1700	3	\$3,400.00	\$10,200.00
1301	1700	3	\$3,400.00	\$10,200.00
501	1700	4	\$3,400.00	\$13,600.00
1801	1700	4	\$3,400.00	\$13,600.00
901	1110	5	\$3,200.00	\$16,000.00
1001	1700	5	\$3,400.00	\$17,000.00
101	1700	5	\$3,400.00	\$17,000.00
401	1200	6	\$3,300.00	\$19,800.00
1601	1700	7	\$3,400.00	\$23,800.00
1201	1200	8	\$3,300.00	\$26,400.00

SQL Keywords

SQL provides three keywords: AS, DISTINCT, and UNIQUE to perform specific operations on the results. Each will be presented in order as follows.

Creating Column Aliases

In situations where data is computed using system functions, statistical functions, or arithmetic operations, a column name or header can be left blank. To prevent this from occurring, users may specify the AS keyword to provide a name to the column or heading itself along with formatting directions. The next example illustrates using the AS keyword to prevent the name for the computed column from being assigned a temporary column name similar to _TEMAXxx. The name assigned with the AS keyword is also used as the column header on output, as shown below.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST * 0.80 AS Discount_Price FORMAT=DOLLAR9.2
    FROM PRODUCTS
   ORDER BY 3;
QUIT;
```

Results

Product Name	Product Type	Discount_Price
Analog Cell Phone	Phone	\$28.00
Office Phone	Phone	\$104.00
Digital Cell Phone	Phone	\$140.00
Spreadsheet Software	Software	\$239.20
Graphics Software	Software	\$239.20
Wordprocessor Software	Software	\$239.20
Database Software	Software	\$319.20
Dream Machine	Workstation	\$2,560.00
Business Machine	Workstation	\$2,640.00
Travel Laptop	Laptop	\$2,720.00

Finding Duplicate Values

In some situations, several rows in a table may contain identical or duplicate column values. To select only one of each identical or duplicate values, SAS supports and processes the DISTINCT and UNIQUE keywords the same and without any noticeable performance differences. On another note, the ANSI standards support the DISTINCT keyword as the keyword of choice with SELECT statements enabling greater code portability to other databases. The DISTINCT keyword can be used in the SELECT statement as follows.

SQL Code

```
PROC SQL;
  SELECT DISTINCT MANUNUM
    FROM INVENTORY;
QUIT;
```

Results

Manufacturer Number
111
170
500
600

Finding Unique Values

In some situations, several rows in a table will contain identical column values. To select each of these duplicate values only once, the UNIQUE keyword can be used in the SELECT statement.

SQL Code

```
PROC SQL;
  SELECT UNIQUE MANUNUM
    FROM INVENTORY;
QUIT;
```

Results

Manufacturer Number
111
170
500
600

SQL Operators, Functions, and Keywords

SQL programmers have a number of ways to accomplish their objectives, particularly when the goal is to retrieve and work with data. The SELECT statement is an extremely powerful statement in the SQL language. Its syntax can be somewhat complex because of the number of ways that columns, tables, operators, functions, and predicates can be combined into executable statements.

There are several types of operators and functions in PROC SQL:

- comparison operators
- logical operators
- arithmetic operators
- character string operators
- summary functions
- predicates
- keywords

Operators and functions provide value-added features for PROC SQL programmers. Each will be presented below.

Comparison Operators

Comparison operators are used in the SQL procedure to compare one character or numeric value to another. As in the DATA step, SQL comparison operators, mnemonics, and their descriptions appear in the following table.

SAS Operator	Mnemonic Operators	Description
=	EQ	Equal to
^= or !=	NE	Not equal to
<	LT	Less than
<=	LE	Less than or equal to
>	GT	Greater than
>=	GE	Greater than or equal to

Suppose you want to select only those products from the PRODUCTS table that cost more than \$300.00. The example illustrates the use of the greater than sign (>) in a WHERE clause to select products meeting the condition.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE PRODCOST > 300;
QUIT;
```

Results

Product Name	Product Type	Product Cost
Dream Machine	Workstation	\$3,200.00
Business Machine	Workstation	\$3,300.00
Travel Laptop	Laptop	\$3,400.00
Database Software	Software	\$399.00

PROC SQL also supports the use of truncated string comparison operators. These operators work by first truncating the longer string to the same length as the shorter string, and then perform the specified

comparison. The result of using any of the comparison operators has no permanent affect on the strings themselves. The list of truncated string comparison operators and their meanings appear below.

Truncated String Comparison Operator	Description
EQT	Equal to
GTT	Greater than
LT	Less than
GET	Greater than or equal to
LET	Less than or equal to
NET	Not equal to

Logical Operators

Logical operators are used to connect two or more expressions together in a WHERE or HAVING clause. The available logical operators consist of AND, OR, and NOT. Suppose you want to select only those software products that cost more than \$300.00. The example illustrates how the AND operator is used to ensure that both conditions are true.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE PRODTYPE = 'Software' AND
         PRODCOST > 300;
QUIT;
```

Results

Product Name	Product Type	Product Cost
Database Software	Software	\$399.00

The next example illustrates the use of the OR logical operator to select software products or products that cost more than \$300.00.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE PRODTYPE = 'Software' OR
         PRODCOST > 300;
QUIT;
```

Results

Product Name	Product Type	Product Cost
Dream Machine	Workstation	\$3,200.00
Business Machine	Workstation	\$3,300.00
Travel Laptop	Laptop	\$3,400.00
Spreadsheet Software	Software	\$299.00
Database Software	Software	\$399.00
Wordprocessor Software	Software	\$299.00
Graphics Software	Software	\$299.00

The next example illustrates the use of the NOT logical operator to select products that are not software products and do not cost more than \$300.00. Should PRODTYPE contain any value other than “Software,” including a null value, the resulting output would include the row.

QL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE NOT PRODTYPE = 'Software' AND
         NOT PRODCOST > 300;
QUIT;
```

Results

Product Name	Product Type	Product Cost
Analog Cell Phone	Phone	\$35.00
Digital Cell Phone	Phone	\$175.00
Office Phone	Phone	\$130.00

Arithmetic Operators

The arithmetic operators used in PROC SQL are the same operators that are used in the DATA step as well as those found in other languages such as C, Pascal, FORTRAN, and COBOL. The arithmetic operators available in the SQL procedure appear below.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Operator	Description
<code>**</code>	Exponent (raises to a power)
<code>=</code>	Equals

To illustrate how arithmetic operators are used, suppose you want to apply a discount of 20% to the product price (PRODCOST) in the PRODUCTS table. Note that the computed column (PRODCOST * 0.80) does not automatically create a column header on output.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST * 0.80
    FROM PRODUCTS;
QUIT;
```

Results

Product Name	Product Type	
Dream Machine	Workstation	2560
Business Machine	Workstation	2640
Travel Laptop	Laptop	2720
Analog Cell Phone	Phone	28
Digital Cell Phone	Phone	140
Office Phone	Phone	104
Spreadsheet Software	Software	239.2
Database Software	Software	319.2
Wordprocessor Software	Software	239.2
Graphics Software	Software	239.2

In the next example, suppose you wanted to reference a column that was calculated in the SELECT statement. PROC SQL allows references to a computed column in the same SELECT statement (or a WHERE clause) using the CALCULATED keyword. Note that the computed columns have column aliases created for them using the AS keyword. If the CALCULATED keyword were not specified preceding the calculated column, an error would have been generated. The results were produced in ascending order by the discounted price.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST * 0.80 AS DISCOUNT_PRICE
                           FORMAT=DOLLAR9.2,
```

```

PROCOST = CALCULATED DISCOUNT_PRICE AS LOSS
      FORMAT=DOLLAR7.2
      FROM PRODUCTS
      ORDER BY 3;
QUIT;

```

Results

Product Name	Product Type	DISCOUNT_PRICE	LOSS
Analog Cell Phone	Phone	\$28.00	\$7.00
Office Phone	Phone	\$104.00	\$26.00
Digital Cell Phone	Phone	\$140.00	\$35.00
Spreadsheet Software	Software	\$239.20	\$59.80
Graphics Software	Software	\$239.20	\$59.80
Wordprocessor Software	Software	\$239.20	\$59.80
Database Software	Software	\$319.20	\$79.80
Dream Machine	Workstation	\$2,560.00	\$640.00
Business Machine	Workstation	\$2,640.00	\$660.00
Travel Laptop	Laptop	\$2,720.00	\$680.00

Character String Operators and Functions

Character string operators and functions are typically used with character data. Numerous operators are presented to acquaint programmers with the power available with in SQL procedure. As you become familiar with each operator, you'll find their real strength as you begin to nest functions within each other.

Concatenating Strings

The following example illustrates a basic concatenation operator that is used to concatenate two columns and a text string. Note that the created column is without a name and has a total length of 23 characters. For more details and special formatting considerations, the concatenation operator “||” will be discussed in greater detail in Chapter 3, “Formatting Output.”

SQL Code

```

PROC SQL;
  SELECT MANUCITY || "," || MANUSTAT
    FROM MANUFACTURERS;
QUIT;

```

Results

Houston ,TX
San Diego ,CA
Miami ,FL
San Mateo ,CA
San Diego ,CA
San Diego ,CA

Two other effective methods of concatenating columns and/or text strings in SQL operations is to use the special concatenation functions, CAT or CATS. The next example illustrates a CAT function being used to concatenate two columns and a text string. The CAT function does not remove leading and trailing blanks, and returns a maximum value of 200 characters in a concatenated character string in PROC SQL.

SQL Code

```
PROC SQL;
  SELECT CAT(MANUCITY," ",MANUSTAT)
    FROM MANUFACTURERS;
QUIT;
```

Results

Houston ,TX
San Diego ,CA
Miami ,FL
San Mateo ,CA
San Diego ,CA
San Diego ,CA

The CATS function can also be used in PROC SQL. Its purpose is to remove leading and trailing blanks, and return a maximum value of 200 characters in a concatenated character string in PROC SQL.

Finding the Length of a String

The LENGTH function is used to obtain the length of a character string column. LENGTH returns a number equal to the number of characters in the argument. Note that the computed column (LENGTH(ProdName)) has a column header created for it called Length by specifying the AS keyword. This example illustrates using the LENGTH function to determine the length of data values.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         PRODNAME,
         LENGTH(ProdName) AS Length
    FROM PRODUCTS;
QUIT;
```

Results

Product Number	Product Name	Length
1110	Dream Machine	13
1200	Business Machine	16
1700	Travel Laptop	13
2101	Analog Cell Phone	17
2102	Digital Cell Phone	18
2200	Office Phone	12
5001	Spreadsheet Software	20
5002	Database Software	17
5003	Wordprocessor Software	22
5004	Graphics Software	17

Combining Functions and Operators

As in the DATA step, many functions can be used in the SQL procedure. To modify one or more existing rows in a table, the UPDATE statement is used (see Chapter 6, “Modifying and Updating Tables and Indexes,” for more details). The UPDATE statement with the SET clause changes the contents of a data value (functioning the same way as a DATA step assignment statement) by assigning a new value to the column identified to the left of the equal sign by a constant or expression referenced to the right of the equal sign.

The UPDATE statement does not automatically produce any output except for the log messages that are based on the operation results itself. To illustrate the use of DATA step functions and operators in the SQL procedure, the next example shows a SCAN function that isolates the first piece of information from product name (PRODNAME), a TRIM function to remove trailing blanks from product type (PRODTYPE), and a concatenation operator “||” that concatenates the first character expression with the second expression. It should be noted that care should be exercised when using the SCAN function because it returns a 200-byte string.

SQL Code

```
PROC SQL;
  UPDATE PRODUCTS
    SET PRODNAME = SCAN(PRODNAME,1) || TRIM(PRODTYPE);
QUIT;
```

SAS Log Results

```
PROC SQL;
  UPDATE PRODUCTS
    SET PRODNAME = SCAN(PRODNAME,1) || TRIM(PRODTYPE);
NOTE: 10 rows were updated in PRODUCTS.
QUIT;
```

An optional WHERE clause can be specified to limit the number of rows to which modifications will be applied. The next example illustrates using a WHERE clause to restrict the number of rows that are updated in the previous example to just “phone,” excluding all the other rows.

SQL Code

```
PROC SQL;
  UPDATE PRODUCTS
    SET PRODNAME = SCAN(PRODNAME,1) || TRIM(PRODTYPE)
      WHERE PRODTYPE IN ('Phone');
QUIT;
```

SAS Log Results

```
PROC SQL;
  UPDATE PRODUCTS
    SET PRODNAME = SCAN(PRODNAME,1) || TRIM(PRODTYPE)
      WHERE PRODTYPE IN ('Phone');
NOTE: 3 rows were updated in PRODUCTS.
QUIT;
```

Aligning Characters

The default alignment for character data is to the left; however, character columns or expressions can also be aligned to the right. Two functions are available for character alignment: LEFT and RIGHT. The next example combines the concatenation operator “||” and the TRIM function with the LEFT function to left align a character expression while inserting a comma “,” and blank between the columns.

SQL Code

```
PROC SQL;
  SELECT LEFT(TRIM(MANUCITY) || ", " || MANUSTAT)
    FROM MANUFACTURERS;
QUIT;
```

Results

Houston, TX
San Diego, CA
Miami, FL
San Mateo, CA
San Diego, CA
San Diego, CA

The next example illustrates how character data can be right aligned using the RIGHT function.

SQL Code

```
PROC SQL;
  SELECT RIGHT(MANUCITY)
    FROM MANUFACTURERS;
QUIT;
```

Results

Houston
San Diego
Miami
San Mateo
San Diego
San Diego

Finding the Occurrence of a Pattern with INDEX

To find the occurrence of a pattern, the INDEX function can be used. Frequently, requirements call for a column to be searched using a specific character string. The INDEX function can be used in the SQL procedure to search for patterns in a character string. The character string is searched from left to right for the first occurrence of the specified value. If the desired string is found, the column position of the first character is returned. Otherwise, a value of zero (0) is returned. The following arguments are used to search for patterns in a column: the character column or expression, and the character string to search for. To find all products with the characters “phone” in the product name (PRODNAME) column, the following code can be specified:

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         PRODNAME,
         PRODTYPE
    FROM PRODUCTS
   WHERE INDEX(PRODNAME, 'phone') > 0;
QUIT;
```

SAS Log Results

```
PROC SQL;
  SELECT PRODNUM,
         PRODNAME,
         PRODTYPE
    FROM PRODUCTS
   WHERE INDEX(PRODNAME, 'phone') > 0;
NOTE: No rows were selected.
QUIT;
```

Analysis

As in the DATA step, no rows were selected because the search is case sensitive and “phone” is specified as all lowercase characters.

Changing the Case in a String

SAS provides two functions that enable you to change the case of a string’s characters: LOWCASE and UPCASE. The LOWCASE function converts all of the characters in a string or expression to lowercase characters. The UPCASE function converts all of the characters in a string or expression to uppercase characters.

Getting back to the previous example, the results of the search were negative even though the character string “phone” appeared multiple times in more than one row. In order to make this search recognize all the possible lower- and uppercase variations of the word “phone,” the search criteria in the WHERE clause could be made “smarter” by combining an UPCASE function with the INDEX function as follows.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         PRODNAME,
         PRODTYPE
    FROM PRODUCTS
   WHERE INDEX(UPCASE(PRODNAME), 'PHONE') > 0;
QUIT;
```

Results

Product Number	Product Name	Product Type
2101	AnalogPhone	Phone
2102	DigitalPhone	Phone
2200	OfficePhone	Phone

In the next example, the LOWCASE function is combined with the INDEX function to produce the identical output from the previous example.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         PRODNAME,
         PRODTYPE
    FROM PRODUCTS
   WHERE INDEX(LOWCASE(PRODNAME), 'phone') > 0;
QUIT;
```

Results

Product Number	Product Name	Product Type
2101	AnalogPhone	Phone
2102	DigitalPhone	Phone
2200	OfficePhone	Phone

Extracting Information from a String

Occasionally, processing requirements call for specific pieces of information to be extracted from a column. In these situations the SUBSTR function can be used with a character column by specifying a starting position and the number of characters to extract. The following example illustrates how the SUBSTR function is used to capture the first 4 bytes from the product type (PRODTYPE) column.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         PRODNAME,
         PRODTYPE,
         SUBSTR(PRODTYPE,1,4)
    FROM PRODUCTS
   WHERE PRODCOST > 100.00;
QUIT;
```

Results

Product Number	Product Name	Product Type	
1110	DreamWorkstation	Workstation	Work
1200	BusinessWorkstation	Workstation	Work
1700	TravelLaptop	Laptop	Lapt
2102	DigitalPhone	Phone	Phon
2200	OfficePhone	Phone	Phon
5001	SpreadsheetSoftware	Software	Soft
5002	DatabaseSoftware	Software	Soft
5003	WordprocessorSoftware	Software	Soft
5004	GraphicsSoftware	Software	Soft

Phonetic Matching (Sounds-Like Operator =*)

A technique for finding names that sound alike or have spelling variations is available in the SQL procedure. This frequently used technique is referred to as phonetic matching and is performed using the Soundex algorithm. In Joe Celko's popular book *SQL for Smarties: Advanced SQL Programming, Third Edition* (Morgan Kaufman, 2005, page 176), he traced the origins of the Soundex algorithm to the developers Margaret O'Dell and Robert C. Russell in 1918. Developed before the first computer, clerks often used the algorithm to manually search for similar sounding names.

Although not technically a function, the sounds-like operator “=**” searches and selects character data based on two expressions: the search value and the matched value. Anyone that has looked for a last name in a local telephone directory is quickly reminded of the possible phonetic variations.

To illustrate how the sounds-like operator works, we will search on CUSTNAME in the CUSTOMERS2 table. The CUSTOMERS2 table is illustrated below. Although each name has phonetic variations and sounds the same, the results of “Laughler,” “Loffler,” and “Laffler” are spelled differently (illustrated below). The following PROC SQL code uses the sounds-like operator to find all customers that sound like “Lafler.”

CUSTOMERS2 Table

CUSTNUM	CUSTNAME	CUSTCITY
1	Smith	San Diego
7	Lafler	Spring Valley
11	Jones	Carmel
13	Thompson	Miami
7	Loffler	Spring Valley
1	Smithe	San Diego
7	Laughler	Spring Valley
7	Laffler	Spring Valley

SQL Code

```
PROC SQL;
  SELECT CUSTNUM,
         CUSTNAME,
         CUSTCITY
    FROM CUSTOMERS2
   WHERE CUSTNAME =* 'Lafler';
QUIT;
```

Results

Customer Number	Customer Name	Customer's Home City
7	Lafler	Spring Valley
7	Loffler	Spring Valley
7	Laffler	Spring Valley

Readers familiar with the DATA step SOUNDDEX(<argument>) function to search a string are cautioned that it cannot be used in an SQL WHERE clause. Instead, the sounds-like operator “=*” must be specified; otherwise, a result of no rows will be selected.

Notice that only three of the four possible phonetic matches were selected in the preceding example (i.e., Lafler, Loffler, and Laffler). The fourth possibility, “Laughler” was not chosen as a “matched” value in the search by the sounds-like algorithm. In an attempt to overcome the inherent limitation with the sounds-like operator, as described in Joe Celko’s popular book *SQL for Smarties: Advanced SQL Programming, Third Edition* (Morgan Kaufman, 2005), and to derive a broader list of “matched” values, programmers should make every attempt to develop a comprehensive list of search values to widen the scope of possibilities. We can expand our original search criteria in the previous example to include the missing possibilities using OR logic.

SQL Code

```
PROC SQL;
  SELECT CUSTNUM,
         CUSTNAME,
         CUSTCITY
    FROM CUSTOMERS2
   WHERE CUSTNAME =* 'Lafler'      OR
         CUSTNAME =* 'Laughler'    OR
         CUSTNAME =* 'Lasler';
QUIT;
```

Results

Customer Number	Customer Name	Customer's Home City
7	Lafler	Spring Valley
7	Loffler	Spring Valley
7	Laughler	Spring Valley
7	Laffler	Spring Valley

Finding the First Nonmissing Value

The first example provides a way to find the first non-missing value in a column or list. Specified in a SELECT statement, the COALESCE function inspects a column, or in the case of a list scans the arguments from left to right, and returns the first non-missing or non-null value. If all values are missing, the result is missing. To take advantage of the COALESCE function, all arguments must be of the same data type. The next example illustrates one approach on computing the total cost for each product purchased from the number of units and unit costs columns in the PURCHASES table. If either the UNITS column or the UNITCOST column contains a missing value, a zero is assigned by the programmer to prevent the propagation of missing values.

SQL Code

```
PROC SQL;
  SELECT CUSTNUM,
         PRODNUM,
         UNITS,
         UNITCOST,
         (COALESCE(UNITS, 0) * COALESCE(UNITCOST, 0))
           AS TOTCOST FORMAT=DOLLAR10.2
    FROM PURCHASES;
QUIT;
```

Results

Custnum	Prodnum	Units	Unitcost	TOTCOST
1701	1110	1	\$3,200.00	\$3,200.00
101	5001	7	\$299.00	\$2,093.00
701	5001	11	\$299.00	\$3,289.00
701	5003	8	\$299.00	\$2,392.00
701	5002	4	\$399.00	\$1,596.00
701	5004	3	\$299.00	\$897.00
701	1700	2	\$3,400.00	\$6,800.00
701	1200	3	\$3,300.00	\$9,900.00
701	1110	2	\$3,200.00	\$6,400.00
1301	5001	3	\$299.00	\$897.00
1301	5003	5	\$299.00	\$1,495.00
1301	5002	2	\$399.00	\$798.00
901	1700	2	\$3,400.00	\$6,800.00
901	1200	3	\$3,300.00	\$9,900.00
901	1110	5	\$3,200.00	\$16,000.00
901	5001	9	\$299.00	\$2,691.00
901	5002	5	\$399.00	\$1,995.00
901	5003	8	\$299.00	\$2,392.00
901	5004	2	\$299.00	\$598.00
401	5001	11	\$299.00	\$3,289.00

401	5002	5	\$399.00	\$1,995.00
401	5003	7	\$299.00	\$2,093.00
401	5004	3	\$299.00	\$897.00
401	1700	3	\$3,400.00	\$10,200.00
401	1200	6	\$3,300.00	\$19,800.00
201	5001	6	\$299.00	\$1,794.00
201	5001	6	\$299.00	\$1,794.00
201	5003	9	\$299.00	\$2,691.00
201	5002	4	\$399.00	\$1,596.00
201	1700	3	\$3,400.00	\$10,200.00
901	5001	2	\$299.00	\$598.00
201	5001	2	\$299.00	\$598.00
201	2102	5	\$175.00	\$875.00
1101	2102	9	\$175.00	\$1,575.00
1301	2102	11	\$175.00	\$1,925.00
1401	2102	7	\$175.00	\$1,225.00
801	2102	5	\$175.00	\$875.00
501	2102	12	\$175.00	\$2,100.00
301	2102	8	\$175.00	\$1,400.00
1101	2200	3	\$130.00	\$390.00
101	2102	9	\$175.00	\$1,575.00
101	5003	3	\$299.00	\$897.00
101	5004	2	\$299.00	\$598.00
101	1200	3	\$3,300.00	\$9,900.00
101	1700	5	\$3,400.00	\$17,000.00
1301	1700	3	\$3,400.00	\$10,200.00
1601	1700	7	\$3,400.00	\$23,800.00
1801	1700	4	\$3,400.00	\$13,600.00
1001	1700	5	\$3,400.00	\$17,000.00
1101	1700	2	\$3,400.00	\$6,800.00
1201	1200	8	\$3,300.00	\$26,400.00
501	5001	3	\$299.00	\$897.00
501	5003	5	\$299.00	\$1,495.00
501	5004	1	\$299.00	\$299.00
501	1700	4	\$3,400.00	\$13,600.00
301	5001	6	\$299.00	\$1,794.00
501	2102	9	\$175.00	\$1,575.00

Producing a Row Number

A unique undocumented, unsupported feature for producing a row (observation) count can be obtained with the MONOTONIC() function. Similar to the row numbers produced and displayed in output from the PRINT procedure (without the NOOBS option specified), the MONOTONIC() function displays row numbers, too. The MONOTONIC() function automatically creates a column (variable) in the output results or in a new table. Because this is an undocumented feature and is not supported in the SQL procedure, users are cautioned to use care when using the MONOTONIC() function because it is possible to obtain duplicate or missing values. The next example illustrates the creation of a row number using the MONOTONIC() function in a SELECT statement.

SQL Code

```
PROC SQL;
  SELECT MONOTONIC() AS Row_Number FORMAT=COMMA6.,
         PRODNUM,
         UNITS,
         UNITCOST
    FROM PURCHASES;
QUIT;
```

Results

Row_Number	Prodnum	Units	Unitcost
1	1110	1	\$3,200.00
2	5001	7	\$299.00
3	5001	11	\$299.00
4	5003	8	\$299.00
5	5002	4	\$399.00
6	5004	3	\$299.00
7	1700	2	\$3,400.00
8	1200	3	\$3,300.00
9	1110	2	\$3,200.00
10	5001	3	\$299.00
11	5003	5	\$299.00
12	5002	2	\$399.00
13	1700	2	\$3,400.00
14	1200	3	\$3,300.00
15	1110	5	\$3,200.00
16	5001	9	\$299.00
17	5002	5	\$399.00
18	5003	8	\$299.00
19	5004	2	\$299.00
20	5001	11	\$299.00

21	5002	5	\$399.00
22	5003	7	\$299.00
23	5004	3	\$299.00
24	1700	3	\$3,400.00
25	1200	6	\$3,300.00
26	5001	6	\$299.00
27	5001	6	\$299.00
28	5003	9	\$299.00
29	5002	4	\$399.00
30	1700	3	\$3,400.00
31	5001	2	\$299.00
32	5001	2	\$299.00
33	2102	5	\$175.00
34	2102	9	\$175.00
35	2102	11	\$175.00
36	2102	7	\$175.00
37	2102	5	\$175.00
38	2102	12	\$175.00
39	2102	8	\$175.00
40	2200	3	\$130.00
41	2102	9	\$175.00
42	5003	3	\$299.00
43	5004	2	\$299.00
44	1200	3	\$3,300.00
45	1700	5	\$3,400.00
46	1700	3	\$3,400.00
47	1700	7	\$3,400.00
48	1700	4	\$3,400.00
49	1700	5	\$3,400.00
50	1700	2	\$3,400.00
51	1200	8	\$3,300.00
52	5001	3	\$299.00
53	5003	5	\$299.00
54	5004	1	\$299.00
55	1700	4	\$3,400.00
56	5001	6	\$299.00
57	2102	9	\$175.00

A row number can also be produced with the documented and supported SQL procedure option, NUMBER. Unlike the MONOTONIC() function, the NUMBER option does not create a new column in a new table. The NUMBER option is illustrated below.

SQL Code

```
PROC SQL NUMBER;
  SELECT PRODNUM,
         UNITS,
         UNITCOST
    FROM PURCHASES;
QUIT;
```

Results

Row	Prodnum	Units	Unitcost
1	1110	1	\$3,200.00
2	5001	7	\$299.00
3	5001	11	\$299.00
4	5003	8	\$299.00
5	5002	4	\$399.00
6	5004	3	\$299.00
7	1700	2	\$3,400.00
8	1200	3	\$3,300.00
9	1110	2	\$3,200.00
10	5001	3	\$299.00
11	5003	5	\$299.00
12	5002	2	\$399.00
13	1700	2	\$3,400.00
14	1200	3	\$3,300.00
15	1110	5	\$3,200.00
16	5001	9	\$299.00
17	5002	5	\$399.00
18	5003	8	\$299.00
19	5004	2	\$299.00
20	5001	11	\$299.00

21	5002	5	\$399.00
22	5003	7	\$299.00
23	5004	3	\$299.00
24	1700	3	\$3,400.00
25	1200	6	\$3,300.00
26	5001	6	\$299.00
27	5001	6	\$299.00
28	5003	9	\$299.00
29	5002	4	\$399.00
30	1700	3	\$3,400.00
31	5001	2	\$299.00
32	5001	2	\$299.00
33	2102	5	\$175.00
34	2102	9	\$175.00
35	2102	11	\$175.00
36	2102	7	\$175.00
37	2102	5	\$175.00
38	2102	12	\$175.00
39	2102	8	\$175.00
40	2200	3	\$130.00
41	2102	9	\$175.00

42	5003	3	\$299.00
43	5004	2	\$299.00
44	1200	3	\$3,300.00
45	1700	5	\$3,400.00
46	1700	3	\$3,400.00
47	1700	7	\$3,400.00
48	1700	4	\$3,400.00
49	1700	5	\$3,400.00
50	1700	2	\$3,400.00
51	1200	8	\$3,300.00
52	5001	3	\$299.00
53	5003	5	\$299.00
54	5004	1	\$299.00
55	1700	4	\$3,400.00
56	5001	6	\$299.00
57	2102	9	\$175.00

Summarizing Data

The SQL procedure is a wonderful tool for summarizing (or aggregating) data. It provides a number of useful summary (or aggregate) functions to help perform calculations, descriptive statistics, and other aggregating operations in a SELECT statement or HAVING clause. These functions are designed to summarize information and not display detail about data.

Without the availability of summary functions, you would have to construct the necessary logic using somewhat complicated SQL programming constructs. When using a summary function without a GROUP BY clause, (see Chapter 3, “Formatting Output”), all the rows in a table are treated as a single group. Consequently, the results are often a single row value.

A number of summary functions are available including facilities to count non-missing values; determine the minimum and maximum values in specific columns; return the range of values; compute the mean, standard deviation, and variance of specific values; and other aggregating functions. In the following table, an alphabetical listing of the available summary functions is displayed and, when multiple names for the same function are available, the ANSI-approved name appears first.

Summary Function	Description
AVG, MEAN	Average or mean of values
COUNT, FREQ, N	Aggregate number of non-missing values
CSS	Corrected sum of squares

Summary Function	Description
CV	Coefficient of variation
MAX	Largest value
MIN	Smallest value
NMISS	Number of missing values
PRT	Probability of a greater absolute value of Student's t
RANGE	Difference between the largest and smallest values
STD	Standard deviation
STDERR	Standard error of the mean
SUM	Sum of values
SUMWGT	Sum of the weight variable values which is 1
T	Testing the hypothesis that the population mean is zero
USS	Uncorrected sum of squares
VAR	Variance

The next example uses the COUNT function with the (*) argument to produce a total number of rows, whether data is missing or not. The asterisk (*) is specified as the argument to the COUNT function to count all rows in the PURCHASES table.

SQL Code

```
PROC SQL;
  SELECT COUNT(*) AS Row_Count
    FROM PURCHASES;
QUIT;
```

Results

Row_Count
57

Unlike the COUNT(*) function syntax that counts all rows, whether data is missing or not, the next example uses the COUNT function with the (column-name) argument to produce a total number of non-missing rows based on the UNITS column.

SQL Code

```
PROC SQL;
  SELECT COUNT(UNITS) AS Non_Missing_Row_Count
    FROM PURCHASES;
QUIT;
```

Results

Non_Missing_Row_Count
57

The MIN summary function can be specified to determine what the least expensive product is in the PRODUCTS table.

SQL Code

```
PROC SQL;
  SELECT MIN(prodcost) AS Cheapest
    Format=dollar9.2 Label='Least Expensive'
   FROM PRODUCTS;
QUIT;
```

Results

Least Expensive
\$35.00

In the next example, the SUM function is specified to sum numeric data types for a selected column. Suppose you want to determine the total costs of all purchases by customers who bought workstations (PRODNUM=1110 and 1200) and laptops (PRODNUM=1700). You could construct the following query to sum all non-missing values for customers who purchased workstations and laptops in the PURCHASES table.

SQL Code

```
PROC SQL;
  SELECT SUM((UNITS) * (UNITCOST))
    AS Total_Purchases FORMAT=DOLLAR12.2
   FROM PURCHASES
  WHERE PRODNUM = 1110 OR
        PRODNUM = 1200 OR
        PRODNUM = 1700;
QUIT;
```

Results

Total_Purchases
\$237,500.00

Data can also be summarized down rows (observations) as well as across columns (variables). This flexibility gives SAS users an incredible range of power, and the ability to take advantage of several summary functions that are supplied (or built-in) by SAS. These techniques permit the average of quantities rather than the set of all quantities. Without the ability to summarize data in PROC SQL, users would be forced to write complicated formulas and/or routines, or even write and test DATA step

programs to summarize data. Two examples will be illustrated to show how SQL can be constructed to summarize data:

- Summarizing data down rows
- Summarizing data across columns

Summarizing Data Down Rows

The SQL procedure can be used to produce a single aggregate value by summarizing data down rows (or observations). The advantages of using a summary function in PROC SQL is that it will generally compute the aggregate quicker than if a user-defined equation were constructed, and it saves the effort of having to construct and test a program containing the user-defined equation in the first place. Suppose you want to know the average product cost for all software in the PRODUCTS table containing a variety of products. The following query computes the average product cost and produces a single aggregate value using the AVG function.

SQL Code

```
PROC SQL;
  SELECT AVG(PRODCOST) AS
    AVERAGE_PRODUCT_COST FORMAT=DOLLAR10.2
  FROM PRODUCTS
  WHERE UPCASE(PRODTYPE) IN
    ("SOFTWARE");
QUIT;
```

Results

AVERAGE_PRODUCT_COST
\$324.00

Summarizing Data Across Columns

When a computation is needed on two or more columns in a row, the SQL procedure can be used to summarize data across columns. Suppose you want to know the average cost of products in inventory. The next example computes the average inventory cost for each product without using a summary function, and once computed displays the value for each row as AVERAGE_PRICE.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
    (INVPRICE / INVQTY) AS
      AVERAGE_PRICE
    FORMAT=DOLLAR8.2
  FROM INVOICE;
QUIT;
```

Results

Product Number	AVERAGE_PRICE
5001	\$299.00
6001	\$799.00
2101	\$35.00
1110	\$3200.00
5002	\$399.00
6000	\$99.00
1200	\$3300.00

Predicates

Predicates are used in PROC SQL to perform direct comparisons between two conditions or expressions. Six predicates will be looked at:

- BETWEEN
- IN
- IS NULL
- IS MISSING
- LIKE
- EXISTS

Selecting a Range of Values

The BETWEEN predicate is a way of simplifying a query by selecting column values within a designated range of values. BETWEEN is equivalent to an AND combination of one LE (less than or equal) and one GE (greater than or equal) conditions; inclusive of both endpoints. It is extremely flexible because it works with character, numeric, and date values. Programmers can also combine two or more BETWEEN predicates with AND or OR operators for more complicated conditions. In the next example, a range of products costing between \$200.00 and \$500.00 inclusively are selected from the PRODUCTS table.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE PRODCOST BETWEEN 200 AND 500;
QUIT;
```

Results

Product Name	Product Type	Product Cost
SpreadsheetSoftware	Software	\$299.00
DatabaseSoftware	Software	\$399.00
WordprocessorSoftware	Software	\$299.00
GraphicsSoftware	Software	\$299.00

In the next example, products are selected from the INVENTORY table that were ordered between the years 1999 and 2000. The YEAR function returns the year portion from a SAS date value and is used as the range of values in the WHERE clause.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         ORDDATE
    FROM INVENTORY
   WHERE YEAR(ORDDATE) BETWEEN 1999 AND 2000;
QUIT;
```

Results

Product Number	Inventory Quantity	Date Inventory Last Ordered
1110	20	09/01/2000
1700	10	08/15/2000
5001	5	08/15/2000
5002	3	08/15/2000
5003	10	08/15/2000
5004	20	09/01/2000
5001	2	09/01/2000

The BETWEEN predicate and OR operator are used together in the next example to select products ordered between 1999 and 2000 or where inventory quantities are greater than 15. The YEAR function returns the year portion from a SAS date value and is used as the range of values in the WHERE clause.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         ORDDATE
    FROM INVENTORY
```

```

WHERE (YEAR(ORDDATE) BETWEEN 1999 AND 2000) OR
INVENQTY > 15;
QUIT;

```

Results

Product Number	Inventory Quantity	Date Inventory Last Ordered
1110	20	09/01/2000
1700	10	08/15/2000
5001	5	08/15/2000
5002	3	08/15/2000
5003	10	08/15/2000
5004	20	09/01/2000
5001	2	09/01/2000

Selecting Nonconsecutive Values

The IN predicate selects one or more rows based on the matching of one or more column values in a set of values. The IN predicate creates an OR condition between each value and returns a Boolean value of True if a column value is equal to one or more of the values in the expression list. Although the IN predicate can be specified with single column values, it may be less costly to specify the “=” sign instead. The “=” sign is used in the next example rather than the IN predicate to select phones from the PRODUCTS table.

SQL Code

```

PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'PHONE';
QUIT;

```

Results

Product Name	Product Type	Product Cost
AnalogPhone	Phone	\$35.00
DigitalPhone	Phone	\$175.00
OfficePhone	Phone	\$130.00

In the next example, both phones and software products are selected from the PRODUCTS table. To avoid having to specify two OR conditions, the IN predicate is specified.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) IN ('PHONE', 'SOFTWARE');
QUIT;
```

Results

Product Name	Product Type	Product Cost
AnalogPhone	Phone	\$35.00
DigitalPhone	Phone	\$175.00
OfficePhone	Phone	\$130.00
SpreadsheetSoftware	Software	\$299.00
DatabaseSoftware	Software	\$399.00
WordprocessorSoftware	Software	\$299.00
GraphicsSoftware	Software	\$299.00

Testing for NULL or MISSING Values

The IS NULL predicate is the ANSI approach of selecting one or more rows by evaluating whether a column value is missing or null (see the “Missing Values and Null section”). The next example selects products from the INVENTORY table that are out-of-stock in inventory.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         INVENCST
    FROM INVENTORY
   WHERE INVENQTY IS NULL;
QUIT;
```

SAS Log Results

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         INVENCST
    FROM INVENTORY
   WHERE INVENQTY IS NULL;
NOTE: No rows were selected.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.05 seconds
```

The next example selects products from the INVENTORY table that are currently stocked in inventory. Note that the predicates NOT IS NULL or IS NOT NULL can be specified to produce the same results.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         INVENCST
    FROM INVENTORY
   WHERE INVENQTY IS NOT NULL;
QUIT;
```

Results

Product Number	Inventory Quantity	Inventory Cost
1110	20	\$45,000.00
1700	10	\$28,000.00
5001	5	\$1,000.00
5002	3	\$900.00
5003	10	\$2,000.00
5004	20	\$1,400.00
5001	2	\$1,200.00

The IS MISSING predicate performs identically to the IS NULL predicate by selecting one or more rows containing a missing value (null). The only difference is that specifying IS NULL is the ANSI standard way of expressing the predicate and IS MISSING is commonly used in SAS. The next example uses the IS MISSING with the NOT predicate to select products from the INVENTORY table that are stocked in inventory.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         INVENCST
    FROM INVENTORY
   WHERE INVENQTY IS NOT MISSING;
QUIT;
```

Finding Patterns in a String (Pattern Matching % and _)

Constructing specific search patterns in string expressions is a simple process with the LIKE predicate. The % (percent sign) acts as a wildcard character representing any number of characters, including any combination of upper or lower case characters. Combining the LIKE predicate with the % permits case-sensitive searches and is a popular technique used by savvy SQL programmers to find patterns in their data.

60 PROC SQL: Beyond the Basics Using SAS, Third Edition

Using the LIKE operator with the % provides a wildcard capability enabling the selection of table rows that match a specific pattern. The LIKE predicate is case-sensitive and should be used with care. To find patterns in product name (PRODNAME) containing the uppercase character “A” in the first position followed by any number of characters is specified with the following WHERE clause.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
    FROM PRODUCTS
   WHERE PRODNAME LIKE 'A%';
QUIT;
```

Results

Product Name
Analog Cell Phone

The next example illustrates the wildcard character “%” preceding and following the search word to select all products whose name contains the word “Soft” in its name. The resulting output contains product types such as “Software” and any other products containing the word “Soft”.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE PRODTYPE LIKE '%Soft%';
QUIT;
```

Results

Product Name	Product Type	Product Cost
SpreadsheetSoftware	Software	\$299.00
DatabaseSoftware	Software	\$399.00
WordprocessorSoftware	Software	\$299.00
GraphicsSoftware	Software	\$299.00

In the next example, the LIKE predicate is used to check a column for the existence of trailing blanks. The wildcard character % followed by a blank space is specified as the search argument.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
    FROM PRODUCTS
   WHERE PRODNAME LIKE ' % ';
QUIT;
```

Results

Product Name
DreamWorkstation
BusinessWorkstation
TravelLaptop
AnalogPhone
DigitalPhone
OfficePhone
SpreadsheetSoftware
DatabaseSoftware
WordprocessorSoftware
GraphicsSoftware

When a pattern search for a specific number of characters is needed, using the LIKE predicate with the underscore (_) provides a way to pattern match character-by-character. Thus, a single underscore (_) in a specific position acts as a wildcard placement holder for that position only. Two consecutive underscores (_) act as a wildcard placement holder for those two positions. Three consecutive underscores act as a wildcard placement holder for those three positions. And so forth. In the next example, the first position used to search product type contains the character “P”, and the next five positions (represented with five underscores) act as a placeholder for any value.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) LIKE 'P_____';
QUIT;
```

Results

Product Name	Product Type	Product Cost
AnalogPhone	Phone	\$35.00
DigitalPhone	Phone	\$175.00
OfficePhone	Phone	\$130.00

The next example illustrates a pattern search of product name (PRODNAME) where the first three positions are represented as a wildcard; the fourth position contains the lowercase character “a”, followed by any combination of uppercase or lowercase characters.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
    FROM PRODUCTS
   WHERE PRODNAME LIKE  '___a%';
QUIT;
```

Results

Product Name
Dream Machine
Database Software

Testing for the Existence of a Value

The EXISTS predicate is used to test for the existence of a set of values. In the next example, a subquery is used to check for the existence of customers in the CUSTOMERS table with purchases from the PURCHASES table. More details on subqueries will be presented in Chapter 7, “Coding Complex Queries.”

SQL Code

```
PROC SQL;
  SELECT CUSTNUM,
         CUSTNAME,
         CUSTCITY
    FROM CUSTOMERS C
   WHERE EXISTS
     (SELECT *
      FROM PURCHASES P
     WHERE C.CUSTNUM = P.CUSTNUM);
QUIT;
```

Results

Customer Number	Customer Name	Customer's Home City
101	La Mesa Computer Land	La Mesa
201	Vista Tech Center	Vista
301	Coronado Internet Zone	Coronado
401	La Jolla Computing	La Jolla
501	Alpine Technical Center	Alpine
701	San Diego Byte Store	San Diego
801	Jamul Hardware & Software	Jamul
901	Del Mar Tech Center	Del Mar
1001	Lakeside Software Center	Lakeside
1101	Bonsall Network Store	Bonsall
1201	Rancho Santa Fe Tech	Rancho Santa Fe
1301	Spring Valley Byte Center	Spring Valley
1401	Poway Central	Poway
1601	Fairbanks Tech USA	Fairbanks Ranch
1701	Blossom Valley Tech	Blossom Valley
1801	Chula Vista Networks	

CALCULATED Keyword

As described earlier, the SELECT statement's purpose is to read data from one or more tables (or data sets). In addition to selecting columns that are stored in one or more tables, SQL can be used to dynamically create new columns containing text or computations, so they exist for the duration of the query. New columns can be created in a SELECT statement by specifying a column alias with an 'AS' keyword. The column name must adhere to the rules for SAS names and persists for that query only. In the next example, a name is assigned to the results of a new computed column (UNITCOST * UNITS) called, TOTALCOST, using the AS keyword and is formatted with a DOLLAR10.2 format for each row in the PURCHASES table.

SQL Code

```
PROC SQL;
  SELECT CUSTNUM
    ,UNITCOST
    ,UNITS
    ,UNITCOST * UNITS AS TOTALCOST FORMAT=DOLLAR10.2
   FROM PURCHASES;
QUIT;
```

Partial Results

Customer Number	Unit Cost	Units Purchased	TOTALCOST
1701	\$3,200.00	1	\$3,200.00
101	\$299.00	7	\$2,093.00
701	\$299.00	11	\$3,289.00
701	\$299.00	8	\$2,392.00
701	\$399.00	4	\$1,596.00
701	\$299.00	3	\$897.00
701	\$3,400.00	2	\$6,800.00
701	\$3,300.00	3	\$9,900.00
701	\$3,200.00	2	\$6,400.00
1301	\$299.00	3	\$897.00
1301	\$299.00	5	\$1,495.00
1301	\$399.00	2	\$798.00

1801	\$3,400.00	7	\$23,800.00
1801	\$3,400.00	4	\$13,600.00
1001	\$3,400.00	5	\$17,000.00
1101	\$3,400.00	2	\$6,800.00
1201	\$3,300.00	8	\$26,400.00
501	\$299.00	3	\$897.00
501	\$299.00	5	\$1,495.00
501	\$299.00	1	\$299.00
501	\$3,400.00	4	\$13,600.00
301	\$299.00	6	\$1,794.00
501	\$175.00	9	\$1,575.00

When an alias is specified to name a new column, the alias referencing the column can be used in the query. In the next example, a name and format is assigned to the results of the computed column, TOTALCOST, for each row in the PURCHASES table along with a WHERE clause to reference the computed column and to subset the results to display rows where the TOTALCOST is greater than \$10,000.

SQL Code

```
PROC SQL;
  SELECT CUSTNUM
    ,UNITCOST
    ,UNITS
    ,UNITCOST * UNITS AS TOTALCOST FORMAT=DOLLAR10.2
  WHERE TOTALCOST > 10000;
```

```

FROM PURCHASES
  WHERE TOTALCOST > 10000;
QUIT;

```

In the previous example you learned that we can perform computations in a SELECT statement and assign an alias to any new columns that we create. So why did SAS stop processing our query in this example and send an ERROR message to the SAS log? Earlier in this chapter, we examined the order of execution of each clause in an SQL query and learned that SAS processes the WHERE clause prior to the SELECT clause. Consequently, an error is produced, as is shown below, if the computed column is used in a WHERE clause as a condition because the WHERE clause is unaware that the computed column, TOTALCOST, exists.

SAS Log Results

```

PROC SQL;
  SELECT CUSTNUM
    ,UNITCOST
    ,UNITS
    ,UNITCOST * UNITS AS TOTALCOST FORMAT=DOLLAR10.2
  FROM PURCHASES
    WHERE TOTALCOST > 10000;
ERROR: The following columns were not found in the contributing tables:
TOTALCOST.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax of
statements.
      QUIT;
NOTE: The SAS System stopped processing this step because of errors.

```

To correct the problem illustrated in the previous example, the keyword **CALCULATED** needs to be specified in the WHERE clause along with the alias to inform SAS that the new column is derived within the query. In the next example, a name and format is assigned to the computed column, TOTALCOST, for each row in the PURCHASES table along with a WHERE clause specifying the CALCULATED keyword followed by the alias to reference, subset and display the computed column's results where the TOTALCOST is greater than \$10,000.

SQL Code

```

PROC SQL;
  SELECT CUSTNUM
    ,UNITCOST
    ,UNITS
    ,UNITCOST * UNITS AS TOTALCOST FORMAT=DOLLAR10.2
  FROM PURCHASES
    WHERE CALCULATED TOTALCOST > 10000;
QUIT;

```

Results

Customer Number	Unit Cost	Units Purchased	TOTALCOST
901	\$3,200.00	5	\$16,000.00
401	\$3,400.00	3	\$10,200.00
401	\$3,300.00	6	\$19,800.00
201	\$3,400.00	3	\$10,200.00
101	\$3,400.00	5	\$17,000.00
1301	\$3,400.00	3	\$10,200.00
1601	\$3,400.00	7	\$23,800.00
1801	\$3,400.00	4	\$13,600.00
1001	\$3,400.00	5	\$17,000.00
1201	\$3,300.00	8	\$26,400.00
501	\$3,400.00	4	\$13,600.00

Dictionary Tables

SAS generates and maintains valuable runtime information (metadata) about SAS libraries, data sets, catalogs, indexes, macros, system options, titles, views, and other content in a collection of read-only tables called *dictionary tables*. Although called tables, dictionary tables are not real tables at all.

Dictionary tables and their contents permit a SAS session's activities to be easily accessed and monitored. Information is automatically produced and each table's contents are made available at the time a SAS session begins. Table 2.2 presents the available Dictionary tables and a description of their contents.

Table 2.2: Available Dictionary Tables and Their Contents

Dictionary Table Name	Contents
DICTIONARY.CATALOGS	SAS catalogs
DICTIONARY.COLUMNS	Data set columns and attributes
DICTIONARY.EXTFILES	Allocated filerefs and external physical path
DICTIONARY.INDEXES	Data set indexes
DICTIONARY.MACROS	Global and automatic macro variables
DICTIONARY.MEMBERS	SAS data sets and other member types
DICTIONARY.OPTIONS	Current SAS system option settings
DICTIONARY.TABLES	SAS data sets and views
DICTIONARY.TITLES	Title and footnote definitions
DICTIONARY.VIEWS	SAS data views

Dictionary Tables and Metadata

SAS collects and populates valuable information (“metadata” or data about data) on SAS libraries, data sets (tables), catalogs, indexes, macros, system options, titles, views, and a collection of other read-only tables called *dictionary tables*. Dictionary tables serve a special purpose by providing system-related information about the current SAS session’s SAS databases and applications. When a query is requested against a dictionary table, SAS automatically launches a discovery process at runtime to collect information pertinent to that table. This information is made available any time after a SAS session is started.

While SAS 9.1 has 22 dictionary tables and SASHELP views, there are 29 “known” dictionary tables and SASHELP views in SAS 9.2, 30 “known” dictionary tables and SASHELP views in SAS 9.3, and 32 “known” dictionary tables and SASHELP views in SAS 9.4. The names of each DICTIONARY table and SASHELP view are illustrated in Table 2.3 below.

Table 2.3: DICTIONARY Tables and SASHELP Views

DICTIONARY Table	SASHELP View	Purpose
CATALOGS	VCATALG	Provides information about SAS catalogs.
CHECK_CONSTRAINTS	VCHKCON	Provides check constraints information.
COLUMNS	VCOLUMN	Provides information about column in tables.
CONSTRAINT_COLUMN_USAGE	VCNCOLU	Provides column integrity constraints information.
CONSTRAINT_TABLE_USAGE	VCNTABU	Provides information related to tables with integrity constraints defined.
DATAITEMS	VDATAIT	Provides information about known data items.
DESTINATIONS	VDEST	Provides information about known ODS destinations.
DICTIONARIES	VDCTNRY	Provides information about all the DICTIONARY tables.
ENGINES	VENGINE	Provides information about known SAS engines available to the session.
EXTFILES	VEXTFL	Provides information related to external files.
FILTERS	VFILTER	Provides information about known filters.
FORMATS	VFORMAT	Provides information related to defined formats and informats.
FUNCTIONS	VFUNC	Provides information about all known functions.
OPTIONS	VGOPT	Provides information about currently defined SAS/GRAFPH software graphics options.
INDEXES	VINDEX	Provides information related to defined indexes.

DICTIONARY Table	SASHELP View	Purpose
INFOMAPS	VINFOMP	Provides information about all known information maps.
LIBNAMES	VLIBNAM	Provides information related to defined SAS libraries.
MACROS	VMACRO	Provides information related to any defined macros.
MEMBERS	VMEMBER	Provides information related to objects currently defined in SAS libraries.
OPTIONS	VOPTION	Provides information related to SAS system options.
DICTIONARY Table	SASHELP View	Purpose
PROMPTS	V_PROMPT	Provides information about all known SAS/GRAFH prompts.
PROMPTSXML	V_PRMXML	Provides information about all known XML prompts.
REFERENTIAL_CONSTRAINTS	VREFCON	Provides information related to tables with referential constraints.
REMEMBER	VREMemb	Provides information about all known remembered text.
STYLES	VSTYLE	Provides information related to select ODS styles.
TABLES	VTABLE	Provides information related to currently defined tables.
TABLE_CONSTRAINTS	VTABCON	Provides information related to tables containing integrity constraints.
TITLES	VTITLE	Provides information related to currently defined titles and footnotes.
VIEWS	VVIEW	Provides information related to currently defined data views.

Displaying Dictionary Table Definitions

You can view a dictionary table's definition and enhance your understanding of each table's contents by specifying a DESCRIBE TABLE statement. The results of the statements used to create each dictionary table can be displayed in the SAS log. For example, a DESCRIBE TABLE statement is illustrated below to display the CREATE TABLE statement used in building the OPTIONS dictionary table containing current SAS system option settings.

SQL Code

```
PROC SQL;
  DESCRIBE TABLE
    DICTIONARY.OPTIONS;
QUIT;
```

SAS Log Results

```
create table DICTIONARY.OPTIONS
(
  optname char(32) label='Option Name',
  setting char(1024) label='Option Setting',
  optdesc char(160) label='Option Description',
  level char(8) label='Option Location'
);
```

Note: The information contained in dictionary tables is also available to DATA and PROC steps outside the SQL procedure. Referred to as dictionary views, each view is prefaced with the letter “V” and may be shortened with abbreviated names. Dictionary view can be accessed by referencing the view by its name in the SASHELP library. See the *SAS Procedures Guide* for further details on accessing and using dictionary views in the SASHELP library.

Dictionary Table Column Names

To help become familiar with each dictionary table’s and dictionary view’s column names and their definitions, Tables 2.4 through 2.13 identify each unique column name, type, length, format, informat, and label.

Table 2.4: DICTIONARY.CATALOGS or SASHELP.VCATALG

Column	Type	Length	Format	Informat	Label
Libname	char	8			Library Name
Memname	char	32			Member Name
Memtype	char	8			Member Type
Objname	char	32			Object Name
Objtype	char	8			Object Type
Objdesc	char	256			Description
Created	num		DATETIME.	DATETIME.	Date Created
Modified	num		DATETIME.	DATETIME.	Date Modified
Alias	char	8			Object Alias

Table 2.5: DICTIONARY.COLUMNS or SASHELP.VCOLUMN

Column	Type	Length	Label
Libname	char	8	Library Name
Memname	char	32	Member Name
Memtype	char	8	Member Type
Name	char	32	Column Name
Type	char	4	Column Type
Length	num		Column Length
Npos	num		Column Position
Varnum	num		Column Number in Table
Label	char	256	Column Label
Format	char	16	Column Format
Informat	char	16	Column Informat
Idxusage	char	9	Column Index Type

Table 2.6: DICTIONARY.EXTFILES or SASHELP.VEXTFL

Column	Type	Length	Label
Fileref	char	8	Fileref
Xpath	char	1024	Path Name
Xengine	char	8	Engine Name

Table 2.7: DICTIONARY.INDEXES or SASHELP.VINDEX

Column	Type	Length	Label
Libname	char	8	Library Name
Memname	char	32	Member Name
Memtype	char	8	Member Type
Name	char	32	Column Name
Idxusage	char	9	Column Index Type
Idxname	char	32	Index Name
Indxpos	num		Position of Column in Concatenated Key
Nomiss	char	3	Nomiss Option
Unique	char	3	Unique Option

Table 2.8: DICTIONARY.MACROS or SASHELP.VMACRO

Column	Type	Length	Label
Scope	char	9	Macro Scope
Name	char	32	Macro Variable Name
Offset	num		Offset into Macro Variable
Value	char	200	Macro Variable Name

Table 2.9: DICTIONARY.MEMBERS or SASHELP.VMEMBER

Column	Type	Length	Label
Libname	char	8	Library Name
Memname	char	32	Member Name
Memtype	char	8	Member Type
Engine	char	8	Engine Name
Index	char	32	Indexes
Path	char	1024	Path Name

Table 2.10: DICTIONARY.OPTIONS or SASHELP.VOPTION

Column	Type	Length	Label
Optname	char	32	Option Name
Setting	char	1024	Option Setting
Optdesc	char	160	Option Description
Level	char	8	Option Location

Table 2.11: DICTIONARY.TABLES or SASHELP.VTABLE

Column	Type	Length	Format	Informat	Label
Libname	char	8			Library Name
Memname	char	32			Member Name
Memtype	char	8			Member Type
Memlabel	char	256			Dataset Label
Typemem	char	8			Dataset Type
Crdate	num		DATETIME.	DATETIME.	Date Created
Modate	num		DATETIME.	DATETIME.	Date Modified

Column	Type	Length	Format	Informat	Label
Nobs	num				# of Obs
Obslen	num				Obs Length
Nvar	num				# of Variables
Protect	char	3			Type of Password Protection
Compress	char	8			Compression Routine
Encrypt	char	8			Encryption
Npage	num				# of Pages
Pcompress	num				% Compression
Reuse	char	3			Reuse Space
BuFSIZE	num				BuFSIZE
Delobs	num				# of Deleted Obs
Indxtype	char	9			Type of Indexes
Datarep	char	32			Data Representation
Reqvector	char	24	\$HEX.	\$HEX.	Requirements Vector

Table 2.12: DICTIONARY.TITLES or SASHELP.VTITLE

Column	Type	Length	Label
Type	char	1	Title Location
Number	num		Title Number
Text	char	256	Title Text

Table 2.13: DICTIONARY.VIEWS or SASHELP.VVIEW

Column	Type	Length	Label
Libname	char	8	Library Name
Memname	char	32	Member Name
Memtype	char	8	Member Type
Engine	char	8	Engine Name

Accessing a Dictionary Table's Contents

The content of a dictionary table is accessed with the SQL procedure's SELECT statement FROM clause. Results are displayed as rows and columns in a table, and can be used in handling common data processing tasks including obtaining a list of allocated libraries, catalogs and data sets, as well as communicating SAS environment settings to custom software applications. Users should take the time

to explore the capabilities of these read-only dictionary tables and become familiar with the type of information they provide.

Dictionary.CATALOGS

Obtaining detailed information about catalogs and their members is quick and easy with the CATALOGS dictionary table. You will be able to capture an ordered list of catalog information by member name including object name and type, description, date created and last modified, and object alias from any SAS library. For example, the following code produces a listing of the catalog objects in the SASUSER library.

Note: Because this dictionary table produces a considerable amount of information, users are advised to specify a WHERE clause when using.

SQL Code

```
PROC SQL;
  SELECT *
  FROM DICTIONARY.CATALOGS
  WHERE LIBNAME="SASUSER";
QUIT;
```

Results

Library Name	Member Name	Member Type	Object Name	Object Type	Object Description	Date Created	Date Modified	Object Alias	Library Concatenation Level
SASUSER	PROFBAK	CATALOG	DMKEYS	KEYS	Function Key Definitions	07NOV12:14:55:49	07NOV12:14:55:49		0
SASUSER	PROFBAK	CATALOG	VT_PRINTLIST	SLIST	vt_printlist.SLIST	22JUL13:03:24:55	22JUL13:03:24:55		0
SASUSER	PROFBAK	CATALOG	MRUWSAVE	WSAVE		22JUL13:03:53:30	22JUL13:03:53:30		0
SASUSER	PROFBAK	CATALOG	V9TUTDLG	WSAVE		22JUL13:02:23:26	22JUL13:02:23:26		0
SASUSER	PROFBAK	CATALOG	VIEWWSAVE	WSAVE	Preferences View save information.	10OCT11:08:05:24	10OCT11:08:05:24		0
SASUSER	PROFILE	CATALOG	DMKEYS	KEYS	Function Key Definitions	07NOV12:14:55:49	07NOV12:14:55:49		0
SASUSER	PROFILE	CATALOG	VT_PRINTLIST	SLIST	vt_printlist.SLIST	24JUL13:04:21:12	24JUL13:04:21:12		0
SASUSER	PROFILE	CATALOG	MRUWSAVE	WSAVE		22JUL13:03:53:30	22JUL13:03:53:30		0
SASUSER	PROFILE	CATALOG	V9TUTDLG	WSAVE		24JUL13:01:27:42	24JUL13:01:27:42		0
SASUSER	PROFILE	CATALOG	VIEWWSAVE	WSAVE	Preferences View save information.	10OCT11:08:05:24	10OCT11:08:05:24		0
SASUSER	PROFILE2	CATALOG	WNTPRINT	WINPRINT		24JUL13:01:27:23	24JUL13:01:27:23		0

Dictionary.COLUMNS

Retrieving information about the columns in one or more data sets is easy with the COLUMNS dictionary table. Similar to the results of the CONTENTS procedure, you will be able to capture column-level information including column name, type, length, position, label, format, informat, and indexes, as well as produce cross-reference listings containing the location of columns in a SAS library. For example, the following code requests a cross-reference listing of the tables containing the CUSTNUM column in the WORK library.

Note: Care should be used when specifying multiple functions on the WHERE clause because the SQL Optimizer is unable to optimize the query resulting in all allocated SAS session librefs being searched. This can cause the query to run much longer than expected.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.COLUMNS
   WHERE UPCASE(LIBNAME) = "WORK" AND
         UPCASE(NAME) = "CUSTNUM";
QUIT;
```

Results

Library Name	Member Name	Member Type	Column Name	Column Type	Column Length	Column Position	Column Number in Table	Column Label	Column Format	Column Informat	Column Index Type	Order in Key Sequence	Extended Type	Not NULL?	Precision	Scale	Transcoded?
WORK	CUSTOMERS	DATA	custnum	num	3	0	1	Customer Number			0	num	no	.	.	yes	
WORK	CUSTOMERS2	DATA	custnum	num	3	0	1	Customer Number			0	num	no	.	.	yes	
WORK	CUSTOMERS_BACKUP	DATA	custnum	num	8	0	1				0	num	no	.	.	yes	
WORK	INVOICE	DATA	custnum	num	3	6	3	Customer Number			0	num	no	.	.	yes	
WORK	PURCHASES	DATA	custnum	num	4	0	1	Custnum			0	num	no	.	.	yes	
WORK	PURCHASES2	DATA	custnum	num	4	0	1				0	num	no	.	.	yes	

Dictionary.DICTIONARIES

Users can easily identify all available dictionary tables by accessing the read-only DICTIONARIES dictionary table or VDCTNRY SASHELP view. The contents of the DICTIONARIES dictionary table and VDCTNRY SASHELP view reveals the names of supported tables and views. The following PROC SQL query specifies the UNIQUE keyword to generate a listing of existing dictionary tables.

PROC SQL Code:

```
PROC SQL;
  SELECT UNIQUE MEMNAME
    FROM DICTIONARY.DICTIONARIES;
QUIT;
```

Results:

Member Name
CATALOGS
CHECK_CONSTRAINTS
COLUMNS
CONSTRAINT_COLUMN_USAGE
CONSTRAINT_TABLE_USAGE
DATAITEMS
DESTINATIONS
DICTIONARIES
ENGINES
EXTFILES
FILTERS
FORMATS
FUNCTIONS
OPTIONS
INDEXES
INFOMAPS
LIBNAMES
MACROS
MEMBERS
OPTIONS
PROMPTS
PROMPTXML
REFERENTIAL_CONSTRAINTS
REMEMBER
STYLES
TABLES
TABLE_CONSTRAINTS
TITLES
VIEWS
VIEW_SOURCES

Dictionary.EXTFILES

Accessing allocated external files by fileref and corresponding physical path name information is a snap with the EXTFILES dictionary table. The results from this handy table can be used in an application to communicate whether a specific fileref has been allocated with a FILENAME statement. For example, the following code produces a listing of each individual path name by fileref.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.EXTFILES;
QUIT;
```

Results

Fileref	Pathname	Engine Name	Date Modified	Size of File	File Concatenation Level	Directory?	Exists?	Temporary?
#LN000002	TERMINAL	TERMINAL	01JAN60:00:00:00	0	0	no	no	yes
#LN000004	C:\Users\KPL\Documents	DISK	24JUL13:01:27:26	0	0	yes	yes	yes
#LN000006	C:\Users\KPL\Desktop	DISK	24JUL13:01:27:26	0	0	yes	yes	yes
#LN000007	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_	DISK	24JUL13:04:30:59	0	0	yes	yes	yes
#LN000008	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_\sashtml.htm	DISK	24JUL13:04:30:59	135480	0	no	yes	yes
#LN000011	C:\Program Files\SASHome\SASFoundation\9.3\core\sasmgs	DISK	03OCT11:23:50:23	0	1	yes	yes	yes
#LN000011	C:\Program Files\SASHome\SASFoundation\9.3\laccel\mval\sasmgs	DISK	03OCT11:23:50:23	0	2	yes	yes	yes
#LN000011	C:\Program Files\SASHome\SASFoundation\9.3\access\sasmgs	DISK	03OCT11:23:50:23	0	3	yes	yes	yes
#LN000011	C:\Program Files\SASHome\SASFoundation\9.3\dmcore\sasmgs	DISK	03OCT11:23:50:23	0	4	yes	yes	yes
#LN000011	C:\Program Files\SASHome\SASFoundation\9.3\spdsclient\sasmgs	DISK	03OCT11:23:50:23	0	5	yes	yes	yes

Dictionary.INDEXES

It is sometimes useful to display the names of existing simple and composite indexes, or their SAS tables, that reference a specific column name. The INDEXES dictionary table provides important information to help identify indexes that improve a query's performance. For example, to display indexes that reference the CUSTNUM column name in any of the example tables, the following code is specified.

Note: See Chapter 10, “Tuning for Performance and Efficiency,” for performance tuning techniques as they relate to indexes.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.INDEXES
    WHERE UPCASE(NAME) = "CUSTNUM"      /* Column Name */
          AND UPCASE(LIBNAME) = "WORK"; /* Library Name */
QUIT;
```

Dictionary.MACROS

The ability to capture macro variable names and their values is available with the MACROS dictionary table. The MACROS dictionary table provides information for global and automatic macro variables, but not for local macro variables. For example, to obtain columns specific to macros such as global macros SQLOBS, SQLOOPS, SQLXOBS, or SQLRC, the following code is specified.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.MACROS
   WHERE UPCASE(SCOPE) = "GLOBAL";
QUIT;
```

Results

Macro Scope	Macro Variable Name	Offset into Macro Variable	Macro Variable Value
GLOBAL	SQLOBS	0	0
GLOBAL	SQLLOOPS	0	0
GLOBAL	SYS_SQL_IP_ALL	0	-1
GLOBAL	SYS_SQL_IP_STMT	0	
GLOBAL	SQLXOBS	0	0
GLOBAL	SQLRC	0	0
GLOBAL	SQLEXITCODE	0	0

Dictionary.MEMBERS

Accessing a detailed list of data sets, views, and catalogs is the hallmark of the MEMBERS dictionary table. You will be able to access a terrific resource of information by library, member name and type, engine, indexes, and physical path name. For example, to obtain a list of the individual files in the WORK library, the following code is specified.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.MEMBERS
   WHERE UPCASE(LIBNAME) = "WORK";
QUIT;
```

Results

Library Name	Member Name	Member Type	DBMS Member Type	Engine Name	Indexes	Pathname
WORK	CUSTOMERS	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	CUSTOMERS2	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	CUSTOMERS_BACKUP	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	INVENTORY	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	INVENTORY_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	INVOICE	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	INVOICE_1K_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	JOINED_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	LAPTOP_DISCOUNT_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	LAPTOP_PRODUCTS_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	LARGEST_AMOUNT_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	MANUFACTURERS	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	MANUFACTURERS_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	PRODUCTS	DATA		V9	yes	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	PRODUCTS_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	PRODUCTS_WITH_NULLS	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	PURCHASES	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	PURCHASES2	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	SOFTWARE_PRODUCTS	DATA		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	SOFTWARE_PRODUCTS_TAX_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	SOFTWARE_PRODUCTS_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	VIEW_CUSTOMERS	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-
WORK	WORKSTATION_PRODUCTS_VIEW	VIEW		V9	no	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD8872_KPL-PC_-

Dictionary.OPTIONS

The OPTIONS dictionary table provides a list of the current SAS session's option settings including the option name, its setting, description, and location. Obtaining option settings is as easy as 1-2-3. Simply submit the following SQL query referencing the OPTIONS dictionary table as follows. A partial listing of the results from the OPTIONS dictionary table is displayed below in rich text format.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.OPTIONS;
QUIT;
```

Results

Option Name	Option type	Option Setting	Option Description	Option Location	Option Set	Option Group
APPEND	char		Append at the end of the option value	Portable	anytime	ENV/FILES
APPLETLOC	char		Location of Java applets	Portable	anytime	ENV/FILES
ARMAGENT	char		ARM Agent to use to collect ARM records	Portable	anytime	PERFORMANCE
ARMILOC	char	ARMLOG LOG	Identify location where ARM records are to be written	Portable	anytime	PERFORMANCE
ARMSUBSYS	char	(ARM_NONE)	Enable/Disable ARMING of SAS subsystems	Portable	anytime	PERFORMANCE
ASYNCHIO	Boolean	NOASYNCHIO	Enable asynchronous input/output	Portable	startup	SASFIES
AUTOCORRECT	Boolean	AUTOCORRECT	Perform auto-correction for misspelled procedure names, keywords or global statement names	Portable	anytime	ERRORHANDLING
AUTOEXEC	char		Identifies AUTOEXEC files used during initialization	Portable	startup	ENV/FILES
AUTOSAVELOC	char		Identifies the location where program editor contents are auto saved	Portable	anytime	ENV/DISPLAY
AUTOSIGNON	Boolean	NOAUTOSIGNON	SAS/CONNECT remote submit will automatically attempt to SIGNON	Portable	anytime	COMMUNICATIONS
BINDING	char	DEFAULT	Controls the binding edge for duplexed output	Portable	anytime	ODSPRINT
NLSCOMPATMODE	Boolean	NONLSCOMPATMODE	Uses the default encoding to process character data	Host	startup	LANGUAGECONTROL
OPLIST	Boolean	NOOPLIST	Write the settings of the SAS system options to the SAS log.	Host	startup	LOGCONTROL
PRINT	char		Specifies the destination for SAS output in batch or noninteractive mode	Host	startup	ENV/FILES
REALMEMSIZE	num	0	Limit on the total amount of real memory to be used by the SAS System	Host	startup	MEMORY
SASHOST	char		Specify path of sashost.dll to be loaded.	Host	startup	ENV/FILES
SETBUFFERPOOLS	char		Configure buffer memory pools.	Host	startup	MEMORY
STIMEFMT	char	(NLDATEM2.HMS TIMEAMPM KB MEMFULL TSFULL NC)	Specified the output format for FULLSTIMER and STIMER. This controls the timestamp, memory, CPU and elapsed time.	Host	anytime	LOGCONTROL
STIMER	Boolean	STIMER	Writes a subset of system performance statistics to the SAS log	Host	anytime	LOGCONTROL
SYSIN	char		Specifies the default location of SAS source code when running in batch or noninteractive mode	Host	startup	ENV/FILES
VERBOSE	Boolean	NOVERBOSE	Print configuration options to the screen.	Host	startup	LOGCONTROL
XCMD	Boolean	XCMD	The X Command is valid in this SAS session.	Host	startup	ENV/DISPLAY

Dictionary.TABLES

When you need more information about SAS files than what the MEMBERS dictionary table provides, consider using the TABLES dictionary table. The TABLES dictionary table provides such file details as library name, member name and type, date created and last modified, number of observations, observation length, number of variables, password protection, compression, encryption, number of pages, reuse space, buffer size, number of deleted observations, type of indexes, and requirements vector. For example, to obtain a detailed list of files in the WORK library, the following code is specified.

Note: Because the TABLES dictionary table produces a considerable amount of information, users should specify a WHERE clause when using it.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.TABLES
   WHERE UPCASE(LIBNAME) = "WORK";
QUIT;
```

Results

Library Name	Member Name	Member Type	DBMS Member type	Data Set Label	Data Set Type	Date Created	Date Modified	Number of Physical Observations	Observation Length	Number of Variables	Type of Protection	Compression Routine	Encryption	Number of Pages
WORK	CUSTOMERS	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	18	48	3	---	NO	NO	1
WORK	CUSTOMERS2	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	8	48	3	---	NO	NO	1
WORK	CUSTOMERS_BACKUP	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	3	48	3	---	NO	NO	1
WORK	INVENTORY	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	7	20	5	---	NO	NO	1
WORK	INVOICE	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	7	20	6	---	NO	NO	1
WORK	LAPTOP_DISCOUNT_VIEW	VIEW		VIEW	VIEW	24JUL13:04:19:12	24JUL13:04:19:12	.	56	4	---	NO	NO	0
WORK	LAPTOP_PRODUCTS_VIEW	VIEW		VIEW	VIEW	24JUL13:04:19:12	24JUL13:04:19:12	.	56	4	---	NO	NO	0
WORK	MANUFACTURERS	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	6	50	4	---	NO	NO	1
WORK	PRODUCTS	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	10	51	5	---	NO	NO	2
WORK	PRODUCTS_WITH_NULLS	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	13	51	5	---	NO	NO	1
WORK	PURCHASES	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	57	16	4	---	NO	NO	1
WORK	PURCHASES2	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	57	16	4	---	NO	NO	1
WORK	SOFTWARE_PRODUCTS	DATA		DATA	DATA	24JUL13:04:19:12	24JUL13:04:19:12	4	48	4	---	NO	NO	1
WORK	SOFTWARE_PRODUCTS_TAX_VIEW	VIEW		VIEW	VIEW	24JUL13:04:19:12	24JUL13:04:19:12	.	64	5	---	NO	NO	0

Dictionary.TITLES

The TITLES dictionary table provides a listing of the currently defined titles and footnotes in a session. The table output distinguishes between titles and footnotes using a “T” or “F” in the TITLE LOCATION column. For example, the following code displays a single title and two footnotes.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.TITLES;
QUIT;
```

Results

Title Location	Title Number	Title Text
T	1	The SAS System
F	1	Prepared by Kirk Paul Lafler, Software Intelligence Corporation
F	2	July 2013

Dictionary.VIEWS

The VIEWS dictionary table provides a listing of views for selected SAS libraries. The result of the VIEWS dictionary table displays the library name, member names and type, and engine used. For example, the following code displays a single view called VIEW_CUSTOMERS from the WORK library.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.VIEWS
   WHERE UPCASE(LIBNAME) = "WORK";
QUIT;
```

Results

Library Name	Member Name	Member Type	Engine Name
WORK	INVENTORY_VIEW	VIEW	SASESQL
WORK	INVOICE_1K_VIEW	VIEW	SASESQL
WORK	JOINED_VIEW	VIEW	SASESQL
WORK	LAPTOP_DISCOUNT_VIEW	VIEW	SASESQL
WORK	LAPTOP_PRODUCTS_VIEW	VIEW	SASESQL
WORK	LARGEST_AMOUNT_VIEW	VIEW	SASESQL
WORK	MANUFACTURERS_VIEW	VIEW	SASESQL
WORK	PRODUCTS_VIEW	VIEW	SASESQL
WORK	SOFTWARE_PRODUCTS_TAX_VIEW	VIEW	SASESQL
WORK	SOFTWARE_PRODUCTS_VIEW	VIEW	SASESQL
WORK	VIEW_CUSTOMERS	VIEW	SASESQL
WORK	WORKSTATION_PRODUCTS_VIEW	VIEW	SASESQL

Summary

1. The available data types in SQL are 1) numeric and 2) character (see the “Data Types Overview” section).
2. When a table is created with PROC SQL, numeric columns are assigned a default length of 8 bytes (see the “Numeric Data” section).
3. SAS tables store date and time information in the form of a numeric data type (see the “Date and Time Column Definitions” section).
4. A CHAR column stores a default of 8 characters (see the “Character Data” section).
5. Comparison operators are used in the SQL procedure to compare one character or numeric value to another (see the “Comparison Operators” section).
6. Logical operators are used to connect one or more expressions together in a WHERE clause and consist of AND, OR, and NOT (see the see the “Logical Operators” section).
7. The arithmetic operators used in the SQL procedure are the same as those used in the DATA step (see the “Arithmetic Operators” section).
8. Character string operators and functions are typically used with character data (see the “Character String Operators and Functions” section).
9. Predicates are used in the SQL procedure to perform direct comparisons between two conditions or expressions (see the “Predicates” section).
10. Missing or unknown information is supported by PROC SQL in a form known as a null value. A null value is not the same as a zero value (see the “Missing Values and Null” section).
11. When a new column is derived within a query, the **CALCULATED** keyword is specified before the column name, or alias, in a WHERE clause.
12. Dictionary tables provide information about the SAS environment (see the “Dictionary Tables” section).

Chapter 3: Formatting Output

Introduction	83
Formatting Output	83
Writing a Blank Line between Each Row	84
Displaying Row Numbers	85
Using the FORMAT= Column Modifier to Format Output	87
Concatenating Character Strings	88
Inserting Text and Constants between Columns	90
Using Scalar Expressions with Selected Columns	91
Ordering Output by Columns	94
Grouping Data with Summary Functions	97
Grouping Data and Sorting	99
Subsetting Groups with the HAVING Clause	100
Formatting Output with the Output Delivery System	102
ODS and Output Formats	102
Sending Output to a SAS Data Set	103
Converting Output to Rich Text Format	104
Exporting Data and Output to Excel	105
Delivering Results to the Web	107
Summary	108

Introduction

Programmers want and expect to be able to format output in a variety of ways. The SQL procedure has forged innovative ways to enhance the appearance of output including double-spacing rows of output, concatenating two or more columns, inserting text and constants between selected columns, displaying column headers for derived fields, and much more. As a value-added feature, the SQL procedure (not part of ANSI-standard SQL—see the “Introduction” section of this book) can be integrated with the Output Delivery System to enhance and format output in ways not otherwise available. As you review the many examples in this chapter, the following points should be kept in mind:

Formatting Output

As a language, PROC SQL consists of a standard set of statements and options to create, retrieve, alter, transform, and transfer data regardless of the operating system or where the data is located. These features provide tremendous power as well as control when integrating information from a variety of sources in a number of ways. Because emphasis is placed on PROC SQL’s data manipulation capabilities and not on its format and output capabilities, many programmers are unfamiliar with the SQL procedure’s output-producing side. Consequently, programmers resort to using report writers or special outputting tools to create

the best looking output. To illustrate the virtues of PROC SQL in SAS, this chapter presents numerous examples of how output can be formatted and produced.

Writing a Blank Line between Each Row

The ability to display a blank line between each row of output is available as a procedure option in PROC SQL. As with the PRINT procedure, specifying DOUBLE in the SQL procedure inserts a blank line between each physical row of output (NODOUBLE is the default). Setting this option is especially useful when one or more flowed lines spans or wraps in the output because it provides visual separation between each row of data. This example illustrates using the DOUBLE option to double-space output.

SQL Code

```
PROC SQL DOUBLE;
  SELECT *
    FROM INVOICE;
QUIT;
```

Results

The SAS System

Invoice Number	Manufacturer Number	Customer Number	Invoice Quantity - Units Sold	Invoice Unit Price	Product Number
1001	500	201	5	\$1,495.00	5001
1002	600	1301	2	\$1,598.00	6001
1003	210	101	7	\$245.00	2101
1004	111	501	3	\$9,600.00	1110
1005	500	801	2	\$798.00	5002
1006	500	901	4	\$396.00	6000
1007	500	401	7	\$23,100.00	1200

To revert back to single-spaced output, the RESET statement can be specified as long as the QUIT statement has not been issued to turn off the SQL procedure. When PROC SQL is active, the RESET statement can be specified with or without options to reestablish each option's original settings. All SQL options are reset to their original settings when RESET is specified without any options. When the RESET statement is specified with one or more options, only those options are reset. This example illustrates using the NODOUBLE option to turn off double-spaced output and reset printing to the default single-spaced output.

SQL Code

```
PROC SQL;
  RESET NODOUBLE;
QUIT;
```

Displaying Row Numbers

You can specify an SQL procedure option called NUMBER to display row numbers on output under the column heading Row. As with the Obs column produced by the PRINT procedure, the NUMBER option displays row numbers on output. The next example shows the NUMBER option being specified with the SQL procedure to display row numbers on output.

SQL Code

```
PROC SQL NUMBER;
  SELECT PRODNUM,
         UNITS,
         UNITCOST
    FROM PURCHASES;
QUIT;
```

Results

The SAS System

Row	Prodnum	Units	Unitcost
1	1110	1	\$3,200.00
2	5001	7	\$299.00
3	5001	11	\$299.00
4	5003	8	\$299.00
5	5002	4	\$399.00
6	5004	3	\$299.00
7	1700	2	\$3,400.00
8	1200	3	\$3,300.00
9	1110	2	\$3,200.00
10	5001	3	\$299.00
11	5003	5	\$299.00
12	5002	2	\$399.00
13	1700	2	\$3,400.00
14	1200	3	\$3,300.00
15	1110	5	\$3,200.00
16	5001	9	\$299.00
17	5002	5	\$399.00
18	5003	8	\$299.00
19	5004	2	\$299.00
20	5001	11	\$299.00

21	5002	5	\$399.00
22	5003	7	\$299.00
23	5004	3	\$299.00
24	1700	3	\$3,400.00
25	1200	6	\$3,300.00
26	5001	6	\$299.00
27	5001	6	\$299.00
28	5003	9	\$299.00
29	5002	4	\$399.00
30	1700	3	\$3,400.00
31	5001	2	\$299.00
32	5001	2	\$299.00
33	2102	5	\$175.00
34	2102	9	\$175.00
35	2102	11	\$175.00
36	2102	7	\$175.00
37	2102	5	\$175.00
38	2102	12	\$175.00
39	2102	8	\$175.00
40	2200	3	\$130.00
41	2102	9	\$175.00
42	5003	3	\$299.00
43	5004	2	\$299.00
44	1200	3	\$3,300.00
45	1700	5	\$3,400.00
46	1700	3	\$3,400.00
47	1700	7	\$3,400.00
48	1700	4	\$3,400.00
49	1700	5	\$3,400.00
50	1700	2	\$3,400.00
51	1200	8	\$3,300.00
52	5001	3	\$299.00
53	5003	5	\$299.00
54	5004	1	\$299.00
55	1700	4	\$3,400.00
56	5001	6	\$299.00
57	2102	9	\$175.00

Using the FORMAT= Column Modifier to Format Output

The SQL procedure supports the specification of a FORMAT= column modifier for purposes of providing instructions on how to write character and numeric data values by a query expression. The general syntax for writing a format follows:

<\$>format<w>.<d>

where

\$ is used to reference a character format and its absence references a numeric format.

format is the name of the desired SAS format or user-defined format to be used (see the *SAS Language Reference: Dictionary, Fourth Edition* for a complete listing and description of the available character, numeric, and date formats).

w references the format width, which is typically represented by the number of output columns used for the data.

d represents the optional decimal positioning for numeric data.

Referring to the example first shown in Chapter 2, “Working with Data in PROC SQL,” the next example illustrates the computation of a discounted value (80% of PRODCOST) from the PRODUCTS table, the assignment of a column alias of “Discount_Price” with the AS keyword, and the DOLLAR9.2 format applied to the resulting value with a FORMAT= column modifier.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST * 0.80 AS Discount_Price FORMAT=DOLLAR9.2
    FROM PRODUCTS
   ORDER BY 3;
QUIT;
```

Results

The SAS System		
Product Name	Product Type	Discount_Price
Analog Cell Phone	Phone	\$28.00
Office Phone	Phone	\$104.00
Digital Cell Phone	Phone	\$140.00
Spreadsheet Software	Software	\$239.20
Graphics Software	Software	\$239.20
Wordprocessor Software	Software	\$239.20
Database Software	Software	\$319.20
Dream Machine	Workstation	\$2,560.00
Business Machine	Workstation	\$2,640.00
Travel Laptop	Laptop	\$2,720.00

Concatenating Character Strings

As was presented in Chapter 2, two or more strings can be concatenated to produce a combined and longer string of characters. The concatenation character string operator, which is represented by two vertical bars “||”, “!!”, or “|||” (depending on the operating system and keyboard being used), combines two or more strings or columns together to form a new string value. In the next example, the manufacturer city and manufacturer state columns from the MANUFACTURERS table are concatenated so that the second column immediately follows the first. Although the two character strings are successfully concatenated, the output illustrates potential problems as a result of using the concatenation operator.

The next example shows that because the first column has a “fixed” length, blanks are automatically padded to the entire length of the first concatenated column for each row of data. As a result of using the concatenation operator, the column headers for both columns are suppressed. Readers are cautioned that due to the loss of some or all of the column header information, a true understanding of the contents of the output may be in jeopardy.

SQL Code

```
PROC SQL;
  SELECT manucity || manustat
    FROM  MANUFACTURERS;
QUIT;
```

Results

The SAS System

Houston TX
San Diego CA
Miami FL
San Mateo CA
San Diego CA
San Diego CA

To make the preceding output appear a bit more readable and complete, you should consider a few modifications. First, column headers could be assigned as aliases with the AS operator. The maximum size of a user-defined column header is 32 bytes in length (adhering to SAS naming conventions). Finally, the TRIM function (which is described in Chapter 2, “Working with Data in PROC SQL”) could be used to remove trailing blanks from the city column. This allows the second column to act as a floating field.

SQL Code

```
PROC SQL;
  SELECT TRIM(manucity) || manustat AS Headquarters
    FROM MANUFACTURERS;
QUIT;
```

Results

The SAS System

Headquarters
HoustonTX
San DiegoCA
MiamiFL
San MateoCA
San DiegoCA
San DiegoCA

Although the preceding output illustrates that some changes were made, it is still difficult to read. A few more cosmetic changes should be made to make it more aesthetically appealing and readable. In the next section, the output will be customized to give the data further separation.

Inserting Text and Constants between Columns

At times, it is useful to be able to insert text and/or constants in query output. This enables special characters including symbols and comments to be inserted in the output. We can improve the output in the previous example by inserting a comma “,” and a single blank space between the manufacturer city and manufacturer state information. The final output illustrates an acceptable way to display columnar data using a “free-floating” presentation as opposed to fixed columns.

SQL Code

```
PROC SQL;
  SELECT trim(manucity) || ', ' || manustat
    AS Headquarters
   FROM MANUFACTURERS;
QUIT;
```

Results

The SAS System

Headquarters
Houston, TX
San Diego, CA
Miami, FL
San Mateo, CA
San Diego, CA
San Diego, CA

Another method of automatically concatenating character strings, removing leading and trailing blanks, and inserting text and constants is with the CATX function. The next example shows the CATX function with a “,” specified as a separator (a blank character immediately follows the comma) between character strings MANUCITY and MANUSTAT. Although a comma and blank character are successfully inserted between the MANUCITY and MANUSTAT columns, the output is automatically left justified conforming to the value assigned in the LINESIZE= system option.

SQL Code

```
PROC SQL;
  SELECT CATX(', ', manucity, manustat)
    AS Headquarters
   FROM MANUFACTURERS;
QUIT;
```

Results

The SAS System

Headquarters
Houston, TX
San Diego, CA
Miami, FL
San Mateo, CA
San Diego, CA
San Diego, CA

Using Scalar Expressions with Selected Columns

In computing terms, a *scalar* refers to a quantity represented by a single number or value. The value is not represented as an array or list of values, but as a single value. For example, the value 7 is a scalar value, but (0,0,7) is not. PROC SQL allows the use of scalar expressions and constants with selected columns. Typically, these expressions replace or augment one or more columns in the SELECT statement. To illustrate how a scalar expression is used, assume that a value of 7.5% representing the sales tax percentage is computed for each product in the PRODUCTS table. The results consist of the product name, product cost, and derived computed sales tax column.

Note: Although the computed column contains the results of the sales tax computation for each product, it does not contain a column heading.

SQL Code

```
PROC SQL;
  SELECT prodname, prodcost,
    .075 * prodcost
  FROM PRODUCTS;
QUIT;
```

Results

The SAS System

Product Name	Product Cost	
Dream Machine	\$3,200.00	240
Business Machine	\$3,300.00	247.5
Travel Laptop	\$3,400.00	255
Analog Cell Phone	\$35.00	2.625
Digital Cell Phone	\$175.00	13.125
Office Phone	\$130.00	9.75
Spreadsheet Software	\$299.00	22.425
Database Software	\$399.00	29.925
Wordprocessor Software	\$299.00	22.425
Graphics Software	\$299.00	22.425

In the next two examples, a column header or alias is assigned to the derived sales tax column computed in the previous example. Two methods exist for achieving this. As illustrated in the next example, the first method uses the AS keyword to assign a name to the derived column as well as to permit referencing the column later in the query.

SQL Code

```
PROC SQL;
  SELECT prodname, prodcost,
         .075 * prodcost AS Sales_Tax
    FROM PRODUCTS;
QUIT;
```

Results

The SAS System

Product Name	Product Cost	Sales_Tax
Dream Machine	\$3,200.00	240
Business Machine	\$3,300.00	247.5
Travel Laptop	\$3,400.00	255
Analog Cell Phone	\$35.00	2.625
Digital Cell Phone	\$175.00	13.125
Office Phone	\$130.00	9.75
Spreadsheet Software	\$299.00	22.425
Database Software	\$399.00	29.925
Wordprocessor Software	\$299.00	22.425
Graphics Software	\$299.00	22.425

The next example illustrates the second method of assigning a column heading for the computed sales tax column with the LABEL= option. To further enhance the output's readability, a numeric dollar format is specified.

Note: Because the next example is a query and the table is not being updated, the assigned attributes are only available for the duration of the step and are not permanently saved in the table's record descriptor.

SQL Code

```
PROC SQL;
  SELECT proiname, prodcost,
    .075 * prodcost FORMAT=DOLLAR7.2
    LABEL='Sales Tax'
  FROM PRODUCTS;
QUIT;
```

Results

The SAS System

Product Name	Product Cost	Sales Tax
Dream Machine	\$3,200.00	\$240.00
Business Machine	\$3,300.00	\$247.50
Travel Laptop	\$3,400.00	\$255.00
Analog Cell Phone	\$35.00	\$2.63
Digital Cell Phone	\$175.00	\$13.13
Office Phone	\$130.00	\$9.75
Spreadsheet Software	\$299.00	\$22.43
Database Software	\$399.00	\$29.93
Wordprocessor Software	\$299.00	\$22.43
Graphics Software	\$299.00	\$22.43

Ordering Output by Columns

By definition, tables are unordered sets of data. The data that comes from a table does not automatically appear in any particular order. To offset this behavior, the SQL procedure provides the ability to impose order in a table by using an ORDER BY clause. When used, this clause orders the query results according to the values in one or more selected columns; it must be specified after the FROM clause.

Rows of data can be ordered in ascending (default order) or descending (DESC) order for each column specified (ascending is the default order). To illustrate how selected columns of data can be ordered, let's first view the PRODUCTS table and all its columns arranged in ascending order by product number (PRODNUM).

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
    ORDER BY prodnum;
QUIT;
```

Results

The SAS System

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
1700	Travel Laptop	170	Laptop	\$3,400.00
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00
5001	Spreadsheet Software	500	Software	\$299.00
5002	Database Software	500	Software	\$399.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00

The next example uses the third column ordinal position in an ORDER BY clause to order and display the query results.

SQL Code

```
PROC SQL;
  SELECT prodname, prodcost,
    .075 * prodcost AS Sales_Tax
  FROM PRODUCTS
  ORDER BY 3;
QUIT;
```

Results

The SAS System

Product Name	Product Cost	Sales_Tax
Analog Cell Phone	\$35.00	2.625
Office Phone	\$130.00	9.75
Digital Cell Phone	\$175.00	13.125
Spreadsheet Software	\$299.00	22.425
Graphics Software	\$299.00	22.425
Wordprocessor Software	\$299.00	22.425
Database Software	\$399.00	29.925
Dream Machine	\$3,200.00	240
Business Machine	\$3,300.00	247.5
Travel Laptop	\$3,400.00	255

The next example illustrates a query that selects and orders multiple columns of data from the PRODUCTS table. Output is arranged first in ascending order by product type (PRODTYPE) and then within product type in descending order by product cost (PRODCOST). The code and output are shown.

SQL Code

```
PROC SQL;
  SELECT prodname, prodtype, prodcost, prodnum
  FROM PRODUCTS
    ORDER BY prodtype, prodcost DESC;
QUIT;
```

Results

The SAS System

Product Name	Product Type	Product Cost	Product Number
Travel Laptop	Laptop	\$3,400.00	1700
Digital Cell Phone	Phone	\$175.00	2102
Office Phone	Phone	\$130.00	2200
Analog Cell Phone	Phone	\$35.00	2101
Database Software	Software	\$399.00	5002
Spreadsheet Software	Software	\$299.00	5001
Graphics Software	Software	\$299.00	5004
Wordprocessor Software	Software	\$299.00	5003
Business Machine	Workstation	\$3,300.00	1200
Dream Machine	Workstation	\$3,200.00	1110

Grouping Data with Summary Functions

Occasionally it may be useful to display data in designated groups. A GROUP BY clause is used in these situations to aggregate and order groups of data using a designated column with the same value. When a GROUP BY clause is used without a summary function, SAS issues a warning in the SAS log with the message, “A GROUP BY clause has been transformed into an ORDER BY clause because neither the SELECT clause nor the optional HAVING clause of the associated table-expression referenced a summary function.” The GROUP BY clause is transformed into an ORDER BY clause and then processed.

When a GROUP BY clause is used without specifying a summary function in the SELECT statement, the entire table is treated as a single group and is ordered in ascending order. The next example illustrates a GROUP BY clause without any summary function specifications. Due to the absence of any summary functions, the GROUP BY clause is automatically transformed into an ORDER BY clause, with the rows being ordered in ascending order by product type (PRODTYPE).

SQL Code

```
PROC SQL;
  SELECT prodtype,
         prodcost
    FROM PRODUCTS
   GROUP BY prodtype;
QUIT;
```

Log Results

```
PROC SQL;
  SELECT prodtype,
         prodcost
    FROM PRODUCTS
   GROUP BY prodtype;
WARNING: A GROUP BY clause has been transformed into an ORDER BY clause
because neither the SELECT clause nor the optional HAVING clause of the
associated table-expression referenced a summary function.
QUIT;
```

Results

The SAS System

Product Type	Product Cost
Laptop	\$3,400.00
Phone	\$130.00
Phone	\$175.00
Phone	\$35.00
Software	\$299.00
Software	\$299.00
Software	\$299.00
Software	\$399.00
Workstation	\$3,200.00
Workstation	\$3,300.00

When a GROUP BY clause is used with a summary function, the rows are aggregated in a series of groups. This means that an aggregate function is evaluated on a group of rows and not on a single row at a time. Suppose that the least expensive product in each product type category (PRODTYPE) needs to be identified. A separate query for each product category could be specified using the MIN function to determine the cheapest product. But this would require separate runs to be executed—not a very good approach. A better way to do this would be to specify a GROUP BY clause in a single statement as follows.

SQL Code

```
PROC SQL;
  SELECT prodtype,
    MIN(prodcost) AS Cheapest
      Format=dollar9.2 Label='Least Expensive'
    FROM PRODUCTS
    GROUP BY prodtype;
QUIT;
```

Results

Product Type	Least Expensive
Laptop	\$3,400.00
Phone	\$35.00
Software	\$299.00
Workstation	\$3,200.00

Grouping Data and Sorting

In the absence of an ORDER BY clause, the SQL procedure automatically sorts the results from a grouped query in the same order as specified in the GROUP BY clause. When both an ORDER BY clause and GROUP BY clause are specified for the same column(s) or column order, no additional processing occurs to satisfy the request. Because the ORDER BY clause and GROUP BY clause are not mutually exclusive, they can be used together. Internally, the GROUP BY clause first sorts the results on the grouping column(s) and then aggregates the rows of the query by the same grouping column.

But what happens when the column(s) specified in the ORDER BY clause and GROUP BY clause are not the same? In these situations, additional processing requirements are generally required. The additional processing, in a worse-case scenario, may require remerging summary statistics with the original data. In other cases, additional sorting requirements may be necessary. Suppose that information about the least expensive product in each product category is desired. But instead of automatically sorting the results in ascending order by product type, as before, the results will be displayed in ascending order by the least expensive product for each product type.

SQL Code

```
PROC SQL;
  SELECT prodtype,
    MIN(prodcost) AS Cheapest
      Format=dollar9.2 Label='Least Expensive'
    FROM PRODUCTS
    GROUP BY prodtype
    ORDER BY cheapest;
QUIT;
```

Results

The SAS System

Product Type	Least Expensive
Phone	\$35.00
Software	\$299.00
Workstation	\$3,200.00
Laptop	\$3,400.00

Subsetting Groups with the HAVING Clause

When processing groups of data, it is frequently useful to subset aggregated rows (or groups) of data. This way, aggregated data can be filtered one group at a time in contrast to filtering one row at a time. SQL provides a convenient way to subset (or filter) groups of data by using the GROUP BY clause and the HAVING clause. The HAVING clause is optional, but when specified is used in combination with the GROUP BY clause.

The HAVING clause is similar to the WHERE clause in that it specifies conditions that must be satisfied in order for results to become part of the subset. It differs from the WHERE clause in that it can refer to the results derived from summary functions after the aggregation of all observations for each of the groups. The groups that evaluate to true based on the HAVING clause are retained, and the groups that evaluate to false are automatically removed from consideration.

Suppose that you wanted to identify only those product groupings that have an average cost that is less than \$400.00 from the PRODUCTS table. Your first inclination might be to use a summary function in a WHERE clause. But, this would not be valid because a WHERE clause restricts the number of rows selected. In contrast, the HAVING clause restricts the number of groups selected, and is always performed after the GROUP BY clause. For those already familiar with subqueries as discussed in Chapter 7, “Coding Complex Queries,” you could also approach the problem as a complex query. But, there is an easier and more straightforward way of identifying and selecting the desired product groups using the GROUP BY clause and HAVING clause, as follows.

SQL Code

```
PROC SQL;
  SELECT prodtype,
         AVG(prodcost)
    FORMAT=DOLLAR9.2 LABEL='Average Product Cost'
   FROM PRODUCTS
  GROUP BY prodtype
  HAVING AVG(prodcost) <= 400.00;
QUIT;
```

Results

The SAS System

Product Type	Average Product Cost
Phone	\$113.33
Software	\$324.00

To illustrate the GROUP BY clause and the HAVING clause further, suppose you have a \$400.00 spending limit and you want to identify the least expensive product within each product grouping from the PRODUCTS table. The next example returns only those product types that are within the \$400.00 spending limit, as follows.

SQL Code

```
PROC SQL;
  SELECT prodtype,
         MIN(prodcost) AS Cheapest
           FORMAT=DOLLAR9.2 LABEL='Least Expensive'
    FROM PRODUCTS
   GROUP BY prodtype
  HAVING Cheapest <= 400.00;
QUIT;
```

Results

The SAS System

Product Type	Least Expensive
Phone	\$35.00
Software	\$299.00

When the WHERE clause, GROUP BY clause, and HAVING clause are used together in a SELECT statement, the WHERE clause is processed first. Then, the rows returned by the WHERE clause are grouped in accordance with the GROUP BY clause. Finally, the conditions specified in the HAVING clause are applied to the groups before the results are produced. The next example returns those software and hardware product types that are within the \$400.00 spending limit.

SQL Code

```
PROC SQL;
  SELECT prodtype,
         AVG(prodcost) AS Average_Product_Cost
           FORMAT=dollar12.2
    FROM PRODUCTS
   WHERE prodtype IN ('Software', 'Hardware')
   GROUP BY prodtype
  HAVING Average_Product_Cost < 400;
QUIT;
```

Results

The SAS System	
Product Type	Average_Product_Cost
Software	\$324.00

Formatting Output with the Output Delivery System

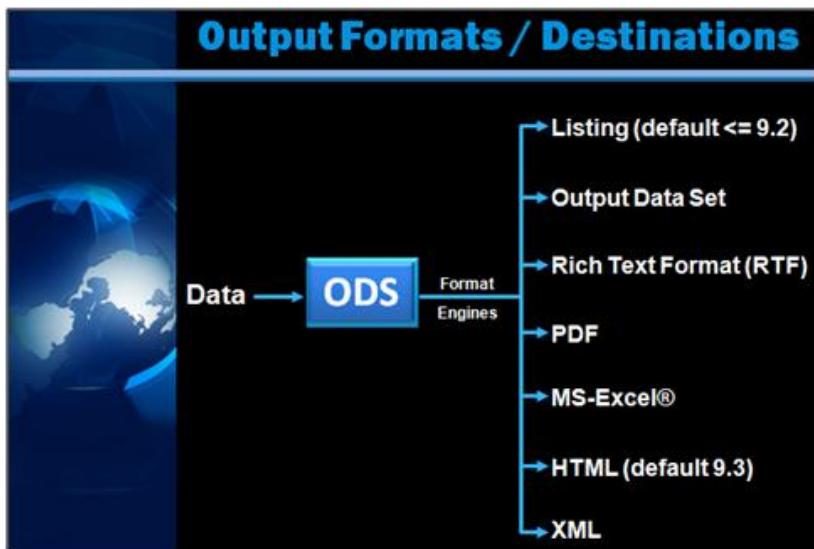
SAS provides users with a familiar and automatic way to look at output in a listing file. Although easy to use, it is not extremely flexible when it comes to creating nice-looking output. The SAS Output Delivery System (ODS) provides many ways to format output by controlling the way it is accessed and formatted. Many output formats are available with ODS, including traditional SAS monospace font (i.e., Listing).

ODS was first introduced in version 7 as a way to improve the way traditional SAS output looks. It enables quality looking output to be produced without the need to import it into word processors such as Microsoft Word. Since then, many additional output formatting features and options are available for SAS users to take advantage of. With ODS, users have a powerful and easy way to create and access formatted procedure and DATA step output.

ODS and Output Formats

ODS statements are classified as global statements and are processed immediately by SAS. With built-in format engines, which are referred to as output destinations, ODS prepares output using special formats and layouts. Figure 3.1 illustrates the types of output that can be produced with ODS.

Figure 3.1: ODS Output Destinations



The PROC step and the DATA step produce output in the form of an output object to any and all open destinations. An output destination controls what format engine is turned on during a step or until another ODS statement is specified. One or more output destinations can be opened concurrently. When a destination is open, one or more output objects can be sent to it. Conversely, when a destination is closed, output objects are not sent to the destination.

Several really good ODS books are available for further study on this exciting facility including *Output Delivery System: The Basics and Beyond*, by Lauren Haworth, Cynthia L. Zender, and Michele Burlew; and *Quick Results with the Output Delivery System*, by Sunil Kumar Gupta.

Sending Output to a SAS Data Set

Output produced by the SQL procedure can also be used as input to another vendor's SQL procedure, or DATA step. ODS provides an easy and consistent alternative for creating a SAS table of results. For users who are already familiar with ODS, this approach will consist of a simple process of specifying the OUTPUT destination in an ODS statement. For users who prefer a more traditional ANSI SQL approach, the CREATE TABLE statement (see Chapter 5, “Creating, Populating, and Deleting Tables,” for more details) will be the method of choice. Depending on the method you ultimately use, the advantage of creating a SAS table of output is a handy feature that all SAS users should become familiar with.

Although the CREATE TABLE statement is the standard method of creating a table in PROC SQL, the ODS OUTPUT statement can also be specified to produce a table. Using the ODS OUTPUT statement, the result table is a rectangular structure consisting of one or more rows and columns. In this example, the SQL procedure’s results are stored in object SQL_Results and are then sent to data set SQL_DATA using the ODS OUTPUT destination. The resulting data set is displayed using VIEWTABLE.

SQL Code

```
ODS LISTING CLOSE;
ODS OUTPUT SQL_Results = SQL_DATA;

PROC SQL;
  TITLE1 'Delivering Output to a Data Set';
  SELECT prodname, prodtype, prodcost, prodnum
  FROM PRODUCTS
  ORDER BY prodtype;
QUIT;

ODS OUTPUT CLOSE;
ODS LISTING;
```

Results

	Product Name	Product Type	Product Cost	Product Number
1	Travel Laptop	Laptop	\$3,400.00	1700
2	Office Phone	Phone	\$130.00	2200
3	Digital Cell Phone	Phone	\$175.00	2102
4	Analog Cell Phone	Phone	\$35.00	2101
5	Spreadsheet Software	Software	\$299.00	5001
6	Graphics Software	Software	\$299.00	5004
7	Wordprocessor Software	Software	\$299.00	5003
8	Database Software	Software	\$399.00	5002
9	Dream Machine	Workstation	\$3,200.00	1110
10	Business Machine	Workstation	\$3,300.00	1200

Converting Output to Rich Text Format

Rich Text Format (RTF) is text that includes codes that represent special formatting attributes. Although most frequently associated with a word processing program's ability to read and create encapsulated text fonts and highlighting attributes during copy-and-paste operations, the ODS RTF destination permits output that is generated by SAS to be packaged as RTF. This enables you to produce output that can be shared.

The next example illustrates SQL output being sent to an external RTF file using the RTF format engine. First, the default Listing destination is closed, and then the RTF format engine is opened with an external file destination to which SQL results will be routed. After the SQL procedure executes, the RTF destination is closed and the default Listing destination is opened.

Note: Opening the RTF file automatically invokes your system's default word processor and displays the file contents.

SQL Code

```
ODS LISTING CLOSE;
ODS RTF FILE='c:\SQL_Results.rtf';

PROC SQL;
  TITLE1 'Delivering Output to Rich Text Format';
  SELECT proiname, prodtype, prodcost, prodnum
    FROM PRODUCTS
   ORDER BY prodtype;
QUIT;

ODS RTF CLOSE;
ODS LISTING;
```

Results

Delivering Output to Rich Text Format

Product Name	Product Type	Product Cost	Product Number
Travel Laptop	Laptop	\$3,400.00	1700
Office Phone	Phone	\$130.00	2200
Digital Cell Phone	Phone	\$175.00	2102
Analog Cell Phone	Phone	\$35.00	2101
Spreadsheet Software	Software	\$299.00	5001
Graphics Software	Software	\$299.00	5004
Wordprocessor Software	Software	\$299.00	5003
Database Software	Software	\$399.00	5002
Dream Machine	Workstation	\$3,200.00	1110
Business Machine	Workstation	\$3,300.00	1200

Exporting Data and Output to Excel

SAS provides users with an assortment of features and options for exporting data and output to Microsoft Excel. One approach enables the transfer of data and output to a Comma Separated Values (CSV) file. A CSV file is a widely supported file format used to transfer and store tabular data from database applications to a spreadsheet. CSV files store text and numbers as plain text that can then be viewed and/or read using a text editor. Using the Output Delivery System (ODS) with the SQL procedure, data and output can be sent from SAS to a CSV file. In the following example, the ODS CSV statement is specified to produce a CSV file that can then be opened in Microsoft Excel and saved as a Microsoft Excel spreadsheet file. The resulting CSV file is displayed below in Microsoft Excel.

SQL Code

```
ODS LISTING CLOSE;
ODS CSV file='sas-to-excel.csv';

PROC SQL;
  SELECT *
  FROM PRODUCTS;
QUIT;

ODS CSV CLOSE;
ODS LISTING;
```

Results

	A	B	C	D	E
1	Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
2		1110 Dream Machine	111	Workstation	\$3,200.00
3		1200 Business Machine	120	Workstation	\$3,300.00
4		1700 Travel Laptop	170	Laptop	\$3,400.00
5		2101 Analog Cell Phone	210	Phone	\$35.00
6		2102 Digital Cell Phone	210	Phone	\$175.00
7		2200 Office Phone	220	Phone	\$130.00
8		5001 Spreadsheet Software	500	Software	\$299.00
9		5002 Database Software	500	Software	\$399.00
10		5003 Wordprocessor Software	500	Software	\$299.00
11		5004 Graphics Software	500	Software	\$299.00

Another way of exporting data and output to Microsoft Excel is by using the ODS HTML statement (see the “Delivering Results to the Web” section for details). The HyperText Markup Language (HTML) destination is actually a tagset that contains instructions for importing data and output into the Excel format. In the next example, the ODS HTML statement is specified to import output into a Microsoft Excel spreadsheet file. The resulting XLS file is displayed below in Microsoft Excel.

SQL Code

```
ODS LISTING CLOSE;
ODS HTML file='sas-to-excel.xls';

PROC SQL;
  SELECT *
    FROM PRODUCTS;
QUIT;

ODS HTML CLOSE;
ODS LISTING;
```

Results

	A	B	C	D	E
1	Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
2					
3		1110 Dream Machine	111	Workstation	\$3,200.00
4		1200 Business Machine	120	Workstation	\$3,300.00
5		1700 Travel Laptop	170	Laptop	\$3,400.00
6		2101 Analog Cell Phone	210	Phone	\$35.00
7		2102 Digital Cell Phone	210	Phone	\$175.00
8		2200 Office Phone	220	Phone	\$130.00
9		5001 Spreadsheet Software	500	Software	\$299.00
10		5002 Database Software	500	Software	\$399.00
11		5003 Wordprocessor Software	500	Software	\$299.00
12		5004 Graphics Software	500	Software	\$299.00

Delivering Results to the Web

With the popularity of the Internet, you may find it useful to deploy selected pieces of output on your intranet or website. ODS makes deploying output to the web a snap. The HTML destination creates syntactically correct HTML code to be used with one of the leading Internet browsers.

Four types of files can be produced with the ODS HTML destination:

- body
- contents
- page
- frame

A unique file name must be assigned to each file that is created with the ODS HTML statement. A custom and integrated file structure is automatically created when each file is combined with a frame file. To improve navigation and access of information, the web browser automatically places horizontal and vertical scroll bars on the generated page if necessary.

The next example illustrates PROC SQL output being sent to external HTML files using the HTML format engine. First, the default Listing destination is closed, and then the HTML format engine is opened specifying BODY, CONTENTS, PAGE, and FRAME external files for the routing of SQL procedure results. After the SQL procedure executes, the HTML destination is closed and the default Listing destination is opened.

SQL Code

```
ODS LISTING CLOSE;
ODS HTML      BODY='Products-body.html'
              CONTENTS='Products-contents.html'
              PAGE='Products-page.html'
              FRAME='Products-frame.html';

PROC SQL;
  TITLE1 'Products List';
  SELECT proiname, prodtype, prodcost, prodnum
  FROM PRODUCTS
  ORDER BY prodtype;
QUIT;

ODS HTML CLOSE;
ODS LISTING;
```

Results

<i>Table of Contents</i>		Products List			
1. SQL					
		<i>Query Results</i>			
<i>Table of Pages</i>					
1. SQL		Product Name	Product Type	Product Cost	Product Number
		Travel Laptop	Laptop	\$3,400.00	1700
		Office Phone	Phone	\$130.00	2200
		Digital Cell Phone	Phone	\$175.00	2102
		Analog Cell Phone	Phone	\$35.00	2101
		Spreadsheet Software	Software	\$299.00	5001
		Graphics Software	Software	\$299.00	5004
		Wordprocessor Software	Software	\$299.00	5003
		Database Software	Software	\$399.00	5002
		Dream Machine	Workstation	\$3,200.00	1110
		Business Machine	Workstation	\$3,300.00	1200

Summary

1. A blank line can be displayed between each row of output (see the “Writing a Blank Line between Each Row” section).
2. Columns can be concatenated to form a single column of data (see the “Concatenating Character Strings” section).
3. Text and constants can be inserted between selected columns (see the “Inserting Text and Constants between Columns” section).
4. Numeric or character scalar values can be produced with expressions (see the “Using Scalar Expressions with Selected Columns” section).
5. User-defined values can be assigned to derived column headers (see the “Using Scalar Expressions with Selected Columns” section).
6. Formats can be assigned and stored permanently to automatically display a user-defined formatted value instead of the unformatted value (see the “Using Scalar Expressions with Selected Columns” section).
7. Columns do not have to appear as unordered sets of data. One or more columns can be arranged in ascending or descending order (see the “Ordering Output by Columns” section).
8. Selected columns can be organized and displayed in groups (see the “Grouping Data with Summary Functions” section).
9. PROC SQL can be coupled with ODS to extend output formatting capabilities (see the “ODS and Output Formats” section).

Chapter 4: Coding PROC SQL Logic

Introduction	109
Conditional Logic	109
WHERE versus ON Clause	110
WHERE versus HAVING Clause.....	110
Conditional Logic with Predicates (Operators)	113
CASE Expressions.....	114
Simple CASE Expression	115
Searched CASE Expression.....	126
Case Logic versus COALESCE Expression.....	131
Assigning Labels and Grouping Data.....	133
Logic and Nulls	135
IFC and IFN Functions.....	137
Interfacing PROC SQL with the Macro Language	139
Exploring Macro Variables and Values	139
Creating Multiple Macro Variables.....	144
Using Automatic Macro Variables to Control Processing.....	147
Building Macro Tools and Applications	149
Creating Simple Macro Tools	149
Cross-Referencing Columns	149
Determining the Number of Rows in a Table	150
Identifying Duplicate Rows in a Table.....	151
Summary	152

Introduction

Expressions in the SQL procedure can be simple or complex and are represented by a combination of columns, symbols, operators, functions, constants, and literals. Specified in SQL statements and clauses, expressions are typically used in conditional logic constructs to test or compare a value against another value. The application of an expression in a CASE expression allows individual rows of data to be processed and grouped using one or more expressions. In particular, data can be recoded and reshaped to expand the data analysis and processing perspective.

Conditional Logic

As experienced PROC SQL programmers know, it is often necessary to test and evaluate conditions as true or false. From a programming perspective, the evaluation of a condition determines which of the alternate paths a program will follow. Conditional logic for selecting rows in one or more tables in the SQL procedure is most frequently specified using a WHERE or ON clause, or a WHERE or HAVING clause, to reference constants and

relationships among columns, values, or aggregates. Essentially, a WHERE, ON, or HAVING clause along with its associated expression defines a condition for selecting rows or aggregates from one or more tables.

The SQL procedure also allows the identification and assignment of data values in a SELECT statement using CASE expressions (which are described in the next section). To show how constants and relationships are referenced in a WHERE clause, a number of examples will be presented including a single column (variable) name or constant, a SAS function, a predicate, and a compound expression consisting of a series of simple expressions.

WHERE versus ON Clause

Conversations frequently arise about whether a WHERE clause or an ON clause should be specified when a query performs a join on two or more tables. Here is a brief and simple explanation of what happens when a WHERE clause is specified versus an ON clause in a join query.

A join query containing a WHERE clause results in SAS performing the filtering operation after the tables have been joined. For example, when a conventional inner join operation containing a WHERE clause is executed the tables are first joined, and then filtered, followed by the results being produced. You are asked to contrast these operations with a join query that contains an ON clause. A join query containing an ON clause results in one or both of the tables being filtered prior to being joined. As a result, the flow of operations when a WHERE clause is specified differs from when an ON clause is specified in a join query.

WHERE versus HAVING Clause

When specified, the optional WHERE and HAVING clause applies subsetting conditions on the rows selected from the table(s) specified in the FROM clause. A WHERE clause is specified to process rows of data using any valid SAS expression, the specification of an aggregate (e.g., COUNT, MIN, MAX, etc.) is not allowed. To process aggregated data, a HAVING clause is specified in place of a WHERE clause. Note: As was described in Chapter 2, “Working with Data in PROC SQL,” a WHERE clause is specified before a GROUP BY clause (pre-filter), and a HAVING clause is specified after a GROUP BY clause (post-filter).

The next example shows a WHERE clause subsetting products from the PRODUCTS table that cost less than \$400.00. During execution, the expression evaluates to true when the value of PRODCOST is less than \$400.00. Otherwise, when the value of PRODCOST is greater than or equal to \$400.00, the expression evaluates to false. This is an important concept because data rows are only selected when the WHERE clause expression evaluates to true.

SQL Code

```
PROC SQL;
  SELECT *
  FROM PRODUCTS
    WHERE PRODCOST < 400 . 00;
QUIT;
```

In the next example, the following relation evaluates whether the cost of a product (PRODCOST) is greater than \$400.00. When the WHERE clause expression evaluates to

true, which means that PRODCOST is greater than \$400.00, then the rows of data are selected. Otherwise, when the value is less than or equal to \$400.00, the expression evaluates to false.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
      WHERE PRODCOST > 400.00;
QUIT;
```

A relation can also be used with nonnumeric literals and nonnumeric columns. In the next example, a case-sensitive expression is constructed to represent the type of product (PRODTYPE) made by a manufacturer. When evaluated, a condition of true or false is produced depending on whether the current value of PRODTYPE is identical (character-by-character) to the literal value “Software”. When a condition of true occurs, then the rows of data satisfying the expression are selected; otherwise, they are not selected.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
      WHERE PRODTYPE = "Software";
QUIT;
```

To ensure a character-by-character match of a character value, the previous expression could be specified in a WHERE clause with the UPCASE function as follows.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) = "SOFTWARE";
QUIT;
```

The previous query’s conditional expression could also be specified in a HAVING clause with the UPCASE function as follows.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
      HAVING UPCASE(PRODTYPE) = "SOFTWARE";
QUIT;
```

When the relations < and > are defined for nonnumeric values, the issue of implementation-dependent collating sequence for characters comes into play. For example, “A” < “B” is true, “Y” < “Z” is “true”, “B” < “A” is “false”, and so on. For more information about character collating sequences, refer to your specific operating system documentation.

To continue contrasting the differences between a WHERE clause and a HAVING clause, the next example specifies a WHERE clause to count and subset the product types from the PRODUCTS table so that product types containing four or more products are displayed. As can be seen in the SAS log results, unfortunately, the SAS System stopped processing this query because the use of summary functions in a WHERE clause is not permitted.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
        ,PRODTYPE
        ,PRODCOST
  FROM PRODUCTS
  WHERE COUNT(PRODTYPE) > 3
    GROUP BY PRODTYPE
    ORDER BY PRODNAME;
QUIT;
```

SAS Log Results

```
PROC SQL;
  SELECT PRODNAME
        ,PRODTYPE
        ,PRODCOST
  FROM PRODUCTS
  WHERE COUNT(PRODTYPE) > 3
    GROUP BY PRODTYPE
    ORDER BY PRODNAME;
ERROR: Summary functions are restricted to the SELECT and HAVING
clauses only.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax
of statements.

QUIT;
NOTE: The SAS System stopped processing this step because of errors.
```

To successfully count and subset products containing four or more product types, the next query replaces the WHERE clause in the previous example with a HAVING clause to avoid the restrictions noted earlier. As shown in the results, the products matching the post-filtering operation performed by the HAVING clause are selected and displayed without error.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
        ,PRODTYPE
        ,PRODCOST
  FROM PRODUCTS
  GROUP BY PRODTYPE
  HAVING COUNT(PRODTYPE) > 3
    ORDER BY PRODNAME;
QUIT;
```

Results

Product Name	Product Type	Product Cost
Database Software	Software	\$399.00
Graphics Software	Software	\$299.00
Spreadsheet Software	Software	\$299.00
Wordprocessor Software	Software	\$299.00

Conditional Logic with Predicates (Operators)

Conditional logic and the use of predicates (e.g., IN, BETWEEN, and CONTAINS) provide WHERE and HAVING clauses with added value and flexibility. In Hermansen and Legum (2008), the authors describe using predicates to perform complex table look-up, subsetting, and other operations. Predicates are used in WHERE and HAVING clauses when a Boolean value is necessary. From fully bounded range conditions, and operators such as IN, BETWEEN, and CONTAINS, predicates provide users with a way to streamline WHERE and HAVING clause expressions.

In the next example, a WHERE clause with an UPCASE function and an OR logical operator is constructed to select rows matching the literal value “LAPTOP” or “WORKSTATION”. When a condition of true occurs for either value, then the rows of data satisfying the expression are selected; otherwise, they are not selected.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = "LAPTOP" OR UPCASE(PRODTYPE) =
 "WORKSTATION";
QUIT;
```

However, an easier and more convenient way of specifying the WHERE clause in the previous example is to use an IN operator to select product types from the PRODUCTS table that match the list of character values, as follows.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) IN ("LAPTOP", "WORKSTATION");
QUIT;
```

In the next example, a WHERE clause specifies a fully bounded range condition that selects and orders products that cost between \$200 and \$400.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
```

```

WHERE 200 <= PRODCOST <= 400
      ORDER BY PRODCOST, PRODNAME;
QUIT;

```

However, a more convenient way of specifying the WHERE clause in the previous example is to use a BETWEEN operator which selects and orders products from the PRODUCTS table that cost between \$200 and \$400, as follows.

SQL Code

```

PROC SQL;
  SELECT *
  FROM PRODUCTS
    WHERE PRODCOST BETWEEN 200 AND 400
          ORDER BY PRODCOST, PRODNAME;
QUIT;

```

Another handy operator to use in a WHERE or HAVING clause is CONTAINS. The CONTAINS operator selects rows by searching for a specified set of characters contained in a character variable. The next example selects the company names from the CUSTOMERS table that contain the characters “TECH” in the customer name, as follows.

SQL Code

```

PROC SQL;
  SELECT *
  FROM CUSTOMERS
    WHERE UPCASE(CUSTNAME) CONTAINS "TECH"
          ORDER BY CUSTNAME;
QUIT;

```

The next example shows how a NOT logical operator can be specified to select the company names from the CUSTOMERS table that do not contain the characters “TECH” in the customer name, as follows.

SQL Code

```

PROC SQL;
  SELECT *
  FROM CUSTOMERS
    WHERE UPCASE(CUSTNAME) NOT CONTAINS "TECH"
          ORDER BY CUSTNAME;
QUIT;

```

CASE Expressions

In the SQL procedure, a CASE expression provides a way of determining what the resulting value will be from all the rows in a table (or view). Similar to a DATA step SELECT statement (or IF-THEN/ELSE statement), a CASE expression is based on some condition and the condition uses a WHEN-THEN clause to determine what the resulting value will be. An optional ELSE expression can be specified to handle an alternative action if none of the expression(s) identified in the WHEN condition(s) is satisfied.

The SQL procedure supports two forms of CASE expressions: simple and searched. CASE expressions can be specified in a SELECT clause, a WHERE clause, an ORDER BY clause, a HAVING clause, a join construct, and anywhere an expression can be used. A CASE expression must be a valid PROC SQL expression and conform to syntax rules similar to DATA step SELECT-WHEN statements. Before specific CASE expression examples are shown, it's important to illustrate the basic syntax, which is shown below.

```
CASE <column-name>
    WHEN when-condition THEN result-expression
    <WHEN when-condition THEN result-expression> ...
    <ELSE result-expression>
END
```

The CASE syntax contains everything from CASE to END. A column-name can optionally be specified as part of the CASE expression. If present, it is automatically made available to each WHEN condition. When it is not specified, the column name must be coded in each WHEN condition. Let's examine how a CASE expression works.

One or more WHEN conditions can be specified in a CASE expression and are evaluated in the order listed. If a WHEN condition is satisfied by a row in a table (or view), then it is considered “true” and the result expression following the THEN keyword is processed. The remaining WHEN conditions in the CASE expression are skipped. If a WHEN condition is “false,” the next WHEN condition is evaluated. SQL evaluates each WHEN condition until a “true” condition is found. Or, in the event that all WHEN conditions are “false,” it then executes the ELSE expression and assigns its value to the CASE expression’s result. A missing value is assigned to a CASE expression when an ELSE expression is not specified and each WHEN condition is “false.”

Simple CASE Expression

As its name implies, a simple CASE expression provides a useful way to perform the simplest type of comparisons. The syntax requires a column name from an underlying table to be specified as part of the CASE expression. This not only eliminates having to continually repeat the column name in each WHEN condition, it also reduces the number of keystrokes, making the code easier to read (and support). Simple CASE expressions possess the following features:

- Allows only equality checks.
- Evaluates the specified WHEN conditions in the order specified.
- Evaluates the input-expression for each WHEN condition.
- Returns the result-expression of the first input-expression that evaluates to “true.”
- If no input-expressions evaluate to “true,” then the ELSE condition is processed.
- When an ELSE condition isn’t specified, a NULL value is assigned.

Simple CASE Expression in a SELECT Clause

To best show how a simple CASE expression works, an example in a SELECT clause is illustrated. The objective calls for the value assignment of “East” to manufacturers in Florida, “Central” to manufacturers in Texas, “West” to manufacturers in California, or “Unknown” to manufacturers not residing in Florida, Texas, and California. The manufacturer’s state of

residence (MANUSTAT) column is specified in the CASE expression along with its associated WHEN conditions with assigned values. Finally, a column heading of Region is assigned to the derived output column using the AS keyword.

SQL Code

```
PROC SQL;
  SELECT MANUNAME,
         MANUSTAT,
    CASE MANUSTAT
      WHEN 'CA' THEN 'West'
      WHEN 'FL' THEN 'East'
      WHEN 'TX' THEN 'Central'
      ELSE 'Unknown'
    END AS Region
  FROM MANUFACTURERS;
QUIT;
```

Results

The SAS System		
Manufacturer Name	Manufacturer State	Region
Cupid Computer	TX	Central
Global Comm Corp	CA	West
World Internet Corp	FL	East
Storage Devices Inc	CA	West
KPL Enterprises	CA	West
San Diego PC Planet	CA	West

The next example illustrates the process of classifying products with a simple CASE expression in a SELECT clause. The PRODTYPE column from the PRODUCTS table is used to assign a character value of “Hardware,” “Software,” or “Unknown” to each product type (e.g., Laptop, Phone, Software, and Workstation). Similar to the assignment process in the FORMAT procedure, new data values are associated with values in the PRODTYPE column. The WHEN-THEN conditions equate “Laptop” to “Hardware,” “Phone” to “Hardware,” “Software” to “Software,” and “Workstation” to “Hardware.” A value of “Unknown” is assigned to products not matching any of the WHEN-THEN logic conditions. Finally, a column heading of Product_Classification is assigned to the new column with the AS keyword.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
    CASE PRODTYPE
      WHEN 'Laptop'      THEN 'Hardware'
```

```

WHEN 'Phone'      THEN 'Hardware'
WHEN 'Software'   THEN 'Software'
WHEN 'Workstation' THEN 'Hardware'
ELSE 'Unknown'
END AS Product_Classification
FROM PRODUCTS;
QUIT;

```

Results

The SAS System

Product Name	Product_Classification
Dream Machine	Hardware
Business Machine	Hardware
Travel Laptop	Hardware
Analog Cell Phone	Hardware
Digital Cell Phone	Hardware
Office Phone	Hardware
Spreadsheet Software	Software
Database Software	Software
Wordprocessor Software	Software
Graphics Software	Software

The next example classifies products (e.g., Laptop/Workstation, Phone, and Software) in the PRODUCTS table using a simple CASE expression. A PUT function converts the numeric-defined PRODNUM column to a character value, and the SUBSTR function then extrapolates the first position in the PRODNUM column for classification purposes in the WHEN-THEN logic conditions. Finally, a column heading is assigned to the output column with the AS keyword.

SQL Code

```

PROC SQL;
SELECT PRODNAME,
       PRODNUM,
CASE SUBSTR(PUT(PRODNUM,4.),1,1)
    WHEN '1' THEN 'Laptop/Workstation'
    WHEN '2' THEN 'Phone'
    WHEN '5' THEN 'Software'
    ELSE 'Unknown'
END AS Product_Classification
FROM PRODUCTS
ORDER BY PRODNUM;
QUIT;

```

Results

The SAS System

Product Name	Product Number	Product_Classification
Dream Machine	1110	Laptop/Workstation
Business Machine	1200	Laptop/Workstation
Travel Laptop	1700	Laptop/Workstation
Analog Cell Phone	2101	Phone
Digital Cell Phone	2102	Phone
Office Phone	2200	Phone
Spreadsheet Software	5001	Software
Database Software	5002	Software
Wordprocessor Software	5003	Software
Graphics Software	5004	Software

The next example applies a subquery (for more information, see the “Subqueries” section in Chapter 7, “Coding Complex Queries”) to subset software products from the Product_Classification results created in the previous example.

SQL Code

```

PROC SQL;
  SELECT *
  FROM
    (SELECT PRODNAME,
           PRODNUM,
           CASE SUBSTR(PUT(PRODNUM,4.),1,1)
             WHEN '1' THEN 'Laptop/Workstation'
             WHEN '2' THEN 'Phone'
             WHEN '5' THEN 'Software'
             ELSE 'Unknown'
           END AS Product_Classification
    FROM PRODUCTS
  )
  WHERE Product_Classification = 'Software';
QUIT;

```

Results

The SAS System

Product Name	Product Number	Product_Classification
Spreadsheet Software	5001	Software
Database Software	5002	Software
Wordprocessor Software	5003	Software
Graphics Software	5004	Software

Simple CASE Expression in a WHERE Clause

In the previous section, we examined the syntax and application of simple CASE expressions in a SELECT clause. In this section, we'll explore the syntax and application of simple CASE expressions in a WHERE clause. Because the SQL procedure supports the use of a CASE expression anywhere an expression can be used, we'll learn that a query can benefit from the capabilities of passing a result value from a CASE expression directly to a WHERE clause in place of a hardcoded value.

SQL Code

```
PROC SQL;
  SELECT MANUNAME,
         MANUSTAT
    FROM MANUFACTURERS
   WHERE CASE MANUSTAT
         WHEN 'CA' THEN 1
         ELSE 0
      END ;
QUIT;
```

Results

The SAS System

Manufacturer Name	Manufacturer State
Global Comm Corp	CA
Storage Devices Inc	CA
KPL Enterprises	CA
San Diego PC Planet	CA

Creating a Customized List with a Simple CASE Expression

A customized display and order can be coded using a simple CASE expression in a SELECT and ORDER BY clause. The following example illustrates a unique way to create a list of products and availability by coding individual Case logic for each product in the PRODUCTS

table. The resulting output displays a value of “1” to indicate that it contributed to the output list, or a value of “0” to indicate that it didn’t contribute to the output list.

SQL Code

```
OPTIONS LS=120;

PROC SQL;

SELECT PRODNAME,
       PRODCOST,
       CASE PRODTYPE WHEN 'Laptop'      THEN 1 ELSE 0 END
                     AS LaptopRequest,
       CASE PRODTYPE WHEN 'Workstation' THEN 1 ELSE 0 END
                     AS WorkstationRequest,
       CASE PRODTYPE WHEN 'Phone'       THEN 1 ELSE 0 END
                     AS PhoneRequest,
       CASE PRODTYPE WHEN 'Software'    THEN 1 ELSE 0 END
                     AS SoftwareRequest
FROM PRODUCTS
ORDER BY CASE PRODTYPE WHEN 'Laptop'      THEN 1 ELSE 0 END,
         CASE PRODTYPE WHEN 'Workstation' THEN 1 ELSE 0 END,
         CASE PRODTYPE WHEN 'Phone'       THEN 1 ELSE 0 END,
         CASE PRODTYPE WHEN 'Software'    THEN 1 ELSE 0 END;
QUIT;
```

Results

The SAS System						
Product Name	Product Cost	LaptopRequest	WorkstationRequest	PhoneRequest	SoftwareRequest	
Wordprocessor Software	\$299.00	0	0	0	1	
Graphics Software	\$299.00	0	0	0	1	
Spreadsheet Software	\$299.00	0	0	0	1	
Database Software	\$399.00	0	0	0	1	
Office Phone	\$130.00	0	0	1	0	
Digital Cell Phone	\$175.00	0	0	1	0	
Analog Cell Phone	\$35.00	0	0	1	0	
Dream Machine	\$3,200.00	0	1	0	0	
Business Machine	\$3,300.00	0	1	0	0	
Travel Laptop	\$3,400.00	1	0	0	0	

Simple CASE Expression in a HAVING Clause

A HAVING clause is used in an SQL query to filter out aggregates specified in a GROUP BY clause. In this example, the SELECT clause returns the names of the manufacturers along with their state of residence from the MANUFACTURERS table. The CASE expression in the HAVING clause restricts what is returned by the SELECT clause to only the manufacturers residing in the state of California.

SQL Code

```
PROC SQL;
  SELECT MANUNAME,
         MANUSTAT
    FROM MANUFACTURERS
   HAVING CASE MANUSTAT
          WHEN 'CA' THEN 1
          ELSE 0
        END ;
QUIT;
```

Results

The SAS System	
Manufacturer Name	Manufacturer State
Global Comm Corp	CA
Storage Devices Inc	CA
KPL Enterprises	CA
San Diego PC Planet	CA

In the next example, the SELECT clause returns the most expensive product(s) from the PRODUCTS table, and the CASE expression in the HAVING clause restricts the results to software products costing more than \$300.00.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         MAX(PRODCOST) AS Product_Cost FORMAT=DOLLAR10.2
    FROM PRODUCTS
   HAVING CASE PRODTYPE
             WHEN 'Software' THEN PRODCOST
             ELSE 0
           END > 300;
QUIT;
```

Results

The SAS System		
Product Name	Product Type	Product_Cost
Database Software	Software	\$3,400.00

Simple CASE Expression in a Join Construct

A CASE expression can be specified in a JOIN construct to filter out aggregates specified in a GROUP BY clause. In this example, the SELECT clause returns the names of the manufacturers along with their state of residence from the MANUFACTURERS table. The CASE expression in the HAVING clause restricts what is returned by the SELECT clause to just the manufacturers residing in the state of California.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         MANUNAME
    FROM PRODUCTS P,
         MANUFACTURERS M
   WHERE P.MANUNUM = M.MANUNUM AND
         CASE PRODTYPE
             WHEN 'Software' THEN 1
             ELSE 0
           END;
QUIT;
```

Results

The SAS System

Product Name	Product Type	Manufacturer Name
Spreadsheet Software	Software	KPL Enterprises
Database Software	Software	KPL Enterprises
Wordprocessor Software	Software	KPL Enterprises
Graphics Software	Software	KPL Enterprises

Preventing Division by Zero with a Simple CASE Expression

Division by zero errors can cause many problems for a query, including severe performance degradation. In the next example, a simple CASE expression in a SELECT clause is specified to prevent division by zero errors. Using the INVENTORY table, a CASE expression tells SAS to ignore performing any computations for products containing an INVENQTY value of zero (by assigning a Boolean value of false). Otherwise, the equation INVENCST / INVENQTY is computed and products costing more than \$1,000.00 are selected by the WHERE clause. A column heading called, Average_Cost is assigned to the new column with the AS keyword.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         INVENCST,
         CASE INVENQTY
           WHEN 0 THEN 0
           ELSE INVENCST / INVENQTY
        END AS Average_Cost FORMAT=DOLLAR10.2
  FROM INVENTORY
  WHERE INVENCST / INVENQTY > 1000;
QUIT;
```

Results

The SAS System

Product Number	Inventory Quantity	Inventory Cost	Average_Cost
1110	20	\$45,000.00	\$2,250.00
1700	10	\$28,000.00	\$2,800.00

In the next example, a simple CASE expression is implemented in a WHERE clause to prevent division by zero errors. As before, the CASE expression's WHEN condition tells SAS to ignore performing any computations for products containing an INVENQTY value of zero (by assigning a Boolean value of false). Otherwise, the equation INVENCST /

INVENQTY is computed and products costing more than \$1,000.00 are selected by the WHERE clause.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
         INVENQTY,
         INVENCST
    FROM INVENTORY
   WHERE CASE INVENQTY
         WHEN 0 THEN 0
         ELSE INVENCST / INVENQTY
      END > 1000;
QUIT;
```

Results

The SAS System		
Product Number	Inventory Quantity	Inventory Cost
1110	20	\$45,000.00
1700	10	\$28,000.00

Nesting Simple CASE Expressions

Much has been written on the techniques associated with software construction, particularly about aspects related to program and/or code design. When discussions of program complexity arise, it is widely accepted that a program's degree of complexity can often be reduced by dividing its solution into smaller pieces or parts. This rule of thinking is perhaps best discussed in Steve McConnell's book, *Code Complete: A Practical Handbook of Software Construction, Second Edition* (2004), "Humans tend to have an easier time comprehending several simple pieces of information than one complicated piece."

Although many design strategies exist to improve the process of program design, one strategy frequently mentioned is the process of nesting. Nesting involves converting complex and/or cumbersome logic scenarios into two (or more) simpler logic conditions. The primary objective is to improve a program's maintainability and readability. This rule of thought is further reinforced by the SQL procedure's support for nesting using a CASE expression construct.

Nesting can be a useful technique because it provides SQL users with an effective way to handle complex code. But, unnecessary nesting or nesting just for the sake of nesting can actually make code more difficult to comprehend and maintain. It's been my experience that nesting should never exceed three or four levels deep. Anything more than that and the degree of complexity can increase significantly. The general rule is to develop code that's not only easy to understand, but hopefully contains fewer errors. To illustrate the process of nesting, a simple CASE expression is nested two levels deep for identifying software products in the PRODUCTS table that cost less than \$300.00.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         PRODTYPE,
         PRODCOST,
         CASE PRODTYPE
           WHEN 'Software' THEN
             CASE
               WHEN PRODCOST < 300 THEN 'Match'
               ELSE 'No Match'
             END
           ELSE 'Not Software'
         END AS Product_Type_Cost
   FROM PRODUCTS;
QUIT;
```

Results

The SAS System

Product Name	Product Type	Product Cost	Product_Type_Cost
Dream Machine	Workstation	\$3,200.00	Not Software
Business Machine	Workstation	\$3,300.00	Not Software
Travel Laptop	Laptop	\$3,400.00	Not Software
Analog Cell Phone	Phone	\$35.00	Not Software
Digital Cell Phone	Phone	\$175.00	Not Software
Office Phone	Phone	\$130.00	Not Software
Spreadsheet Software	Software	\$299.00	Match
Database Software	Software	\$399.00	No Match
Wordprocessor Software	Software	\$299.00	Match
Graphics Software	Software	\$299.00	Match

Conditionally Updating a Table with a Simple CASE Expression

A CASE expression can be specified to conditionally update the contents of a table. For more information about updating data in a table, see Chapter 6, “Modifying and Updating Tables and Indexes.” The example illustrates the process of conditionally updating data in a table using an UPDATE statement and a simple CASE expression to add \$10.00 the unit cost (UNITCOST) for a “Chair” in the PURCHASES table from \$179.00 to \$189.00.

Note: A table must be open in update mode to be able to update its contents.

SQL Code

```
PROC SQL;
  TITLE1 'Before Update Operation';
  SELECT *
    FROM PURCHASES2;
```

```

UPDATE PURCHASES2
  SET UNITCOST = UNITCOST +
    CASE ITEM
      WHEN 'Chair' THEN 10.00
      ELSE 0
    END;
TITLE1 'After Update Operation';
SELECT *
  FROM PURCHASES2;
QUIT;

```

Results

Before Update Operation

Custnum	Item	Units	Unitcost
1	Chair	1	\$179.00
1	Pens	12	\$0.89
1	Paper	4	\$6.95
1	Stapler	1	\$8.95
7	Mouse Pad	1	\$11.79
7	Pens	24	\$1.59
13	Markers	.	\$0.99

After Update Operation

Custnum	Item	Units	Unitcost
1	Chair	1	\$189.00
1	Pens	12	\$0.89
1	Paper	4	\$6.95
1	Stapler	1	\$8.95
7	Mouse Pad	1	\$11.79
7	Pens	24	\$1.59
13	Markers	.	\$0.99

Searched CASE Expression

A searched CASE expression provides SQL users with the capability to perform more complex comparisons. Although the number of keystrokes can be more than with a simple

CASE expression, the searched CASE expression offers the greatest flexibility and is the primary form used by SQL programmers. The noticeable absence of a column name as part of the CASE expression permits any number of columns to be specified from the underlying table(s) in the WHEN-THEN/ELSE logic scenarios.

The searched CASE expression evaluates one or more WHEN conditions until it encounters one that evaluates to “true.” It then assigns the corresponding result expression specified following the THEN keyword. If none of the WHEN conditions evaluates to “true,” then the result following the specified ELSE keyword is returned. If none of the WHEN conditions evaluates to “true” and an ELSE condition is not specified, then the searched CASE expression returns a null value. Searched CASE expressions possess the following features:

- Allow arithmetic operators (i.e., =, <, <=, >, >=, etc.).
- Use logical operators (i.e., AND, OR, and NOT) for combining any number of expressions.
- Evaluate compound expressions in the following default order: NOT, AND, or OR, unless parentheses are specified to control the order of evaluation.
- Evaluate the specified WHEN conditions in the order specified.
- Evaluate the input-expression for each WHEN condition.
- Return the result-expression of the first input-expression that evaluates to “true.”
- Process the ELSE condition if no input-expressions evaluate to “true.”
- Assign a NULL value when an ELSE condition isn’t specified.

Searched CASE Expression in a SELECT Clause

To illustrate how a searched CASE expression works, consider an example that uses a calculated dollar amount from the UNITS and UNITCOST columns to assign a value of “Small Purchase” for purchases less than \$1,000, “Average Purchase” for purchases between \$1,000 and \$7,500, “Large Purchase” for purchases greater than \$7,500, and “Unknown Purchase” for all other values in the PURCHASES table. Using the calculated amount, Purchase_Amount, from the UNITS and UNITCOST columns, a CASE expression is constructed to assign the desired value in each row of data. Finally, a column heading of Type_of_Purchase is assigned to the calculated column with the AS keyword. **Note:** The CALCULATED keyword must be specified with any column created in a SELECT clause that does not exist in the table referenced in the FROM clause.

SQL Code

```

PROC SQL;
  SELECT
    PRODNUM
    UNITS,
    UNITCOST,
    UNITS * UNITCOST
      AS Purchase_Amount
    FORMAT=DOLLAR12.2,
    CASE
      WHEN CALCULATED
Purchase_Amount < 1000 THEN 'Small Purchase'
      WHEN CALCULATED
Purchase_Amount BETWEEN 1000 AND 7500 THEN 'Average Purchase'
      WHEN CALCULATED
Purchase_Amount > 7500 THEN 'Large Purchase'
      ELSE 'Unknown Purchase'
    END AS Type_of_Purchase
  FROM PURCHASES
    ORDER BY CALCULATED
Purchase_Amount DESC;
QUIT;

```

Results

The SAS System

Manufacturer Name	Manufacturer State	Region
Cupid Computer	TX	Central
Global Comm Corp	CA	West
World Internet Corp	FL	East
Storage Devices Inc	CA	West
KPL Enterprises	CA	West
San Diego PC Planet	CA	West

Complex Comparisons with Searched CASE Expressions

As described earlier, searched CASE expressions provide SQL users with the capability to perform more complex comparisons. Combined with logical and comparison operators, CASE expressions along with their WHERE clause counterparts provide the capabilities to construct complex logic scenarios. In the next example, a listing of manufacturers and their products are displayed using a searched CASE expression to assign a value of “East” to manufacturers in Florida, “Central” to manufacturers in Texas, “West” to manufacturers in California, and “Unknown” to all other manufacturers in the MANUFACTURERS table. Using the manufacturer’s state of residence (MANSTAT) column, a CASE expression is constructed to assign the desired value in each row of data. Finally, a column heading of Region is assigned to the new column with the AS keyword.

SQL Code

```
PROC SQL;
  SELECT MANUNAME,
    CASE
      WHEN MANSTAT IN ('FL','TX','CA') THEN
        CASE
          WHEN PRODTYPE IN ('Laptop', 'Workstation') THEN
            "Computer Hardware Manufacturer"
          ELSE "Not a Computer Manufacturer"
        END
      ELSE "Unknown Manufacturer"
    END AS ManufacturerType,
    PRODNAME,
    PRODCOST
  FROM MANUFACTURERS M,
    PRODUCTS P
  WHERE M.MANUNUM = P.MANUNUM;
QUIT;
```

Results

The SAS System

Manufacturer Name	ManufacturerType	Product Name	Product Cost
Cupid Computer	Computer Hardware Manufacturer	Dream Machine	\$3,200.00
Storage Devices Inc	Computer Hardware Manufacturer	Business Machine	\$3,300.00
Global Comm Corp	Not a Computer Manufacturer	Analog Cell Phone	\$35.00
Global Comm Corp	Not a Computer Manufacturer	Digital Cell Phone	\$175.00
KPL Enterprises	Not a Computer Manufacturer	Spreadsheet Software	\$299.00
KPL Enterprises	Not a Computer Manufacturer	Database Software	\$399.00
KPL Enterprises	Not a Computer Manufacturer	Wordprocessor Software	\$299.00
KPL Enterprises	Not a Computer Manufacturer	Graphics Software	\$299.00

Creating a Customized List with a Searched CASE Expression

Similar to the simple CASE expression illustrated earlier, a customized display can be specified using a searched CASE expression in a SELECT clause. The following example illustrates a unique way to create a list of products and availability by coding individual Case logic conditions for each product in the PRODUCTS table. The resulting output displays a value of “1” to indicate that it contributed to the output list, or a value of “0” to indicate that it didn’t contribute to the output list.

SQL Code

```
OPTIONS LS=120;
PROC SQL;
  SELECT PRODNAME,
         PRODCOST,
         CASE WHEN PRODTYPE='Laptop' AND
               PRODCOST < 1000 THEN 1
                  ELSE 0
            END AS InexpensiveLaptop,
         CASE WHEN PRODTYPE='Laptop' AND
               PRODCOST BETWEEN 1000 AND 2500 THEN 1
                  ELSE 0
            END AS MediumPricedLaptop,
         CASE WHEN PRODTYPE='Laptop' AND
               PRODCOST > 2500 THEN 1
                  ELSE 0
            END AS ExpensiveLaptop
    FROM PRODUCTS
   WHERE PRODTYPE='Laptop'
   ORDER BY PRODCOST;
      ELSE 0
    END;
QUIT;
```

Results

The SAS System				
Product Name	Product Cost	InexpensiveLaptop	MediumPricedLaptop	ExpensiveLaptop
Travel Laptop	\$3,400.00	0	0	1

Another Customized List with a Searched CASE Expression

Extending the features from the previous searched CASE expression example, a customized list is created. In this example, a unique approach is used to create a list of products with their respective inventory quantities. The query performs an equijoin on the PRODUCTS and INVENTORY tables with Case logic conditions nested two levels deep to capture the inventory quantities for each product in the PRODUCTS table.

SQL Code

```
PROC SQL;
  SELECT PRODNAME,
         P.PRODNUM,
         CASE WHEN SUBSTR(PUT(P.PRODNUM,4.),1,1) = '1' THEN
```

```

CASE WHEN INVENQTY > 0 THEN INVENQTY
END
END AS ComputerHardware,
CASE WHEN SUBSTR(PUT(P.PRODNUM,4.),1,1) = '2' THEN
    CASE WHEN INVENQTY > 0 THEN INVENQTY
    END
END AS OfficeEquipment,
CASE WHEN SUBSTR(PUT(P.PRODNUM,4.),1,1) = '5' THEN
    CASE WHEN INVENQTY > 0 THEN INVENQTY
    END
END AS ComputerSoftware
FROM PRODUCTS P,
     INVENTORY I
WHERE P.PRODNUM = I.PRODNUM
      ORDER BY PRODNAME;
QUIT;

```

Results

The SAS System

Product Name	Product Number	ComputerHardware	OfficeEquipment	ComputerSoftware
Database Software	5002	.	.	3
Dream Machine	1110	20	.	.
Graphics Software	5004	.	.	20
Spreadsheet Software	5001	.	.	5
Spreadsheet Software	5001	.	.	2
Travel Laptop	1700	10	.	.
Wordprocessor Software	5003	.	.	10

Case Logic versus COALESCE Expression

A popular convention among SQL programmers is to specify a COALESCE function in an expression to perform Case logic. As described in Chapter 2, “Working with Data in PROC SQL,” the COALESCE function permits a new value to be substituted for one or more missing column values. By specifying COALESCE in an expression, PROC SQL evaluates each argument from left to right for the occurrence of a non-missing value. The first non-missing value found in the list of arguments is returned; otherwise, a missing value, or assigned value, is returned. This approach not only saves programming time, it makes coding constructs simpler to maintain.

Expressing logical expressions in one or more WHEN-THEN/ELSE conditions are frequently easy to code, understand, and maintain. But as the complexities associated with Case logic increase, the amount of coding also increases. In the following example, a simple CASE expression is presented to illustrate how a value of “Unknown” is assigned and displayed when CUSTCITY is missing.

SQL Code

```
PROC SQL;
  SELECT CUSTNAME,
    CASE
      WHEN CUSTCTY IS NOT NULL THEN CUSTCITY
      ELSE 'Unknown'
    END AS Customer_City
  FROM CUSTOMER;
QUIT;
```

To illustrate the usefulness of the COALESCE function as an alternative to Case logic, the same query can be modified to achieve the same results as before. By replacing the Case logic with a COALESCE expression as follows, the value of CUSTCITY is automatically displayed unless it is missing. In cases of character data, a value of “Unknown” is displayed. This technique makes the COALESCE function a very useful and is a shorthand approach indeed.

SQL Code

```
PROC SQL;
  SELECT CUSTNAME,
    COALESCE(CUSTCTY, 'Unknown')
    AS Customer_City
  FROM CUSTOMER;
QUIT;
```

In cases where a COALESCE expression is used with numeric data, the value assigned or displayed must be of the same type as the expression. The next example shows a value of “0” (zero) being assigned and displayed when units (UNITS) from the PURCHASES table are processed.

SQL Code

```
PROC SQL;
  SELECT ITEM,
    COALESCE(UNITS, 0)AS Units
  FROM PURCHASES2;
QUIT;
```

Results

The SAS System

Item	Units
Chair	1
Pens	12
Paper	4
Stapler	1
Mouse Pad	1
Pens	24
Markers	0

Assigning Labels and Grouping Data

The ability to assign data values and group data based on the existence of distinct values for specified table columns is a popular and frequently useful operation. Suppose that you want to assign a specific data value and then group the output based on this assigned value. As a savvy SAS user, you are probably thinking, “Hey, this is easy—I’ll just create a user-defined format and use it in the PRINT or REPORT procedure.”

In the next example, the FORMAT procedure is used to assign temporary formatted values based on a range of values for INVENQTY. The result from executing this simple three-step (non-SQL procedure) program shows that the actual INVENQTY value is temporarily replaced with the “matched” value in the user-defined format. The FORMAT statement performs a look-up process to determine how the data should be displayed. The actual data value being looked up is not changed (or altered) during the process, but a determination is made as to how its value should be displayed. The BY statement specifies how BY-group processing is to be constructed. The displayed results show the product numbers in relation to their respective inventory quantity status.

Non-SQL Code

```

PROC FORMAT;
  VALUE INVQTY
    0 - 5 = 'Low on Stock - Reorder'
    6 - 10 = 'Stock Levels OK'
    11 - 99 = 'Plenty of Stock'
    100 - 999 = 'Excessive Quantities';
RUN;

PROC SORT DATA=INVENTORY;
  BY INVENQTY;
RUN;

PROC PRINT DATA=INVENTORY (KEEP=PRODNUM INVENQTY) NOOBS;
  FORMAT INVENQTY INVQTY.;
RUN;

```

Results

The SAS System

prodnum	invenqty
5001	Low on Stock - Reorder
5002	Low on Stock - Reorder
5001	Low on Stock - Reorder
1700	Stock Levels OK
5003	Stock Levels OK
1110	Plenty of Stock
5004	Plenty of Stock

The same results can also be derived using a CASE expression in the SQL procedure. In the next example, a CASE expression is constructed using the INVENTORY table to assign values to the user-defined column Inventory_Status. The biggest difference between the FORMAT procedure approach and a CASE expression is that the latter uses one step and does not replace the actual data value with the recoded result. Instead, it creates a new column that contains the result of the CASE expression.

SQL Code

```
PROC SQL;
  SELECT PRODNUM,
    CASE
      WHEN INVENQTY LE 5
        THEN 'Low on Stock - Reorder'
      WHEN 6 LE INVENQTY LE 10
        THEN 'Stock Levels OK'
      WHEN 11 LE INVENQTY LE 99
        THEN 'Plenty of Stock'
      ELSE 'Excessive Quantities'
    END AS Inventory_Status
  FROM INVENTORY
  ORDER BY INVENQTY;
QUIT;
```

Results

The SAS System

Product Number	Inventory_Status
5001	Low on Stock - Reorder
5002	Low on Stock - Reorder
5001	Low on Stock - Reorder
1700	Stock Levels OK
5003	Stock Levels OK
1110	Plenty of Stock
5004	Plenty of Stock

Logic and Nulls

The existence of null values frequently introduces complexities for programmers. Instead of coding two-valued logic conditions, such as “True” and “False,” logic conditions must be designed to handle three-valued logic: “True,” “False,” and “Unknown.” When developing logic conditions, programmers need to be ready to deal with the possibility of having null values. Program logic should test whether the current value of an expression contains a value or is empty (null).

Let’s examine a CASE expression that is meant to handle the possibility of having missing values in a table. Returning to an example presented earlier in this chapter, suppose that you want to assign a value of “South East,” “Central,” “South West,” “Missing,” or “Unknown” to each of the manufacturers based on their state of residence.

SQL Code

```
PROC SQL;
  SELECT MANUNAME,
         MANUSTAT,
    CASE
      WHEN MANUSTAT = 'CA' THEN 'South West'
      WHEN MANUSTAT = 'FL' THEN 'South East'
      WHEN MANUSTAT = 'TX' THEN 'Central'
      WHEN MANUSTAT = ' ' THEN 'Missing'
      ELSE 'Unknown'
    END AS Region
  FROM MANUFACTURERS;
QUIT;
```

Results

The SAS System

Manufacturer Name	Manufacturer State	Region
Cupid Computer	TX	Central
Global Comm Corp	CA	South West
World Internet Corp	FL	South East
Storage Devices Inc	CA	South West
KPL Enterprises	CA	South West
San Diego PC Planet	CA	South West

The results indicate that there were no missing or null values in our database for the column being tested. But, suppose a new row of data was added containing null values in the manufacturer's city and state of residence columns so our new row looked something like following:

```
Manufacturer Number: 800
Manufacturer Name: Spring Valley Products
Manufacturer City: <Missing>
Manufacturer State: <Missing>.
```

If you rerun the previous code, the result would look something like the following.

SQL Code

```
PROC SQL;
  SELECT MANUNAME,
         MANUSTAT,
         CASE
           WHEN MANUSTAT = 'CA' THEN 'South West'
           WHEN MANUSTAT = 'FL' THEN 'South East'
           WHEN MANUSTAT = 'TX' THEN 'Central'
           WHEN MANUSTAT = ''   THEN 'Missing'
           ELSE 'Unknown'
         END AS Region
  FROM MANUFACTURERS;
QUIT;
```

Results

The SAS System

Manufacturer Name	Manufacturer State	Region
Cupid Computer	TX	Central
Global Comm Corp	CA	South West
World Internet Corp	FL	South East
Storage Devices Inc	CA	South West
KPL Enterprises	CA	South West
San Diego PC Planet	CA	South West
Spring Valley Products		Missing

IFC and IFN Functions

Two relatively new and handy functions available to the PROC SQL user are the IFC and IFN functions. These functions provide a convenient way to incorporate conditional logic in a WHERE and HAVING clause, much like the CASE expression in PROC SQL, and the IF-THEN/ELSE and SELECT-WHEN/OTHERWISE statements in the DATA step. Although not part of the SQL ANSI guidelines, the IFC and IFN functions cannot be used with external database connections such as SAS to Oracle, SAS to DB2, SAS to SQL Server, etc., but are useful constructs for encoding, decoding, and flagging values in the construction of conditional logic in a SAS SQL query environment.

IFC and IFN Syntax

The IFC function returns a character value based on whether an expression is true, false, or missing. In contrast, the IFN function returns a numeric value based on whether an expression is true, false, or missing. The specific IFC and IFN functions syntax and arguments are illustrated below.

IFC Syntax:

```
IFC ( logical-expression,
      valueReturnedWhenTrue,
      valueReturnedWhenFalse,
      <valueReturnedWhenMissing> )
```

Arguments for IFC:

Logical-expression specifies a numeric constant, variable, or expression.

ValueReturnedWhenTrue specifies a character constant, variable, or expression when the value of a logical expression is true.

ValueReturnedWhenFalse specifies a character constant, variable, or expression when the value of a logical expression is false.

ValueReturnedWhenMissing is an optional argument that specifies a character constant, variable, or expression when the value of a logical expression is missing.

IFN Syntax:

```
IFN ( logical-expression,
      valueReturnedWhenTrue,
      valueReturnedWhenFalse,
      <valueReturnedWhenMissing> )
```

Arguments for IFN:

Logical-expression specifies a numeric constant, variable, or expression.

ValueReturnedWhenTrue specifies a numeric constant, variable, or expression when the value of a logical expression is true.

ValueReturnedWhenFalse specifies a numeric constant, variable, or expression when the value of a logical expression is false.

ValueReturnedWhenMissing is an optional argument that specifies a numeric constant, variable, or expression when the value of a logical expression is missing.

Application of the IFC and IFN Functions

The IFC and IFN are SAS specific functions that PROC SQL users can enjoy and use. It is worth noting that these functions, as with many other SAS functions, can also be specified as arguments to other functions such as CAT, CATQ, CATS, CATT, and CATX, among others.

SQL Code

```
PROC SQL;
  SELECT PRODNAME
        , INVENQTY
        , IFC(P.PRODNUM = 1700 AND I.INVENQTY < 11, "Yes", "No",
"Missing")
          AS IFC_Results
        , IFN(P.PRODNUM = 1700 AND I.INVENQTY < 11, 1, 0, 999)
          AS IFN_Results
  FROM INVENTORY I
    , PRODUCTS P
  WHERE P.PRODNUM = 1700
    AND I.PRODNUM = 1700;
QUIT;
```

Results

Product Name	Inventory Quantity	IFC_Results	IFN_Results
Travel Laptop	10	Yes	1

Interfacing PROC SQL with the Macro Language

Many software vendors' SQL implementation permits SQL to be interfaced with a host language. The SAS SQL implementation is no different. The SAS macro language enables you customize the way SAS software behaves, and in particular enables you to extend the capabilities of the SQL procedure. PROC SQL users can apply the macro facility's many powerful features by interfacing the SQL procedure with the macro language to provide a wealth of programming opportunities.

From creating and using user-defined macro variables and automatic variables (which are supplied by SAS), reducing redundant code, and performing common and repetitive tasks, to building powerful and simple macro applications, the macro language has the tools PROC SQL users need to improve efficiency. The best part is that you do not have to be a macro language heavyweight to begin reaping the rewards of this versatile interface between two powerful Base SAS software languages.

This section will introduce you to a number of techniques that, with a little modification, could be replicated and used in your own programming environment. You will learn how to use the SQL procedure with macro programming techniques, as well as learn to explore how dictionary tables (see Chapter 2, “Working with Data in PROC SQL,” for details) and the SAS macro facility can be combined with PROC SQL to develop useful utilities to inquire about the operating environment and other information. For more information about the SAS macro language, see *Carpenter’s Complete Guide to the SAS Macro Language, Third Edition* by Art Carpenter; *SAS Macro Programming Made Easy, Second Edition* by Michele M. Burlew; and *SAS Macro Language: Reference* by SAS Institute Inc.

Exploring Macro Variables and Values

Macro variables and their values provide PROC SQL users with a convenient way to store text strings in SAS code. Whether user-defined macro variables are created or automatic macro variables supplied by SAS are referenced, macro variables can be defined and used to improve a program's efficiency and usefulness. A number of useful techniques are presented in this section to illustrate the capabilities afforded users when interfacing PROC SQL with macro variables.

Creating a Macro Variable with %LET

The %LET macro statement creates a single macro variable and assigns or changes a text string value. It can be specified inside or outside a macro and used with PROC SQL. In the next example, a macro variable called PRODTYPE is created with a value of SOFTWARE assigned in a %LET statement. The PRODTYPE macro variable is referenced in the TITLE statement and enclosed in quotation marks in the PROC SQL WHERE clause. This approach of assigning macro variable values at the beginning of a program makes it easy and convenient to make changes because the values are all at the beginning of the program.

SQL Code

```
%LET PRODTYPE=SOFTWARE;
TITLE "Listing of &PRODTYPE Products";
PROC SQL;
  SELECT PRODNAME,
```

```

    PRODCOST
  FROM PRODUCTS
 WHERE UPCASE(PRODTYPE) = "&PRODTYPE"
   ORDER BY PRODCOST;
QUIT;

```

Results

Listing of SOFTWARE Products

Product Name	Product Cost
Wordprocessor Software	\$299.00
Spreadsheet Software	\$299.00
Graphics Software	\$299.00
Database Software	\$399.00

In the next example, a macro named VIEW creates a macro variable called NAME and assigns a value to it with a %LET statement. When VIEW is executed, a value of PRODUCTS, MANUFACTURERS, or INVENTORY is substituted for the macro variable. The value supplied for the macro variable determines what view is referenced. If the value supplied to the macro variable is not one of these three values, then a program warning message is displayed in the SAS log. Invoking the macro with %VIEW(Products) produces the following results.

SQL Code

```

%MACRO VIEW(NAME);

%IF %UPCASE(&NAME) ^= %STR(PRODUCTS) AND
  %UPCASE(&NAME) ^= %STR(MANUFACTURERS) AND
  %UPCASE(&NAME) ^= %STR(INVENTORY) %THEN %DO;
  %PUT A valid view name was not supplied and no output
  will be generated!;

%END;

%ELSE %DO;
  PROC SQL;
    TITLE "Listing of &NAME View";
    %IF %UPCASE(&NAME)=%STR(PRODUCTS) %THEN %DO;
      SELECT PRODNAME,
        PRODCOST

```

```

FROM &NAME._view
ORDER BY PRODCOST;

%END;

%ELSE %IF %UPCASE (&NAME)=%STR (MANUFACTURERS) %THEN %DO;
SELECT MANUNAME,
       MANUCITY,
       MANUSTAT
FROM &NAME._view
ORDER BY MANUCITY;

%END;

%ELSE %IF %UPCASE (&NAME)==%STR (INVENTORY) %THEN %DO;
SELECT PRODNUM,
       INVENQTY,
       INVENCST
FROM &NAME._view
ORDER BY INVENCST;

%END;

QUIT;

%END;

%MEND VIEW;

```

In the previous example, if a name is supplied to the macro variable &NAME that is not valid, then the user-defined program warning message would be displayed in the SAS log. Suppose we invoked the VIEW macro by entering %VIEW(Customers). The results are displayed in the SAS log.

SQL Code

```
%VIEW(Customers);
```

SAS Log Results

```
%VIEW(Customers);
A valid view name was not supplied and no output will be
generated!
```

Creating a Macro Variable from a Table Row Column

A macro variable can be created from a column value in the first row of a table in PROC SQL by specifying the INTO clause. The macro variable is assigned using the value of the column that is specified in the SELECT list from the first row selected. A colon (:) is used in conjunction with the macro variable name being defined. In the next example, output results are suppressed with the NOPRINT option, while two macro variables are created using the INTO clause and their values displayed in the SAS log. **Note:** In the absence of a WHERE clause, the first row in the specified table is the one selected for populating macro variables.

SQL Code

```
PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODNAME,
         :PRODCOST
   FROM PRODUCTS;
QUIT;
%PUT &PRODNAME &PRODCOST;
```

SAS Log Results

```
PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODNAME,
         :PRODCOST
   FROM PRODUCTS;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.38 seconds

%PUT &PRODNAME &PRODCOST;
Dream Machine           $3,200.00
```

In the next example, two macro variables are created using the INTO clause and a WHERE clause to control what row is used in the assignment of macro variable values. Using the WHERE clause enables a row other than the first row to always be used in the assignment of macro variables. Their values are displayed in the SAS log.

SQL Code

```
PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODNAME,
         :PRODCOST
   FROM PRODUCTS
  WHERE UPCASE(PRODTYPE) IN ('SOFTWARE');
QUIT;
%PUT &PRODNAME &PRODCOST;
```

SAS Log Results

```
PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODNAME,
         :PRODCOST
   FROM PRODUCTS
  WHERE UPCASE(PRODTYPE) IN ('SOFTWARE');
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.04 seconds

%PUT &PRODNAME &PRODCOST;
Spreadsheet Software      $299.00
```

Creating a Macro Variable with Aggregate Functions

Turning data into information and then saving the results as macro variables is easy with summary (aggregate) functions. The SQL procedure provides a number of useful summary functions to help perform calculations, descriptive statistics, and other aggregating computations in a SELECT statement or HAVING clause. These functions are designed to summarize information and are not designed to display detail about data. In the next example, the MIN summary function is used to determine the least expensive product from the PRODUCTS table with the value stored in the macro variable MIN_PRODCOST using the INTO clause. The results are displayed in the SAS log.

SQL Code

```
PROC SQL NOPRINT;
  SELECT MIN(PRODCOST) FORMAT=DOLLAR10.2
    INTO :MIN_PRODCOST
   FROM PRODUCTS;
QUIT;
%PUT &MIN_PRODCOST;
```

SAS Log Results

```

PROC SQL NOPRINT;
  SELECT MIN(PRODCOST) FORMAT=DOLLAR10.2
    INTO :MIN_PRODCOST
    FROM SQL.PRODUCTS;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.05 seconds

%PUT &MIN_PRODCOST;
$35.00

```

Creating Multiple Macro Variables

PROC SQL enables you to create a macro variable for each row returned by a SELECT statement. Using the PROC SQL keyword THROUGH or hyphen (-) with the INTO clause, a range of two or more macro variables is easily created. This is a handy feature for creating macro variables from multiple rows in a table. For example, suppose we want to create macro variables for the three least expensive products in the PRODUCTS table. The INTO clause creates three macro variables and assigns values from the first three rows of the PRODNAME and PRODCOST columns. The ORDER BY clause is also specified to perform an ascending sort on product cost (PRODCOST) to assure that the data is in the desired order from least to most expensive. The results are displayed in the SAS log.

SQL Code

```

PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODUCT1 - :PRODUCT3,
         :COST1 - :COST3
    FROM PRODUCTS
    ORDER BY PRODCOST;
QUIT;
%PUT &PRODUCT1 &COST1;
%PUT &PRODUCT2 &COST2;
%PUT &PRODUCT3 &COST3;

```

SAS Log Results

```

PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODUCT1 - :PRODUCT3,
         :COST1 - :COST3
   FROM PRODUCTS
      ORDER BY PRODCOST;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.26 seconds

%PUT &PRODUCT1 &COST1;
Analog Cell Phone $35.00
%PUT &PRODUCT2 &COST2;
Office Phone $130.00
%PUT &PRODUCT3 &COST3;
Digital Cell Phone $175.00

```

Controlling the Selection and Population of Macro Variables with a WHERE Clause

Unlike the previous example where little control is allowed over the rows that are selected for processing, a WHERE clause in a SELECT clause provides SQL programmers with the control they need to select rows for populating macro variables. This effective technique creates and populates a range of two or more macro variables using an INTO clause, the keyword THROUGH or hyphen (-), and a WHERE clause to control what rows and values are selected and populated as macro variables. Suppose that you want to create and populate macro variables for the first four ‘Software’ products in the PRODUCTS table. The INTO clause creates four macro variables and assigns values from the first four rows that match the WHERE clause expression. Any rows that do not match the WHERE clause expression are omitted from the results. The ORDER BY clause is specified to perform an ascending sort on the product name (PRODNAME), and macro resolution results are displayed in the SAS log.

SQL Code

```

PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODUCT1 - :PRODUCT4,
         :COST1 - :COST4
   FROM PRODUCTS
      WHERE PRODTYPE = 'Software'
      ORDER BY PRODNAME;
QUIT;
%PUT &PRODUCT1 &COST1;
%PUT &PRODUCT2 &COST2;
%PUT &PRODUCT3 &COST3;
%PUT &PRODUCT4 &COST4;

```

SAS Log Results

```

PROC SQL NOPRINT;
  SELECT PRODNAME,
         PRODCOST
    INTO :PRODUCT1 - :PRODUCT4,
         :COST1 - :COST4
   FROM PRODUCTS
  WHERE PRODTYPE = 'Software'
    ORDER BY PRODNAME;
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.06 seconds
      cpu time           0.00 seconds

%PUT &PRODUCT4 &COST4;
Database Software $399.00
%PUT &PRODUCT3 &COST3;
Graphics Software $299.00
%PUT &PRODUCT2 &COST2;
Spreadsheet Software $299.00
%PUT &PRODUCT1 &COST1;
Wordprocessor Software $299.00

```

Creating a List of Values in a Macro Variable

Concatenating values of a single column into one macro variable enables you to create a list of values that can be displayed in the SAS log or can be output to a SAS data set. Using the INTO clause with the SEPARATED BY keyword creates a list of values. For example, suppose that you want to create a blank-delimited list containing manufacturer names (MANUNAME) from the MANUFACTURERS table. Create a macro variable called &MANUNAME and assign the manufacturer names to a blank-delimited list with each name separated with two blank spaces. The WHERE clause restricts the list's contents to only manufacturers who are located in San Diego.

SQL Code

```

PROC SQL NOPRINT;
  SELECT MANUNAME
    INTO :MANUNAME SEPARATED BY ' '
     FROM MANUFACTURERS
    WHERE UPCASE(MANUCITY)=' SAN DIEGO' ;
QUIT;
%PUT &MANUNAME ;

```

SAS Log Results

```

PROC SQL NOPRINT;
  SELECT MANUNAME
    INTO :MANUNAME SEPARATED BY ' '
      FROM MANUFACTURERS
        WHERE UPCASE(MANUCITY)='SAN DIEGO';
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

%PUT &MANUNAME;
Global Comm Corp  Global Software  San Diego PC Planet

```

In the next example, a similar list that contains manufacturers from San Diego is created. But instead of separating each name with two blanks as in the previous example, a comma is used instead.

SQL Code

```

PROC SQL NOPRINT;
  SELECT MANUNAME
    INTO :MANUNAME SEPARATED BY ', '
      FROM MANUFACTURERS
        WHERE UPCASE(MANUCITY)=' SAN DIEGO';
  QUIT;
%PUT &MANUNAME;

```

SAS Log Results

```

PROC SQL NOPRINT;
  SELECT MANUNAME
    INTO :MANUNAME SEPARATED BY ', '
      FROM MANUFACTURERS
        WHERE UPCASE(MANUCITY)='SAN DIEGO';
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

%PUT &MANUNAME;
Global Comm Corp, Global Software, San Diego PC Planet

```

Using Automatic Macro Variables to Control Processing

Three automatic macro variables that are supplied by SAS are assigned values during SQL processing to provide process control information. SQL users can determine the number of rows processed with the SQLOBS macro variable, assess whether a PROC SQL statement was successful or not with the SQLRC macro variable, and identify the number of iterations the inner loop of an SQL query processes with the SQLOOPS macro variable. To inspect the values of all automatic macro variables at your installation, use the _AUTOMATIC_ option in a %PUT statement.

SQL Code

```
%PUT _AUTOMATIC_;
```

SAS Log Results

```
%PUT _AUTOMATIC_;  
AUTOMATIC AFDSID 0  
AUTOMATIC AFDSNAME  
AUTOMATIC AFLIB  
AUTOMATIC AFSTR1  
AUTOMATIC AFSTR2  
AUTOMATIC FSPBDV  
AUTOMATIC SYSBUFFR  
AUTOMATIC SYSCC 0  
AUTOMATIC SYSCHARWIDTH 1  
AUTOMATIC SYSCMD  
AUTOMATIC SYSDATE 10JUN04  
AUTOMATIC SYSDATE9 10JUN2004  
AUTOMATIC SYSDAY Thursday  
AUTOMATIC SYSDEVIC  
AUTOMATIC SYSDMG 0  
AUTOMATIC SYSDSN WORK INVENTORY  
AUTOMATIC SYSENDIAN LITTLE  
AUTOMATIC SYSENV FORE  
AUTOMATIC SYSERR 0  
AUTOMATIC SYSFILRC 0  
AUTOMATIC SYSINDEX 3  
AUTOMATIC SYSINFO 0  
AUTOMATIC SYSJOBID 3580  
AUTOMATIC SYSLAST WORK.INVENTORY  
AUTOMATIC SYSLCKRC 0  
AUTOMATIC SYSLIBRC 0  
AUTOMATIC SYSMACRONAME  
AUTOMATIC SYSMAXLONG 2147483647  
AUTOMATIC SYSMENV S  
AUTOMATIC SYMSG  
AUTOMATIC SYSNCPU 1  
AUTOMATIC SYSPARM  
AUTOMATIC SYSPBUFF  
AUTOMATIC SYSPROCESSID 41D4E6142950312740200000000000000  
AUTOMATIC SYSPROCESSNAME DMS Process  
AUTOMATIC SYSPROCNAME  
AUTOMATIC SYSRC 0  
AUTOMATIC SYSSCP WIN  
AUTOMATIC SYSSCPL XP_HOME  
AUTOMATIC SYSSITE 0045254001  
AUTOMATIC SYSSIZEOFLONG 4  
AUTOMATIC SYSSIZEOFUNICODE 2  
AUTOMATIC SYSSTARTID  
AUTOMATIC SYSSTARTNAME  
AUTOMATIC SYSTIME 12:50
```

```
AUTOMATIC SYSUSERID Valued Sony Customer
AUTOMATIC SYSVER 9.1
AUTOMATIC SYSVLONG 9.01.01M0P111803
AUTOMATIC SYSVLONG4 9.01.01M0P11182003
```

Building Macro Tools and Applications

The macro facility, combined with the capabilities of the SQL procedure, enables the creation of versatile macro tools and general purpose applications. A principle design goal when developing user-written macros should be that they are useful and simple to use. It is best to avoid using a macro that does not meet your needs or that has a name that is complicated and hard to remember.

As tools, macros should be designed to serve the needs of as many users as possible. They should contain no ambiguities, consist of distinctive macro variable names, avoid the possibility of naming conflicts between macro variables and data set variables, and not try to do too many things. This utilitarian approach to macro design helps gain widespread approval and acceptance by users.

Creating Simple Macro Tools

Macro tools can be constructed to perform a variety of useful tasks. The most effective macros are those that are simple and perform a common task. Before embarking on the construction of one or more macro tools, explore what processes are currently being performed, and then identify common users' needs with affected personnel by addressing voids. Once this has been accomplished, you will be in a better position to construct simple and useful macro tools that will be accepted by users.

Suppose that during an informal requirements analysis phase you identified users who, in the course of their jobs, use a variety of approaches and methods to create data set and variable cross-reference listings. To prevent unnecessary and wasteful duplication of effort, you decide to construct a simple macro tool that can be used by all users to retrieve information about the columns in one or more SAS data sets.

Cross-Referencing Columns

Column cross-reference listings are useful when you need to quickly identify all of the SAS library data sets that a column is defined in. Using the COLUMNS dictionary table (for more information, see Chapter 2, “Working with Data in PROC SQL”), a macro can be created that captures column-level information including column name, type, length, position, label, format, informat, indexes, as well as a cross-reference listing that contains the location of a column within a designated SAS library. In the next example, the COLUMNS macro consists of a PROC SQL query that accesses any single column in a SAS library. If the macro was invoked with a user-request consisting of

```
%COLUMNS (WORK,CUSTNUM) ;
```

then the macro would produce a cross-reference listing on the user library WORK for the column CUSTNUM in all DATA types.

SQL Code

```
%MACRO COLUMNS(LIB, COLNAME) ;
  PROC SQL;
    SELECT LIBNAME, MEMNAME, NAME, TYPE, LENGTH
    FROM DICTIONARY.COLUMNS
    WHERE LIBNAME = "&LIB" AND
      UPCASE(NAME) = "&COLNAME" AND
      MEMTYPE = "DATA";
  QUIT;
%MEND COLUMNS;
%COLUMNS (WORK,CUSTNUM) ;
```

It is worth noting that multiple matches could be found in databases that contains case-sensitive names. This would allow both “employee” and “EMPLOYEE” to be displayed as matches. This is not likely to occur too often in practice, but it is definitely a possibility.

Results

Library Name	Member Name	Column Name	Column Type	Column Length
WORK	CUSTOMERS	custnum	num	3
WORK	CUSTOMERS2	custnum	num	3
WORK	CUSTOMERS_BACKUP	custnum	num	8
WORK	INVOICE	custnum	num	3
WORK	PURCHASES	custnum	num	4
WORK	PURCHASES2	Custnum	num	8

Determining the Number of Rows in a Table

Sometimes it is useful to know the number of observations (or rows) in a table without first having to read all of the rows. Although the number of rows in a table is available for true SAS tables, the number of rows is not available for DBMS tables that use a LIBNAME engine. In the next example, the TABLES dictionary table is accessed in a user-defined macro called NOBS (for more information, see Chapter 2). Macro NOBS is designed to accept and process two user-supplied values: the library reference and the table name. Once these values are supplied, the results are displayed in the Output window.

SQL Code

```
%MACRO NOBS(LIB, TABLE);
PROC SQL;
  SELECT LIBNAME, MEMNAME, NOBS
  FROM DICTIONARY.TABLES
  WHERE UPCASE(LIBNAME) = "&LIB" AND
    UPCASE(MEMNAME) = "&TABLE" AND
    UPCASE(MEMTYPE) = "DATA";
QUIT;
%MEND NOBS;

%NOBS(WORK, PRODUCTS);
```

Results

Library Name	Member Name	Number of Physical Observations
WORK	PRODUCTS	10

Identifying Duplicate Rows in a Table

Sometimes it is useful to be able to identify duplicate rows in a table. In the next example, the SELECT statement with a COUNT summary function and HAVING clause are used in a user-defined macro called DUPS. Macro DUPS is designed to accept and process three user-supplied values: the library reference, table name, and column(s) in a group by list. Once these values are supplied by submitting macro DUPS, the macro is executed with the results displayed in the Output window.

SQL Code

```
%MACRO DUPS(LIB, TABLE, GROUPBY);
PROC SQL;
  SELECT &GROUPBY, COUNT(*) AS Duplicate_Rows
  FROM &LIB..&TABLE
  GROUP BY &GROUPBY
  HAVING COUNT(*) > 1;
QUIT;
%MEND DUPS;

%DUPS(WORK, PRODUCTS, PRODTYPE);
```

Results

Product Type	Duplicate_Rows
Phone	3
Software	4
Workstation	2

Summary

1. Conditional logic with the use of predicates (e.g., IN, BETWEEN, and CONTAINS) provides WHERE and HAVING clauses with added value and flexibility (see “Conditional Logic with Predicates (Operators)” section).
2. A CASE expression is a construct in PROC SQL that is used to evaluate whether a particular condition has been met (see the “CASE Expressions” section).
3. A CASE expression can be used to conditionally process a table’s rows (see the “CASE Expressions” section).
4. A single value is returned from its evaluation of each row in a table (see the “CASE Expressions” section).
5. Logic conditions can be combined using the AND and OR logical operators (see the “Logic and Nulls” section).
6. A missing or NULL value is returned when an ELSE expression is not specified and each WHEN condition is “false” (see the “Logic and Nulls” section).
7. A missing value is not the same as a value of 0 (zero), or as a blank character because it represents a unique value or a lack of a value (see the “Logic and Nulls” section).
8. The IFC and IFN functions are useful constructs for encoding, decoding, and flagging values in the construction of conditional logic in a SAS SQL query environment (see the “IFC and IFN Functions” section).
9. Although not part of the SQL ANSI guidelines, the IFC and IFN functions cannot be used with external database connections such as SAS to Oracle, SAS to DB2, SAS to SQL Server, etc. (see the “IFC and IFN Functions” section).
10. PROC SQL can be used with the SAS macro facility to perform common and repetitive tasks (see the “Interfacing PROC SQL with the Macro Language” section).
11. Simple, but effective, user-defined macros combined with the SQL procedure can be created for all users to use (see the “Building Macro Tools and Applications” section).
12. Single-value macro variables can be defined using the INTO clause (see the “Creating a Macro Variable with Aggregate Functions” section).
13. Value-list macro variables can be defined using the INTO clause and SEPARATED BY keyword (see “Controlling the Selection and Population of Macro Variables with a WHERE Clause” section).

Chapter 5: Creating, Populating, and Deleting Tables

Introduction	153
Creating Tables	154
Creating a Table Using Column-Definition Lists	154
Creating a Table Using the LIKE Clause.....	158
Deriving a Table and Data from an Existing Table.....	159
Populating Tables	160
Adding Data to a Table with a SET Clause	161
Adding Data to All of the Columns in a Row	164
Adding Data to Some of the Columns in a Row.....	169
Adding Data with a SELECT Query	171
Bulk Loading Data from Microsoft Excel.....	172
Integrity Constraints	177
Defining Integrity Constraints.....	178
Types of Integrity Constraints	178
Preventing Null Values with a NOT NULL Constraint	178
Enforcing Unique Values with a UNIQUE Constraint.....	181
Validating Column Values with a CHECK Constraint	182
Referential Integrity Constraints	183
Establishing a Primary Key	184
Establishing a Foreign Key	185
Displaying Integrity Constraints	188
Deleting Rows in a Table	189
Deleting a Single Row in a Table.....	189
Deleting More Than One Row in a Table	189
Deleting All Rows in a Table	190
Deleting Tables.....	190
Deleting a Single Table.....	191
Deleting Multiple Tables.....	191
Deleting Tables That Contain Integrity Constraints.....	191
Summary	193

Introduction

Previous chapters provided tables in the examples that had already been created and populated with data. But what if you need to create a table, populate it with data, or delete rows of data or tables that are no longer needed or wanted?

In this chapter, the discussions and examples focus on the way tables are created, populated, and deleted. These are important operations and essential elements in PROC SQL, especially if you want to increase your comprehension of SQL processes and improve your understanding of this powerful language.

Creating Tables

An important element in table creation is table design. Table design incorporates how tables are structured—how rows and columns are defined, how indexes are created, and how columns refer to values in other columns. If you seek a greater understanding in this area, then review the many references identified at the end of this book. The following overview should be kept in mind during the table design process.

When building a table it is important to devote adequate time to planning its design as well as understanding the needs that each table is meant to satisfy. This process involves a number of activities such as requirements and feasibility analysis, including cost/benefit of the proposed tables, the development of a logical description of the data sources, and physical implementation of the logical data model. Once these tasks are complete, you can assess any special business requirements that each table needs to provide. A business assessment helps by minimizing the number of changes required to a table once it has been created.

Next, determine what tables will be incorporated into your application's database. This requires understanding the value that each table is expected to provide. It also prevents a table of little or no importance from being incorporated into a database. The final step and one of critical importance is to define each table's columns, attributes, and contents.

Once the table design process is complete, each table is then ready to be created with the CREATE TABLE statement. The purpose of creating a table is to create an object that does not already exist. In the SAS implementation, three variations of the CREATE TABLE statement can be specified depending on your needs:

- Creating a table using column-definition lists
- Creating a table using the LIKE clause
- Deriving a table and data from an existing table

Creating a Table Using Column-Definition Lists

Although part of the SQL standard, the column-definition list (like the LENGTH statement in the DATA step) is a laborious and not very elegant way to create a table. The disadvantage of creating a table this way is that it requires the definition of each column's attributes including their data type, length, informat, and format. This method is frequently used to create columns when they are not present in another table. Using this method results in the creation of an empty table (*without rows*). The code used to create the CUSTOMERS table appears below. This example illustrates the creation of a table with column-definition lists.

SQL Code

```
PROC SQL;
  CREATE TABLE CUSTOMERS
    (CUSTNUM   NUM      LABEL='Customer Number',
     CUSTNAME  CHAR(25) LABEL='Customer Name',
     CUSTCITY  CHAR(20) LABEL='Customer''s Home City');
QUIT;
```

SAS Log Results

```

PROC SQL;
  CREATE TABLE CUSTOMERS
    (CUSTNUM      NUM      LABEL='Customer Number',
     CUSTNAME     CHAR(25)  LABEL='Customer Name',
     CUSTCITY     CHAR(20)  LABEL='Customer''s Home City');
NOTE: Table CUSTOMERS created, with 0 rows and 3 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.81 seconds

```

You should be aware that the SQL procedure ignores width specifications for numeric columns. When a numeric column is defined, it is created with a width of 8 bytes, which is the maximum precision allowed by SAS. PROC SQL ignores numeric length specifications when the value is less than 8 bytes. To illustrate this point, a partial CONTENTS procedure output is displayed for the CUSTOMERS table.

Results

The CONTENTS Procedure

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	CUSTCITY	Char	20	Customer's Home City
2	CUSTNAME	Char	25	Customer Name
1	CUSTNUM	Num	8	Customer Number

To conserve storage space (CUSTNUM only requires maximum precision provided in 3 bytes), a LENGTH statement could be used in a DATA step to define CUSTNUM as a 3-byte column rather than an 8-byte column.

DATA Step Code

```

DATA CUSTOMERS;
  LENGTH CUSTNUM 3.;
  SET CUSTOMERS;
  LABEL CUSTNUM = 'Customer Number';
RUN;

```

Results

The CONTENTS Procedure

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	CUSTCITY	Char	20	Customer's Home City
2	CUSTNAME	Char	25	Customer Name
1	CUSTNUM	Num	3	Customer Number

Let's look at the column-definition list that is used to create the PRODUCTS table.

SQL Code

```
PROC SQL;
  CREATE TABLE PRODUCTS
    (PRODNUM  NUM(3)    LABEL='Product Number',
     PRODNAME CHAR(25)  LABEL='Product Name',
     MANUNUM  NUM(3)    LABEL='Manufacturer Number',
     PRODTYPE CHAR(15)  LABEL='Product Type',
     PRODCOST  NUM(5,2)  FORMAT=DOLLAR9.2 LABEL='Product Cost');
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE PRODUCTS
    (PRODNUM  NUM(3)    LABEL='Product Number',
     PRODNAME CHAR(25)  LABEL='Product Name',
     MANUNUM  NUM(3)    LABEL='Manufacturer Number',
     PRODTYPE CHAR(15)  LABEL='Product Type',
     PRODCOST  NUM(5,2)  FORMAT=DOLLAR9.2 LABEL='Product
Cost');
NOTE: Table PRODUCTS created, with 0 rows and 5 columns.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

The CONTENTS output for the PRODUCTS table shows once again that the SQL procedure ignores all width specifications for numeric columns.

Results

The CONTENTS Procedure

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
3	MANUNUM	Num	8		Manufacturer Number
5	PRODCOST	Num	8	DOLLAR9.2	Product Cost
2	PRODNAME	Char	25		Product Name
1	PRODNUM	Num	8		Product Number
4	PRODTYPE	Char	15		Product Type

As before, to conserve storage space you can use a LENGTH statement in a DATA step to override the default 8-byte column definition for numeric columns.

DATA Step Code

```
DATA PRODUCTS;
  LENGTH PRODNUM MANUNUM 3.
    PRODCOST 5.;
  SET PRODUCTS (DROP=PRODNUM MANUNUM PRODCOST);
  LABEL PRODNUM = 'Product Number'
    MANUNUM = 'Manufacturer Number'
    PRODCOST = 'Product Cost';
  FORMAT PRODCOST DOLLAR9.2;
RUN;
```

Results

The CONTENTS Procedure

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
2	MANUNUM	Num	3		Manufacturer Number
3	PRODCOST	Num	5	DOLLAR9.2	Product Cost
4	PRODNAME	Char	25		Product Name
1	PRODNUM	Num	3		Product Number
5	PRODTYPE	Char	15		Product Type

Creating a Table Using the LIKE Clause

Referencing an existing table in a CREATE TABLE statement is an effective way of creating a new table. In fact, it can be a great time-saver, because it prevents having to define each column one at a time as was shown with column-definition lists. The LIKE clause (in the CREATE TABLE statement) triggers the existing table's structure to be copied to the new table minus any columns dropped with the KEEP= or DROP= data set (table) option. It copies the column names and attributes from the existing table structure to the new table structure. Using this method results in the creation of an empty table (*without rows*). To illustrate this method of creating a new table, a table called HOT_PRODUCTS will be created with the LIKE clause.

SQL Code

```
PROC SQL;
  CREATE TABLE HOT_PRODUCTS
    LIKE PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE HOT_PRODUCTS
    LIKE PRODUCTS;
NOTE: Table HOT_PRODUCTS created, with 0 rows and 5 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

As a result of executing the CREATE TABLE statement and the LIKE clause the new table contains zero rows of data but does include all the column definitions without any rows of data from an existing table and/or view. What this means is that only the metadata (data describing data) is copied to the new table. The particular definitions that are copied include: column names, data type, length, precision, label, informat, format, and so on.

The next example illustrates how to create a new table by selecting only the columns that you have an interest in. This method is not supported by the SQL ANSI standard. Suppose that you want three columns (PRODNAME, PRODTYPE, and PRODCOST) from the PRODUCTS table. The following code illustrates how the KEEP= data set (table) option can be used to accomplish this.

Note: Data sets can also be called tables.

SQL Code

```
PROC SQL;
  CREATE TABLE HOT_PRODUCTS (KEEP=PRODNAME PRODTYPE PRODCOST)
    LIKE PRODUCTS;
QUIT;
```

SAS Log Results

```

PROC SQL;
  CREATE TABLE HOT_PRODUCTS (KEEP=PRODNAME PRODTYPE
PRODCOST)
    LIKE PRODUCTS;
NOTE: Table HOT_PRODUCTS created, with 0 rows and 3 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

```

Deriving a Table and Data from an Existing Table

Deriving a new table from an existing table is the most popular and effective way to create a table. This method uses a query expression, and the results are stored in a new table instead of being displayed as SAS output. This method not only stores the column names and their attributes, but also stores the rows of data that satisfy the query expression. The next example illustrates creating a new table with a query expression.

SQL Code

```

PROC SQL;
  CREATE TABLE HOT_PRODUCTS AS
    SELECT *
      FROM PRODUCTS;
QUIT;

```

SAS Log Results

```

PROC SQL;
  CREATE TABLE HOT_PRODUCTS AS
    SELECT *
      FROM PRODUCTS;
NOTE: Table WORK.HOT_PRODUCTS created, with 10 rows and 5
columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

```

You might notice after examining the SAS log in the previous example that the SELECT statement extracted all rows from the existing table (PRODUCTS) and copied them to the new table (HOT_PRODUCTS). In the absence of a WHERE clause, the resulting table (HOT_PRODUCTS) contains the identical number of rows as the parent table PRODUCTS.

The power of the CREATE TABLE statement, then, is in its ability to create a new table from an existing table. What is often overlooked in this definition is the CREATE TABLE statement's ability to form a subset of a parent table. More frequently than not, a new table represents a subset of its parent table. For this reason, this method of creating a table is the most powerful and widely used. Suppose that you want to create a table called HOT_PRODUCTS that contains a subset of the “Software” and “Phones” product types. The following query-expression would accomplish this.

SQL Code

```
PROC SQL;
  CREATE TABLE HOT_PRODUCTS AS
    SELECT *
      FROM PRODUCTS
        WHERE UPCASE(PRODTYPE) IN ("SOFTWARE", "PHONE");
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE HOT_PRODUCTS AS
    SELECT *
      FROM PRODUCTS
        WHERE UPCASE(PRODTYPE) IN ("SOFTWARE", "PHONE");
NOTE: Table WORK.HOT_PRODUCTS created, with 7 rows and 5
columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.38 seconds
```

Let's look at another example. Suppose that you want to create another table called NOT_SO_HOT_PRODUCTS that contains a subset of everything but the "Software" and "Phones" product types. The following query-expression would accomplish this.

SQL Code

```
PROC SQL;
  CREATE TABLE NOT_SO_HOT_PRODUCTS AS
    SELECT *
      FROM PRODUCTS
        WHERE UPCASE(PRODTYPE) NOT IN ("SOFTWARE", "PHONE");
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE NOT_SO_HOT_PRODUCTS AS
    SELECT *
      FROM sql.PRODUCTS
        WHERE UPCASE(PRODTYPE) NOT IN ("SOFTWARE",
"PHONE");
NOTE: Table NOT_SO_HOT_PRODUCTS created, with 3 rows and 5
columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          1.20 seconds
```

Populating Tables

After a table is created, it can then be populated with data. Unless the newly created table is defined as a subset of an existing table or its content is to remain static, one or more rows of

data may eventually need to be added. The SQL standard provides the INSERT INTO statement as the vehicle for adding rows of data. In fact, the INSERT INTO statement doesn't really insert rows of data at all. It simply adds each row to the end of the table.

The examples in this section look at a number of approaches to populate tables. As new rows of data are added to or changed in a target table, many things must be kept in-check behind the scenes including the need to carry out or commit any permanent changes to a table. This important step checks each write operation for errors and, should one or more errors occur, provides SAS with the ability to completely rollback or undo the changes since the last commit. For example, before a row insertion is allowed entry into a table with assigned integrity constraints, it is first marked as uncommitted, then validated against the assigned integrity constraints, and if no violations occur, the row insertion is committed to the table. The following row insertion methods are illustrated:

- adding data to a table with a SET clause
- adding data to all of the columns in a row
- adding data to some of the columns in a row
- adding data with a SELECT query
- bulk loading data with Microsoft Excel

Adding Data to a Table with a SET Clause

You populate tables with data by using an INSERT INTO statement. Three parameters are specified with an INSERT INTO statement: the name of the table, the names of the columns in which values are inserted, and the values themselves. One or more rows of data are inserted into a table with a SET clause. Suppose that you want to insert (or add) a single row of data to the CUSTOMERS table and the row consists of three columns (Customer Number, Customer Name, and Home City).

SQL Code

```
PROC SQL;
  INSERT INTO CUSTOMERS
    SET CUSTNUM=702,
        CUSTNAME='Mission Valley Computing',
        CUSTCITY='San Diego';
QUIT;
```

The SAS log displays the following message noting that one row was inserted into the CUSTOMERS table.

SAS Log Results

```

PROC SQL;
  INSERT INTO CUSTOMERS
    SET CUSTNUM=702,
        CUSTNAME='Mission Valley Computing',
        CUSTCITY='San Diego';
  NOTE: 1 row was inserted into WORK.CUSTOMERS.
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.08 seconds

```

The inserted row of data from the INSERT INTO statement and SET clause is added to the end of the CUSTOMERS table.

	Customer Number	Customer Name	Customer's Home City
1	101	La Mesa Computer Land	La Mesa
2	201	Vista Tech Center	Vista
3	301	Coronado Internet Zone	Coronado
4	401	La Jolla Computing	La Jolla
5	501	Alpine Technical Center	Alpine
6	601	Oceanside Computer Land	Oceanside
7	701	San Diego Byte Store	San Diego
8	801	Jamul Hardware & Software	Jamul
9	901	Del Mar Tech Center	Del Mar
10	1001	Lakeside Software Center	Lakeside
11	1101	Bonsall Network Store	Bonsall
12	1201	Rancho Santa Fe Tech	Rancho Santa Fe
13	1301	Spring Valley Byte Center	Spring Valley
14	1401	Poway Central	Poway
15	1501	Valley Center Tech Center	Valley Center
16	1601	Fairbanks Tech USA	Fairbanks Ranch
17	1701	Blossom Valley Tech	Blossom Valley
18	1801	Chula Vista Networks	
19	702	Mission Valley Computing	San Diego

Entering a new row into a table containing an index will automatically add the value to the index (for more information about indexes, see Chapter 6, “Modifying and Updating Tables and Indexes”). The following example illustrates adding three rows of data using the SET clause.

SQL Code

```

PROC SQL;
  INSERT INTO CUSTOMERS
    SET CUSTNUM=402,
        CUSTNAME='La Jolla Tech Center',
        CUSTCITY='La Jolla'
  SET CUSTNUM=502,
        CUSTNAME='Alpine Byte Center',
        CUSTCITY='Alpine'

```

```
SET CUSTNUM=1702,
   CUSTNAME='Rancho San Diego Tech',
   CUSTCITY='Rancho San Diego';
QUIT;
```

The SAS log shows that three rows of data were inserted into the CUSTOMERS table.

SAS Log Results

```
PROC SQL;
  INSERT INTO CUSTOMERS
    SET CUSTNUM=402,
        CUSTNAME='La Jolla Tech Center',
        CUSTCITY='La Jolla'
    SET CUSTNUM=502,
        CUSTNAME='Alpine Byte Center',
        CUSTCITY='Alpine'
    SET CUSTNUM=1701,
        CUSTNAME='Rancho San Diego Tech',
        CUSTCITY='Rancho San Diego';
  NOTE: 3 rows were inserted into WORK.CUSTOMERS.
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.01 seconds
```

The inserted rows of data from the INSERT INTO statement and SET clause are added to the end of the CUSTOMERS table.

	Customer Number	Customer Name	Customer's Home City
1	101	La Mesa Computer Land	La Mesa
2	201	Vista Tech Center	Vista
3	301	Coronado Internet Zone	Coronado
4	401	La Jolla Computing	La Jolla
5	501	Alpine Technical Center	Alpine
6	601	Oceanside Computer Land	Oceanside
7	701	San Diego Byte Store	San Diego
8	801	Jamul Hardware & Software	Jamul
9	901	Del Mar Tech Center	Del Mar
10	1001	Lakeside Software Center	Lakeside
11	1101	Bonsall Network Store	Bonsall
12	1201	Rancho Santa Fe Tech	Rancho Santa Fe
13	1301	Spring Valley Byte Center	Spring Valley
14	1401	Poway Central	Poway
15	1501	Valley Center Tech Center	Valley Center
16	1601	Fairbanks Tech USA	Fairbanks Ranch
17	1701	Blossom Valley Tech	Blossom Valley
18	1801	Chula Vista Networks	
19	702	Mission Valley Computing	San Diego
20	402	La Jolla Tech Center	La Jolla
21	502	Alpine Byte Center	Alpine
22	1702	Rancho San Diego Tech	Rancho San Diego

Adding Data to All of the Columns in a Row

Another way to populate a table with data uses a VALUES clause with an INSERT INTO statement. As described in the previous section, three parameters are specified with an INSERT INTO statement and a VALUES clause: the name of the table, the names of the columns in which values are inserted, and the values themselves. Data values are inserted into a table with a VALUES clause. Suppose that you want to insert (or add) a single row of data to the CUSTOMERS table, and the row consists of three columns (Customer Number, Customer Name, and Home City).

SQL Code

```
PROC SQL;
  INSERT INTO CUSTOMERS (CUSTNUM, CUSTNAME, CUSTCITY)
    VALUES (703, 'Sonic Boom Analytics', 'Spring Valley');
QUIT;
```

The SAS log displays the following message noting that one row was inserted into the CUSTOMERS table.

SAS Log Results

```
PROC SQL;
  INSERT INTO CUSTOMERS
    (CUSTNUM, CUSTNAME, CUSTCITY)
    VALUES (703, 'Sonic Boom Analytics', 'Spring Valley');
NOTE: 1 row was inserted into WORK.CUSTOMERS.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.54 seconds
```

The inserted row of data from the previous INSERT INTO statement is added to the end of the CUSTOMERS table.

	Customer Number	Customer Name	Customer's Home City
1	101	La Mesa Computer Land	La Mesa
2	201	Vista Tech Center	Vista
3	301	Coronado Internet Zone	Coronado
4	401	La Jolla Computing	La Jolla
5	501	Alpine Technical Center	Alpine
6	601	Oceanside Computer Land	Oceanside
7	701	San Diego Byte Store	San Diego
8	801	Jamul Hardware & Software	Jamul
9	901	Del Mar Tech Center	Del Mar
10	1001	Lakeside Software Center	Lakeside
11	1101	Bonsall Network Store	Bonsall
12	1201	Rancho Santa Fe Tech	Rancho Santa Fe
13	1301	Spring Valley Byte Center	Spring Valley
14	1401	Poway Central	Poway
15	1501	Valley Center Tech Center	Valley Center
16	1601	Fairbanks Tech USA	Fairbanks Ranch
17	1701	Blossom Valley Tech	Blossom Valley
18	1801	Chula Vista Networks	
19	702	Mission Valley Computing	San Diego
20	402	La Jolla Tech Center	La Jolla
21	502	Alpine Byte Center	Alpine
22	1702	Rancho San Diego Tech	Rancho San Diego
23	703	Sonic Boom Analytics	Spring Valley

Entering a new row into a table that contains an index will automatically add the value to the index (for more information about indexes, see Chapter 6, “Modifying and Updating Tables and Indexes”). The `INSERT INTO` statement can also add multiple rows of data to a table. The following example illustrates adding three rows of data using the `VALUES` clause.

SQL Code

```

PROC SQL;
  INSERT INTO CUSTOMERS
    (CUSTNUM, CUSTNAME, CUSTCITY)
    VALUES (402, 'La Jolla Tech Center', 'La Jolla')
    VALUES (502, 'Alpine Byte Center', 'Alpine')
    VALUES (1702, 'Rancho San Diego Tech', 'Rancho San Diego');

  SELECT *
    FROM CUSTOMERS
   ORDER BY CUSTNUM;
QUIT;

```

The SAS log shows three rows of data were inserted into the CUSTOMERS table.

SAS Log Results

```
PROC SQL;
  INSERT INTO CUSTOMERS
    (CUSTNUM, CUSTNAME, CUSTCITY)
    VALUES (402, 'La Jolla Tech Center', 'La Jolla')
    VALUES (502, 'Alpine Byte Center', 'Alpine')
    VALUES (1701, 'Rancho San Diego Tech', 'Rancho San
Diego');
  NOTE: 3 rows were inserted into WORK.CUSTOMERS.

  SELECT *
    FROM CUSTOMERS
    ORDER BY CUSTNUM;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          1.03 seconds
```

The new rows are displayed in ascending order by CUSTNUM.

Customer Number	Customer Name	Customer's Home City
101	La Mesa Computer Land	La Mesa
201	Vista Tech Center	Vista
301	Coronado Internet Zone	Coronado
401	La Jolla Computing	La Jolla
402	La Jolla Tech Center	La Jolla
402	La Jolla Tech Center	La Jolla
501	Alpine Technical Center	Alpine
502	Alpine Byte Center	Alpine
502	Alpine Byte Center	Alpine
601	Oceanside Computer Land	Oceanside
701	San Diego Byte Store	San Diego
702	Mission Valley Computing	San Diego
703	Sonic Boom Analytics	Spring Valley
801	Jamul Hardware & Software	Jamul
901	Del Mar Tech Center	Del Mar
1001	Lakeside Software Center	Lakeside
1101	Bonsall Network Store	Bonsall
1201	Rancho Santa Fe Tech	Rancho Santa Fe
1301	Spring Valley Byte Center	Spring Valley
1401	Poway Central	Poway
1501	Valley Center Tech Center	Valley Center
1601	Fairbanks Tech USA	Fairbanks Ranch
1701	Blossom Valley Tech	Blossom Valley
1702	Rancho San Diego Tech	Rancho San Diego
1702	Rancho San Diego Tech	Rancho San Diego
1801	Chula Vista Networks	

The INSERT INTO statement can also be used to insert rows of data without having to specify the column names as long as a value is keyed for each and every column in the table in the exact order that columns are specified in the table. **Note:** Learning the order of a table's columns may require producing output from the CONTENTS procedure, and/or inspection of the metadata column, VARNUM, from the Dictionary.columns table or SASHELP.VCOLUMNS view. The following example illustrates the absence of the table's column names in the INSERT INTO statement as three rows of data are added using the VALUES clause.

SQL Code

```
PROC SQL;
  INSERT INTO CUSTOMERS
    VALUES (402, 'La Jolla Tech Center', 'La Jolla')
    VALUES (502, 'Alpine Byte Center', 'Alpine')
    VALUES (1702, 'Rancho San Diego Tech', 'Rancho San Diego');
```

```

SELECT *
  FROM CUSTOMERS
 ORDER BY CUSTNUM;
QUIT;

```

The SAS log shows three rows of data were inserted into the CUSTOMERS table.

SAS Log Results

```

PROC SQL;
  INSERT INTO CUSTOMERS
    VALUES (402, 'La Jolla Tech Center', 'La Jolla')
    VALUES (502, 'Alpine Byte Center', 'Alpine')
    VALUES (1701,'Rancho San Diego Tech','Rancho San
Diego');
NOTE: 3 rows were inserted into WORK.CUSTOMERS.

      SELECT *
        FROM CUSTOMERS
       ORDER BY CUSTNUM;
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.04 seconds

```

The new rows are displayed in ascending order by CUSTNUM.

Customer Number	Customer Name	Customer's Home City
101	La Mesa Computer Land	La Mesa
201	Vista Tech Center	Vista
301	Coronado Internet Zone	Coronado
401	La Jolla Computing	La Jolla
402	La Jolla Tech Center	La Jolla
501	Alpine Technical Center	Alpine
502	Alpine Byte Center	Alpine
601	Oceanside Computer Land	Oceanside
701	San Diego Byte Store	San Diego
702	Mission Valley Computing	San Diego
801	Jamul Hardware & Software	Jamul
901	Del Mar Tech Center	Del Mar
1001	Lakeside Software Center	Lakeside
1101	Bonsall Network Store	Bonsall
1201	Rancho Santa Fe Tech	Rancho Santa Fe
1301	Spring Valley Byte Center	Spring Valley
1401	Poway Central	Poway
1501	Valley Center Tech Center	Valley Center
1601	Fairbanks Tech USA	Fairbanks Ranch
1701	Blossom Valley Tech	Blossom Valley
1702	Rancho San Diego Tech	Rancho San Diego
1801	Chula Vista Networks	

Adding Data to Some of the Columns in a Row

It is not uncommon when adding rows of data to a table to have one or more columns with an unassigned value. When this happens, SQL must be able to handle adding the rows to the table as if all of the values were present. But how does SQL handle values that are not specified? You will see in the following example that SQL assigns missing values to columns that do not have a value specified. As before, three parameters are specified with the INSERT INTO statement: the name of the table, the names of the columns in which values are inserted, and the values themselves. Suppose that you had to add two rows of incomplete data to the CUSTOMERS table, where two of three columns were specified (Customer Number and Customer Name).

SQL Code

```
PROC SQL;
  INSERT INTO CUSTOMERS
    (CUSTNUM, CUSTNAME)
    VALUES (102, 'La Mesa Byte & Floppy')
    VALUES (902, 'Del Mar Technology Center');
  SELECT *
    FROM CUSTOMERS
    ORDER BY CUSTNUM;
QUIT;
```

The SAS log shows that two rows of data were added to the CUSTOMERS table.

SAS Log Results

```
PROC SQL;
  INSERT INTO CUSTOMERS
    (CUSTNUM, CUSTNAME)
    VALUES (102, 'La Mesa Byte & Floppy')
    VALUES (902, 'Del Mar Technology Center');
NOTE: 2 rows were inserted into WORK.CUSTOMERS.
  SELECT *
    FROM CUSTOMERS
    ORDER BY CUSTNUM;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

The new rows are displayed in ascending order by CUSTNUM with missing values assigned to the character column CUSTCITY.

Customer Number	Customer Name	Customer's Home City
101	La Mesa Computer Land	La Mesa
102	La Mesa Byte & Floppy	
201	Vista Tech Center	Vista
301	Coronado Internet Zone	Coronado
401	La Jolla Computing	La Jolla
402	La Jolla Tech Center	La Jolla
402	La Jolla Tech Center	La Jolla
501	Alpine Technical Center	Alpine
502	Alpine Byte Center	Alpine
502	Alpine Byte Center	Alpine
601	Oceanside Computer Land	Oceanside
701	San Diego Byte Store	San Diego
702	Mission Valley Computing	San Diego
703	Sonic Boom Analytics	Spring Valley
801	Jamul Hardware & Software	Jamul
901	Del Mar Tech Center	Del Mar
902	Del Mar Technology Center	
1001	Lakeside Software Center	Lakeside
1101	Bonsall Network Store	Bonsall
1201	Rancho Santa Fe Tech	Rancho Santa Fe
1301	Spring Valley Byte Center	Spring Valley
1401	Poway Central	Poway
1501	Valley Center Tech Center	Valley Center
1601	Fairbanks Tech USA	Fairbanks Ranch
1701	Blossom Valley Tech	Blossom Valley
1702	Rancho San Diego Tech	Rancho San Diego
1702	Rancho San Diego Tech	Rancho San Diego
1801	Chula Vista Networks	

In the previous example, missing values were assigned to the character column CUSTCITY. Suppose that you want to add two rows of partial data to the PRODUCTS table, where four of the five columns are specified (Product Number, Product Name, Product Type, and Product Cost), and the missing value for each row is the numeric column MANUNUM.

SQL Code

```

PROC SQL;
  INSERT INTO PRODUCTS
    (PRODNUM, PRODNAME, PRODTYPE, PRODCOST)
    VALUES(6002,'Security Software','Software',375.00)
    VALUES(1701,'Travel Laptop SE', 'Laptop', 4200.00);

  SELECT *
    FROM PRODUCTS
    ORDER BY PRODNUM;
QUIT;

```

The SAS log shows that two rows of data were added to the PRODUCTS table.

SAS Log Results

```

PROC SQL;
  INSERT INTO PRODUCTS
    (PRODNUM, PRODNAME, PRODTYPE, PRODCOST)
    VALUES (6002,'Security Software','Software',375.00)
    VALUES (1701,'Travel Laptop SE', 'Laptop', 4200.00);
NOTE: 2 rows were inserted into WORK.PRODUCTS.

  SELECT *
    FROM PRODUCTS
    ORDER BY PRODNUM;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.75 seconds

```

The new rows are displayed in ascending order by PRODNUM with missing values assigned to the numeric column MANUNUM.

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
1700	Travel Laptop	170	Laptop	\$3,400.00
1701	Travel Laptop SE	.	Laptop	\$4,200.00
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00
5001	Spreadsheet Software	500	Software	\$299.00
5002	Database Software	500	Software	\$399.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00
6002	Security Software	.	Software	\$375.00

Adding Data with a SELECT Query

You can also add data to a table using a SELECT query with an INSERT INTO statement. A query expression essentially executes an enclosed query by first creating a temporary table and then inserting the contents of the temporary table into the target table being populated. In the process of populating the target table, any columns omitted from the column list are automatically assigned to missing values.

In the next example, a SELECT query is used to add a row of data from the PRODUCTS table into the SOFTWARE_PRODUCTS table. The designated query controls the insertion of data into the target SOFTWARE_PRODUCTS table by using a WHERE clause.

SQL Code

```
PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS
    (PRODNUM, PRODNAME, PRODTYPE, PRODCOST)
    SELECT PRODNUM, PRODNAME, PRODTYPE, PRODCOST
      FROM PRODUCTS
     WHERE PRODTYPE IN ('Software') AND
          PRODCOST > 300;
QUIT;
```

The SAS log shows that one row of data was added to the SOFTWARE_PRODUCTS table.

SAS Log Results

```
PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS
    (PRODNUM, PRODNAME, PRODTYPE, PRODCOST)
    SELECT PRODNUM, PRODNAME, PRODTYPE, PRODCOST
      FROM PRODUCTS
     WHERE PRODTYPE IN ('Software') AND
          PRODCOST > 300;
NOTE: 1 row was inserted into WORK.SOFTWARE_PRODUCTS.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.07 seconds
      cpu time          0.01 seconds
```

The inserted row of data from the previous INSERT INTO statement is added to the end of the SOFTWARE_PRODUCTS table.

	Product Number	Product Name	Product Type	Product Cost
1	5001	Spreadsheet Software	Software	\$299.00
2	5002	Database Software	Software	\$399.00
3	5003	Wordprocessor Software	Software	\$299.00
4	5004	Graphics Software	Software	\$299.00
5	5002	Database Software	Software	\$399.00

Bulk Loading Data from Microsoft Excel

Although no specific SQL procedure statement or clause is available for bulk loading large amounts of data into a table, techniques for processing bulk load operations include using the IMPORT procedure, comma separated values (CSV) files, or LIBNAME statement with the Excel engine to populate tables. During the process of populating the target table, any columns omitted from the bulk load are automatically assigned to missing values.

In the next example, the IMPORT procedure is used to access and read data from the Excel file, PURCHASES.xls, and to create the SAS table, PURCHASES. The REPLACE option is specified to replace the PURCHASES table if it already exists. The Excel spreadsheet file appears below.

Table 5.1: PURCHASES Excel File

	A	B	C	D
1	custnum	prodnum	units	unitcost
2	1701	1110	1	\$3,200.00
3	101	5001	7	\$299.00
4	701	5001	11	\$299.00
5	701	5003	8	\$299.00
6	701	5002	4	\$399.00
7	701	5004	3	\$299.00
8	701	1700	2	\$3,400.00
9	701	1200	3	\$3,300.00
10	701	1110	2	\$3,200.00
11	1301	5001	3	\$299.00
12	1301	5003	5	\$299.00
13	1301	5002	2	\$399.00
14	901	1700	2	\$3,400.00
15	901	1200	3	\$3,300.00
16	901	1110	5	\$3,200.00
17	901	5001	9	\$299.00
18	901	5002	5	\$399.00
19	901	5003	8	\$299.00
20	901	5004	2	\$299.00
21	401	5001	11	\$299.00
22	401	5002	5	\$399.00
23	401	5003	7	\$299.00
24	401	5004	3	\$299.00
25	401	1700	3	\$3,400.00
26	401	1200	6	\$3,300.00
27	201	5001	6	\$299.00
28	201	5001	6	\$299.00
29	201	5003	9	\$299.00
30	201	5002	4	\$399.00
31	201	1700	3	\$3,400.00
32	901	5001	2	\$299.00
33	201	5001	2	\$299.00
34	201	2102	5	\$175.00
35	1101	2102	9	\$175.00
36	1301	2102	11	\$175.00
37	1401	2102	7	\$175.00

38	801	2102	5	\$175.00
39	501	2102	12	\$175.00
40	301	2102	8	\$175.00
41	1101	2200	3	\$130.00
42	101	2102	9	\$175.00
43	101	5003	3	\$299.00
44	101	5004	2	\$299.00
45	101	1200	3	\$3,300.00
46	101	1700	5	\$3,400.00
47	1301	1700	3	\$3,400.00
48	1601	1700	7	\$3,400.00
49	1801	1700	4	\$3,400.00
50	1001	1700	5	\$3,400.00
51	1101	1700	2	\$3,400.00
52	1201	1200	8	\$3,300.00
53	501	5001	3	\$299.00
54	501	5003	5	\$299.00
55	501	5004	1	\$299.00
56	501	1700	4	\$3,400.00
57	301	5001	6	\$299.00
58	501	2102	9	\$175.00

SQL Code

```
PROC IMPORT FILE='PURCHASES.XLS'
    OUT=PURCHASES
    DBMS=EXCEL
    REPLACE;
RUN;
```

The SAS log shows that 57 rows of data were imported from the PURCHASES spreadsheet.

SAS Log Results

```
PROC IMPORT FILE='PURCHASES.XLSX'
    OUT=PURCHASES
    DBMS=EXCEL
    REPLACE;
NOTE: 57 rows were imported into WORK.PURCHASES.
RUN;
```

The PURCHASES SAS table after being imported appears below.

Table 5.2: PURCHASES SAS Table

	Custnum	Prodnum	Units	Unitcost
1	1701	1110	1	\$3,200.00
2	101	5001	7	\$299.00
3	701	5001	11	\$299.00
4	701	5003	8	\$299.00
5	701	5002	4	\$399.00
6	701	5004	3	\$299.00
7	701	1700	2	\$3,400.00
8	701	1200	3	\$3,300.00
9	701	1110	2	\$3,200.00
10	1301	5001	3	\$299.00
11	1301	5003	5	\$299.00
12	1301	5002	2	\$399.00
13	901	1700	2	\$3,400.00
14	901	1200	3	\$3,300.00
15	901	1110	5	\$3,200.00
16	901	5001	9	\$299.00
17	901	5002	5	\$399.00
18	901	5003	8	\$299.00
19	901	5004	2	\$299.00
20	401	5001	11	\$299.00
21	401	5002	5	\$399.00
22	401	5003	7	\$299.00
23	401	5004	3	\$299.00
24	401	1700	3	\$3,400.00
25	401	1200	6	\$3,300.00
26	201	5001	6	\$299.00
27	201	5001	6	\$299.00
28	201	5003	9	\$299.00
29	201	5002	4	\$399.00
30	201	1700	3	\$3,400.00
31	901	5001	2	\$299.00
32	201	5001	2	\$299.00

33	201	2102	5	\$175.00
34	1101	2102	9	\$175.00
35	1301	2102	11	\$175.00
36	1401	2102	7	\$175.00
37	801	2102	5	\$175.00
38	501	2102	12	\$175.00
39	301	2102	8	\$175.00
40	1101	2200	3	\$130.00
41	101	2102	9	\$175.00
42	101	5003	3	\$299.00
43	101	5004	2	\$299.00
44	101	1200	3	\$3,300.00
45	101	1700	5	\$3,400.00
46	1301	1700	3	\$3,400.00
47	1601	1700	7	\$3,400.00
48	1801	1700	4	\$3,400.00
49	1001	1700	5	\$3,400.00
50	1101	1700	2	\$3,400.00
51	1201	1200	8	\$3,300.00
52	501	5001	3	\$299.00
53	501	5003	5	\$299.00
54	501	5004	1	\$299.00
55	501	1700	4	\$3,400.00
56	301	5001	6	\$299.00
57	501	2102	9	\$175.00

The IMPORT procedure can also be used to access and read comma separated values (CSV) files to populate SAS tables. In the next example, the PURCHASES CSV file, PURCHASES.csv, is read to populate the SAS table, PURCHASES. The DBMS=CSV option is specified to indicate that a CSV file is to be read, and the REPLACE option allows the PURCHASES table to be replaced if it already exists.

SQL Code

```
PROC IMPORT FILE='PURCHASES.CSV'
            OUT=PURCHASES
            DBMS=CSV
            REPLACE;
RUN;
```

The SAS log shows that 57 rows of data were imported from the PURCHASES CSV file.

SAS Log Results

```
PROC IMPORT FILE='PURCHASES.CSV'
            OUT=PURCHASES
            DBMS=CSV
            REPLACE;
NOTE: 57 rows were imported into WORK.PURCHASES.
RUN;
```

SAS can also access and read Excel spreadsheet files with a LIBNAME statement. This allows worksheet files in an Excel file to be processed in much the same way as SAS data sets are processed in a SAS library. The general syntax for the LIBNAME statement using the EXCEL engine follows:

```
LIBNAME libref EXCEL 'spreadsheet-file-name.xls';
```

The *libref* is the library reference just as if you would use with a SAS data library. *EXCEL* is the name of the engine to use. The *spreadsheet-file-name* refers to the Excel file to use. The next example illustrates assigning the Excel LIBNAME engine to perform a bulk-load insert of the PURCHASES spreadsheet file into the PURCHASES table with the SQL procedure.

Note: A separate SAS/ACCESS software for PC files license is required to use the Excel libname engine.

SQL Code

```
LIBNAME MYEXCEL EXCEL 'PURCHASES.xls';

PROC SQL;
  INSERT INTO PURCHASES
    SELECT *
      FROM MYEXCEL.PURCHASES;
QUIT;
```

The SAS log shows that 57 rows of data were inserted into the WORK.PURCHASES table from the PURCHASES spreadsheet using the Excel LIBNAME engine.

SAS Log Results

```
LIBNAME MYEXCEL EXCEL 'PURCHASES.xls';

PROC SQL;
  INSERT INTO PURCHASES
    SELECT *
      FROM MYEXCEL.PURCHASES;
NOTE: 57 rows were inserted into WORK.PURCHASES.
QUIT;
```

Integrity Constraints

The reliability of databases and the data within them is essential to every organization. Decision-making activities depend on the correctness and accuracy of any and all data contained in key applications, information systems, databases, decision support and query tools, as well as other critical systems. Even the slightest hint of unreliable data can affect decision-making capabilities, accuracy of reports, and, in those worst case scenarios, loss of user confidence in the database environment itself.

Because data should be correct and free of problems, an integral part of every database environment is a set of rules that the data should adhere to. These rules, often referred to as *database-enforced constraints*, are applied to the database table structure itself and determine the type and content of data that is permitted in columns and tables.

By implementing database-enforced integrity constraints, you can dramatically reduce data-related problems and additional programming work in applications. Instead of coding complex data checks and validations in individual application programs, you can build database-enforced constraints into the database itself. This work can eliminate the propagation of column duplication, invalid and missing values, lost linkages, and other data-related problems.

Defining Integrity Constraints

You define integrity constraints by specifying column definitions and constraints at the time a table is created with the CREATE TABLE statement, or by adding, changing, or removing a table's column definitions with the ALTER TABLE statement. The rows in a table are then validated against the defined integrity constraints.

Types of Integrity Constraints

The first type of integrity constraint is referred to as a *column and table constraint*. This type of constraint essentially establishes rules that are attached to a specific table or column. The type of constraint is generally specified through one or two clauses with their distinct values as follows.

Column and Table Constraints

- NOT NULL
- UNIQUE
- CHECK

Preventing Null Values with a NOT NULL Constraint

A null value is essentially a missing or unknown value in the data. When unchecked, null values can often propagate themselves throughout a database. When a NULL appears in a mathematical equation, the returned result is also a null or missing value. When a NULL is used in a comparison or a logical expression, the returned result is unknown. The occurrence of null values presents problems during search, joins, and index operations. The ability to prevent the propagation of null values in a column with a NOT NULL constraint is a powerful feature of the SQL procedure. This constraint should be used as a first line of defense against potential problems that result from the presence of null values and the interaction of queries processing data.

Using the CREATE TABLE or ALTER TABLE statement, you can apply a NOT NULL constraint to any column where missing, unknown, or inappropriate values appear in the data. Suppose that you need to avoid the propagation of missing values in the CUSTCITY (Customer's Home City) column in the CUSTOMER_CITY table. By specifying the NOT NULL constraint for the CUSTCITY column in the CREATE TABLE statement, you prevent the propagation of null values in a table.

SQL Code

```
PROC SQL;
  CREATE TABLE CUSTOMER_CITY
    (CUSTNUM NUM,
     CUSTCITY CHAR(20) NOT NULL);
QUIT;
```

Once the CUSTOMER_CITY table is created and the NOT NULL constraint is defined for the CUSTCITY column, only non-missing data for the CUSTCITY column can be entered. Using the INSERT INTO statement with a VALUES clause, you can populate the CUSTOMER_CITY table while adhering to the assigned NOT NULL integrity constraint.

SQL Code

```
PROC SQL;
  INSERT INTO CUSTOMER_CITY
    VALUES(101,'La Mesa Computer Land')
    VALUES(1301,'Spring Valley Byte Center');
QUIT;
```

The SAS log shows the two rows of data that satisfy the NOT NULL constraint, and the rows successfully being added to the CUSTOMER_CITY table.

SAS Log Results

```
PROC SQL;
  INSERT INTO CUSTOMER_CITY
    VALUES(101,'La Mesa Computer Land')
    VALUES(1301,'Spring Valley Byte Center');
NOTE: 2 rows were inserted into WORK.CUSTOMER_CITY.

  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.22 seconds
      cpu time           0.02 seconds
```

When you define a NOT NULL constraint and then attempt to populate a table with one or more missing data values, the rows will be rejected and the table will be restored to its original state. Essentially, the insert fails because the NOT NULL constraint prevents any missing values for a defined column from populating a table. In the next example, several rows of data with a defined NOT NULL constraint are prevented from being populated in the CUSTOMER_CITY table because one row contains a missing CUSTCITY value.

SQL Code

```
PROC SQL;
  INSERT INTO CUSTOMER_CITY
    VALUES(101,'La Mesa Computer Land')
    VALUES(1301,'Spring Valley Byte Center')
    VALUES(1801,'');
QUIT;
```

The SAS log shows that the NOT NULL constraint has prevented the three rows of data from being populated in the CUSTOMER_CITY table. The violation caused an error message that

resulted in the failure of the add/update operation. The UNDO_POLICY = REQUIRED option reverses all adds/updates that have been performed to the point of the error. This prevents errors or partial data from being propagated in the database table. The following SAS log results illustrate the error condition that caused the add/update operation to fail.

SAS Log Results

```

PROC SQL;
  INSERT INTO CUSTOMER_CITY
    VALUES(101,'La Mesa Computer Land')
    VALUES(1301,'Spring Valley Byte Center')
    VALUES(1801,'');
ERROR: Add/Update failed for data set WORK.CUSTOMER_CITY
because data
  value(s) do not comply with integrity constraint _NM0001_.
NOTE: This insert failed while attempting to add data from
VALUES
  clause 3 to the data set.
NOTE: Deleting the successful inserts before error noted above
to
  restore table to a consistent state.
  QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.02 seconds
      cpu time          0.00 seconds

```

A NOT NULL constraint can also be applied to a column in an existing table that contains data with an ALTER TABLE statement. To successfully impose a NOT NULL constraint, you should not have missing or null values in the column that the constraint is being defined for. This means that the presence of one or more null values in an existing table's column will prevent the NOT NULL constraint from being created.

Suppose that the CUSTOMERS table contains one or more missing values in the CUSTCITY column. If you tried to add a NOT NULL constraint, it would be rejected. You can successfully apply the NOT NULL constraint only when missing values are reclassified or recoded.

SQL Code

```

PROC SQL;
  ALTER TABLE CUSTOMERS
    ADD CONSTRAINT NOT_NULL_CUSTCITY NOT NULL(CUSTCITY);
QUIT;

```

The SAS log shows that the NOT NULL constraint cannot be defined in an existing table when a column's data contains one or more missing values. The violation produces an error message that results in the rejection of the constraint.

SAS Log Results

```

PROC SQL;
  ALTER TABLE CUSTOMERS
    ADD CONSTRAINT NOT_NULL_CUSTCITY NOT NULL(CUSTCITY);
ERROR: Integrity constraint NOT_NULL_CUSTCITY was rejected
because 1
  observations failed the constraint.
  QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.01 seconds
      cpu time          0.00 seconds

```

Enforcing Unique Values with a UNIQUE Constraint

A UNIQUE constraint prevents duplicate values from propagating in a table. If you use a CREATE TABLE statement, then you can apply a UNIQUE constraint to any column where duplicate data is not desired. Suppose that you want to avoid the propagation of duplicate values in the CUSTNUM (Customer Number) column in a new table called CUSTOMER_CITY. By specifying the UNIQUE constraint for the CUSTNUM column with the CREATE TABLE statement, you prevent duplicate values from populating the table.

SQL Code

```

PROC SQL;
  CREATE TABLE CUSTOMER_CITY
    (CUSTNUM NUM UNIQUE,
     CUSTCITY CHAR(20));
QUIT;

```

When you define a UNIQUE constraint and attempt to populate a table with duplicate data values, the rows will be rejected and the table will be restored to its original state prior to the add operation taking place. Essentially, the insert fails because the UNIQUE constraint prevents any duplicate values for a defined column from populating the table. In the next example, several rows of data with a defined UNIQUE constraint are prevented from being populated in the CUSTOMER_CITY table because one row contains a duplicate CUSTNUM value.

SQL Code

```

PROC SQL;
  INSERT INTO CUSTOMER_CITY
    VALUES(101,'La Mesa Computer Land')
    VALUES(1301,'Spring Valley Byte Center')
    VALUES(1301,'Chula Vista Networks');
QUIT;

```

The SAS log shows that the UNIQUE constraint prevented the three rows of data from being populated in the CUSTOMER_CITY table. The violation caused an error message that resulted in the failure of the add/update operation.

SAS Log Results

```

PROC SQL;
  INSERT INTO CUSTOMER_CITY
    VALUES(101,'La Mesa Computer Land')
    VALUES(1301,'Spring Valley Byte Center')
    VALUES(1301,'Chula Vista Networks');
ERROR: Add/Update failed for data set WORK.CUSTOMER_CITY
because data
  value(s) do not comply with integrity constraint _UN0001_.
NOTE: This insert failed while attempting to add data from
VALUES
  clause 3 to the data set.
NOTE: Deleting the successful inserts before error noted above
to
  restore table to a consistent state.
  QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time           1.12 seconds
      cpu time            0.06 seconds

```

Validating Column Values with a CHECK Constraint

A CHECK constraint validates data values against a list of values, minimum and maximum values, as well as a range of values before populating a table. Using either a CREATE TABLE or ALTER TABLE statement, you can apply a CHECK constraint to any column that requires data validation to be performed. In the next example, suppose you want to validate data values in the PRODTYPE (Product Type) column in the PRODUCTS table. When you specify a CHECK constraint against the PRODTYPE column using the ALTER TABLE statement, product type values will first need to match the list of defined values or the rows will be rejected.

SQL Code

```

PROC SQL;
  ALTER TABLE PRODUCTS
    ADD CONSTRAINT CHECK_PRODUCT_TYPE
    CHECK (PRODTYPE IN ('Laptop',
                        'Phone',
                        'Software',
                        'Workstation'));
QUIT;

```

With a CHECK constraint defined, each row must meet the validation rules that are specified for the column before the table is populated. If any row does not pass the validation checks based on the established validation rules, then the add/update operation fails and the table is automatically restored to its original state prior to the operation taking place. In the next example, three rows of data are validated against the defined CHECK constraint established for the PRODTYPE column.

SQL Code

```
PROC SQL;
  INSERT INTO PRODUCTS
    VALUES(5005,'Internet Software',500,'Software',99.)
    VALUES(1701,'Elite Laptop',170,'Laptop',3900.)
    VALUES(2103,'Digital Cell Phone',210,'Fone',199.);
QUIT;
```

The SAS log displays the results after attempting to add the three rows of data. Because one row violates the CHECK constraint with a value of “Fone”, the rows are not added to the PRODUCTS table. The violation produces an error message that results in the failure of the add/update operation.

SAS Log Results

```
PROC SQL;
  INSERT INTO PRODUCTS
    VALUES(5005,'Internet Software',500,'Software',99.)
    VALUES(1701,'Elite Laptop',170,'Laptop',3900.)
    VALUES(2103,'Digital Cell Phone',210,'Fone',199.);
ERROR: Add/Update failed for data set WORK.PRODUCTS because
data
  value(s) do not comply with integrity constraint
CHECK_PRODUCT_TYPE.
NOTE: This insert failed while attempting to add data from
VALUES
  clause 3 to the data set.
NOTE: Deleting the successful inserts before error noted above
to
  restore table to a consistent state.
  QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.09 seconds
      cpu time          0.02 seconds
```

Referential Integrity Constraints

The second type of constraint that is available in the SQL procedure is referred to as a *referential integrity constraint*. Enforced through primary and foreign keys between two or more tables, referential integrity constraints are built into a database environment to prevent data integrity issues from occurring. Specific types of referential integrity constraints and constraint action clauses are used to enforce update and delete operations and consist of the following:

Referential Integrity Constraints

- Primary key
- Foreign key

Referential Integrity Constraint Action Clauses

- RESTRICT (Default)
- SET NULL
- CASCADE

The action clauses are discussed in the “Establishing a Foreign Key” section in this chapter.

Establishing a Primary Key

A primary key consists of one or more columns with a unique value that is used to identify individual rows in a table. Depending on the nature of the columns used, a single column may be all that is necessary to identify specific rows. In other cases, two or more columns may be needed to adequately identify a row in a referenced table. Suppose that you need to uniquely identify specific rows in the MANUFACTURERS table. By establishing the Manufacturer Number (MANUNUM) as the unique identifier for rows, a key is established. The next example specifies the ALTER TABLE statement to create a primary key using MANUNUM in the MANUFACTURERS table.

SQL Code

```
PROC SQL;
  ALTER TABLE MANUFACTURERS
    ADD CONSTRAINT PRIM_KEY PRIMARY KEY (MANUNUM);
QUIT;
```

The SAS log shows that the MANUFACTURERS table has been modified successfully after creating a primary key using the MANUNUM column.

SAS Log Results

```
PROC SQL;
  ALTER TABLE MANUFACTURERS
    ADD CONSTRAINT PRIM_KEY PRIMARY KEY (MANUNUM);
NOTE: Table WORK.MANUFACTURERS has been modified, with 4
columns.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.07 seconds
      cpu time           0.01 seconds
```

Suppose that you also need to uniquely identify specific rows in the PRODUCTS table. By specifying PRODNUM (Product Number) as the primary key, the next example specifies the ALTER TABLE statement to create the unique identifier for rows in the table.

SQL Code

```
PROC SQL;
  ALTER TABLE PRODUCTS
    ADD CONSTRAINT PRIM_PRODUCT_KEY PRIMARY KEY (PRODNUM);
QUIT;
```

The SAS log shows that the PRODUCTS table has been modified successfully after establishing a primary key using the PRODNUM column.

SAS Log Results

```
PROC SQL;
  ALTER TABLE PRODUCTS
    ADD CONSTRAINT PRIM_PRODUCT_KEY PRIMARY KEY (PRODNUM);
NOTE: Table WORK.PRODUCTS has been modified, with 5 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.03 seconds
      cpu time           0.01 seconds
```

Establishing a Foreign Key

A foreign key consists of one or more columns in a table that references or relates to values in another table. The column(s) that is used as a foreign key must match the column(s) in the table that is referenced. The purpose of a foreign key is to ensure that rows of data in one table exist in another table thereby preventing the possibility of lost or missing linkages between tables. The enforcement of referential integrity rules has a positive and direct effect on data reliability issues.

Suppose that you want to ensure that data values in the INVENTORY table have corresponding and matching data values in the PRODUCTS table. By establishing PRODNUM (Product Number) as a foreign key in the INVENTORY table, you ensure a strong level of data integrity between the two tables. This essentially verifies that key data in the INVENTORY table exists in the PRODUCTS table. In the next example, a foreign key is created using the PRODNUM column in the INVENTORY table by specifying the ALTER TABLE statement.

SQL Code

```
PROC SQL;
  ALTER TABLE INVENTORY
    ADD CONSTRAINT FOREIGN_PRODUCT_KEY FOREIGN KEY (PRODNUM)
      REFERENCES PRODUCTS
      ON DELETE RESTRICT
      ON UPDATE RESTRICT;
QUIT;
```

The SAS log displays the successful creation of the PRODNUM column as a foreign key in the INVENTORY table. By specifying the default values ON DELETE RESTRICT and ON UPDATE RESTRICT clauses, you restrict the ability to change the values of primary key data when matching values are found in the foreign key. The execution of any SQL statement that could violate these referential integrity rules is prevented during SQL processing.

SAS Log Results

```

PROC SQL;
  ALTER TABLE INVENTORY
    ADD CONSTRAINT FOREIGN_PRODUCT_KEY FOREIGN KEY (PRODNUM)
      REFERENCES PRODUCTS
      ON DELETE RESTRICT
      ON UPDATE RESTRICT;
NOTE: Table WORK.INVENTORY has been modified, with 5 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.01 seconds
      cpu time          0.01 seconds

```

Suppose that a product of a particular manufacturer is no longer available and has been taken off the market. To handle this type of situation, data values in the INVENTORY table should be set to missing once the product is deleted from the PRODUCTS table. The next example establishes a foreign key using the PRODNUM column in the INVENTORY table and sets values to null with the ON DELETE clause.

SQL Code

```

PROC SQL;
  ALTER TABLE INVENTORY
    ADD CONSTRAINT FOREIGN_MISSING_PRODUCT_KEY FOREIGN KEY (PRODNUM)
      REFERENCES PRODUCTS
      ON DELETE SET NULL;
      QUIT;

```

The SAS log displays the successful creation of the PRODNUM column as a foreign key in the INVENTORY table as well as the effect of the ON DELETE SET NULL clause. Specifying this clause will change foreign key values to missing or null for all of the rows with matching values found in the primary key. The execution of any SQL statement that could violate these referential integrity rules is prevented during SQL processing.

SAS Log Results

```

PROC SQL;
  ALTER TABLE INVENTORY
    ADD CONSTRAINT FOREIGN_MISSING_PRODUCT_KEY FOREIGN KEY
  (PRODNUM)
    REFERENCES PRODUCTS
    ON DELETE SET NULL;
NOTE: Table WORK.INVENTORY has been modified, with 5 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.02 seconds
      cpu time          0.02 seconds

```

Suppose that you want to ensure that changes to key values in the PRODUCTS table automatically flow over or cascade through to rows in the INVENTORY table. This is accomplished by first creating PRODNUM (Product Number) as a foreign key in the

INVENTORY table using the ADD CONSTRAINT clause and referencing the PRODUCTS table. You then specify the ON UPDATE CASCADE clause to enable any changes that are made to the PRODUCTS table to be automatically cascaded through to the INVENTORY table. This ensures that changes to the product number values in the PRODUCTS table automatically occur in the INVENTORY table as well.

SQL Code

```
PROC SQL;
  ALTER TABLE INVENTORY
    ADD CONSTRAINT FOREIGN_PRODUCT_KEY FOREIGN KEY (PRODNUM)
      REFERENCES PRODUCTS
      ON UPDATE CASCADE
      ON DELETE RESTRICT /* DEFAULT VALUE */;
QUIT;
```

The SAS log displays the successful creation of the PRODNUM column as a foreign key in the INVENTORY table. When the ON UPDATE and ON DELETE clauses are specified, the execution of any SQL statement that could violate referential integrity rules is strictly prohibited.

SAS Log Results

```
PROC SQL;
  ALTER TABLE INVENTORY
    ADD CONSTRAINT FOREIGN_PRODUCT_KEY FOREIGN KEY (PRODNUM)
      REFERENCES PRODUCTS
      ON UPDATE CASCADE
      ON DELETE RESTRICT /* DEFAULT VALUE */;
NOTE: Table WORK.INVENTORY has been modified, with 5 columns.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.06 seconds
      cpu time          0.01 seconds
```

Constraints and Change Control

To preserve change control, SAS prohibits changes or modifications to a table that contains a defined referential integrity constraint. When you attempt to delete, rename, or replace a table that contains a referential integrity constraint, an error message is generated and processing stops. The next example illustrates a table copy operation that is performed against a table that contains a referential integrity constraint that generates an error message and stops processing.

SAS Log Results

```
PROC COPY IN=SQLBOOK OUT=WORK;
  SELECT INVENTORY;
RUN;

NOTE: Copying SQLBOOK.INVENTORY to WORK.INVENTORY
(memtype=DATA).
ERROR: A rename/delete/replace attempt is not allowed for a
data set
```

```

involved in a referential integrity constraint.
WORK.INVENTORY.DATA
ERROR: File WORK.INVENTORY.DATA has not been saved because copy
could
not be completed.
NOTE: Statements not processed because of errors noted above.
NOTE: PROCEDURE COPY used:
      real time          0.44 seconds
      cpu time           0.02 seconds
NOTE: The SAS System stopped processing this step because of
errors.

```

Displaying Integrity Constraints

Using the DESCRIBE TABLE statement, the SQL procedure displays integrity constraints along with the table description in the SAS log. The ability to capture this type of information assists with the documentation process by describing the names and types of integrity constraints as well as the contributing columns that they reference.

SQL Code

```

PROC SQL;
  DESCRIBE TABLE MANUFACTURERS;
QUIT;

```

The SAS log shows the SQL statements that were used to create the MANUFACTURERS table as well as an alphabetical list of integrity constraints that have been defined.

SAS Log Results

```

PROC SQL;
  DESCRIBE TABLE MANUFACTURERS;
NOTE: SQL table WORK.MANUFACTURERS was created like:

create table WORK.MANUFACTURERS( bufsize=4096 )
(
  manunum num label='Manufacturer Number',
  manuname char(25) label='Manufacturer Name',
  manucity char(20) label='Manufacturer City',
  manustat char(2) label='Manufacturer State'
);
create unique index manunum on WORK.MANUFACTURERS(manunum);

-----Alphabetic List of Integrity Constraints-
-----

          Integrity
#   Constraint    Type        Variables
-----
1     PRIM_KEY    Primary Key   manunum
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.19 seconds
      cpu time           0.01 seconds

```

Deleting Rows in a Table

In the world of data management, the ability to delete unwanted rows of data from a table is as important as being able to populate a table with data. In fact, data management activities would be severely hampered without the ability to delete rows of data. The DELETE statement and an optional WHERE clause can remove one or more unwanted rows from a table, depending on what is specified in the WHERE clause.

Deleting a Single Row in a Table

The DELETE statement can be specified to remove a single row of data by constructing an explicit WHERE clause on a unique value. The construction of a WHERE clause to satisfy this form of row deletion may require a complex logic construct. So, be sure to test the expression thoroughly before applying it to the table to determine whether it performs as expected. The following example illustrates the removal of a single customer in the CUSTOMERS table by specifying the customer's name (CUSTNAME) in the WHERE clause.

SQL Code

```
PROC SQL;
  DELETE FROM CUSTOMERS2
    WHERE UPCASE(CUSTNAME) = "LAUGHLER";
QUIT;
```

SAS Log Results

```
PROC SQL;
  DELETE FROM CUSTOMERS2
    WHERE UPCASE(CUSTNAME) = "LAUGHLER";
NOTE: 1 row was deleted from WORK.CUSTOMERS2.
QUIT;
NOTE: PROCEDURE SQL used:
      real time           0.37 seconds
```

Deleting More Than One Row in a Table

Frequently, a row deletion affects more than a single row in a table. In these cases, a WHERE clause references a value that occurs multiple times. The following example illustrates the removal of a single customer in the PRODUCTS table by specifying the product type (PRODTYPE) in the WHERE clause.

SQL Code

```
PROC SQL;
  DELETE FROM PRODUCTS
    WHERE UPCASE(PRODTYPE) = `PHONE`;
QUIT;
```

SAS Log Results

```
SAS Log Results
  PROC SQL;
    DELETE FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) = `PHONE`;
  NOTE: 3 rows were deleted from WORK.PRODUCTS.
  QUIT;
  NOTE: PROCEDURE SQL used:
        real time          0.05 seconds
```

Deleting All Rows in a Table

SQL provides a simple way to delete all rows in a table. The following example shows that all rows in the CUSTOMERS table can be removed when the WHERE clause is omitted. Use care when using this form of the DELETE statement because every row in the table is automatically deleted.

SQL Code

```
PROC SQL;
  DELETE FROM CUSTOMERS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  DELETE FROM CUSTOMERS;
  NOTE: 28 rows were deleted from WORK.CUSTOMERS.
  QUIT;
  NOTE: PROCEDURE SQL used:
        real time          0.00 seconds
```

Deleting Tables

The SQL standard permits one or more unwanted tables to be removed (or deleted) from a database (SAS library). During large program processes, temporary tables in the WORK library are frequently created. The creation and build-up of these tables can negatively affect memory and storage performance areas, which can cause potential problems due to insufficient resources. It is important from a database management perspective to be able to delete any unwanted tables to avoid these types of resource problems. Here are a few guidelines to keep in mind.

Before a table can be deleted, complete ownership of the table (that is, exclusive access to the table) should be verified. Although some SQL implementations require a table to be empty in order to delete it, the SAS implementation permits a table to be deleted with or without any rows of data in it. After a table is deleted, any references to that table are no longer recognized and will result in a syntax error. Additionally, any references to a deleted table in a view will also result in an error (see Chapter 8, “Working with Views”). Also, any indexes that are associated with a deleted table are automatically dropped (see Chapter 6, “Modifying and Updating Tables and Indexes”).

Deleting a Single Table

Deleting a table from the database environment is not the same as making a table empty. Although an empty table contains no data, it still possesses a structure; a deleted table contains no data or related structure. Essentially, a deleted table does not exist because the table including its data and structure are physically removed forever. Deleting a single table from a database environment requires a single table name to be referenced in a DROP TABLE statement. In the next example, a single table called HOT_PRODUCTS located in the WORK library is physically removed using a DROP TABLE statement.

SQL Code

```
PROC SQL;
  DROP TABLE HOT_PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  DROP TABLE HOT_PRODUCTS;
NOTE: Table WORK.HOT_PRODUCTS has been dropped.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.38 seconds
```

Deleting Multiple Tables

The SQL standard also permits more than one table to be specified in a single DROP TABLE statement. The next example and corresponding log shows two tables (HOT_PRODUCTS and NOT_SO_HOT_PRODUCTS) being deleted from the WORK library.

SQL Code

```
PROC SQL;
  DROP TABLE HOT_PRODUCTS, NOT_SO_HOT_PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  DROP TABLE HOT_PRODUCTS, NOT_SO_HOT_PRODUCTS;
NOTE: Table WORK.HOT_PRODUCTS has been dropped.
NOTE: Table WORK.NOT_SO_HOT_PRODUCTS has been dropped.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

Deleting Tables That Contain Integrity Constraints

As previously discussed in this chapter, to ensure a high-level of data integrity in a database environment, the SQL standard permits the creation of one or more integrity constraints to be imposed on a table. Under the SQL standard, a table that contains one or more constraints cannot be deleted without first dropping the defined constraints. This behavior further

safeguards and prevents the occurrence of unanticipated surprises such as the accidental deletion of primary or supporting tables.

In the next example, the SAS log shows that an error is produced when an attempt to drop a table containing an ON DELETE RESTRICT referential integrity constraint is performed. The referential integrity constraint causes the DROP TABLE statement to fail, which results in the INVENTORY table not being deleted.

SAS Log Results

```
PROC SQL;
  DROP TABLE INVENTORY;
ERROR: A rename/delete/replace attempt is not allowed for a data set
involved in a referential integrity constraint. WORK.INVENTORY.DATA
WARNING: Table WORK.INVENTORY has not been dropped.
  QUIT;
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
      cpu time          0.00 seconds
```

To enable the deletion of a table that contains one or more integrity constraints, you must specify an SQL statement such as the ALTER TABLE statement and DROP COLUMN or DROP CONSTRAINT clauses. Once a table's integrity constraints are removed, the table can then be deleted.

In the following SAS log, the FOREIGN_PRODUCT_KEY constraint is removed from the INVENTORY table using the DROP CONSTRAINT clause. With the constraint removed, the INVENTORY table is then deleted with the DROP TABLE statement.

SAS Log Results

```
PROC SQL;
  ALTER TABLE INVENTORY
    DROP CONSTRAINT FOREIGN_PRODUCT_KEY;
NOTE: Integrity constraint FOREIGN_PRODUCT_KEY deleted.
NOTE: Table WORK.INVENTORY has been modified, with 5 columns.
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.01 seconds
      cpu time          0.01 seconds

PROC SQL;
  DROP TABLE INVENTORY;
NOTE: Table WORK.INVENTORY has been dropped.
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.34 seconds
      cpu time          0.02 seconds
```

Summary

1. Creating a table using a column-definition list is similar to defining a table's structure with the LENGTH statement in the DATA step (see the "Creating a Table Using Column-Definition Lists" section).
2. Using the LIKE clause copies the column names and attributes from the existing table structure to the new table structure (see the "Creating a Table Using the LIKE Clause" section).
3. Deriving a table from an existing table stores the results in a new table instead of displaying them as SAS output (see the "Deriving a Table and Data from an Existing Table" section).
4. In populating a table, three parameters are specified with an INSERT INTO statement: the name of the table, the names of the columns in which values are inserted, and the values themselves (see the "Adding Data to All of the Columns in a Row" section).
5. Database-enforced constraints can be applied to a database table structure to enforce the type and content of data that is permitted (see the "Integrity Constraints" section).
6. The DELETE statement combined with a WHERE clause selectively removes one or more rows of data from a table (see the "Deleting a Single Row in a Table" section).
7. The SQL standard permits one or more unwanted tables to be removed from a database (SAS library) (see the "Deleting Multiple Tables" section).

Chapter 6: Modifying and Updating Tables and Indexes

Introduction	195
Modifying Tables	195
Adding New Columns	196
Controlling the Position of Columns in a Table.....	198
Changing a Column's Length	200
Changing a Column's Format.....	204
Changing a Column's Label.....	204
Renaming a Column	204
Renaming a Table	206
Indexes	207
Designing Indexes	209
Cardinality	209
Index Selectivity	210
Defining Indexes	211
Creating a Simple Index	212
Creating a Composite Index	213
Preventing Duplicate Values in an Index	214
Modifying Columns Containing Indexes	214
Indexes and Function Calls.....	214
Index Processing Costs	217
Deleting (Dropping) Indexes	217
Updating Data in a Table	218
Summary	219

Introduction

After a table is defined and populated with data, a column as well as its structure might need to be modified. The SQL standard provides Data Definition Language (DDL) statements to permit changes to a table's structure and its data. In this chapter, you will see examples that add and delete columns, modify column attributes, add and delete indexes, rename tables, and update values in rows of data.

Modifying Tables

An important element in PROC SQL is its DDL capabilities. From creating and deleting tables (see Chapter 5, “Creating, Populating, and Deleting Tables”) and indexes to altering table structures and columns, the DDL provides programmers with a way to change (or redefine) the definition of one or more existing tables. The ALTER TABLE statement

permits columns to be added, modified, or dropped in a table with the ADD, MODIFY, or DROP clauses. When a table's columns or attributes are modified, the table's structural dynamics also change. The following sections examine the various ways tables can be modified in the SQL procedure.

Adding New Columns

As requirements and needs change, a database's initial design might require one or more new columns to be added. To accomplish this, complete ownership of the table must be granted. When you have exclusive access, each new column is automatically added at the end of the table's descriptor record. This means that the ALTER TABLE statement's ADD clause modifies the table without reading or writing data.

Suppose that you were given a new requirement to improve your ability to track the status of inventory levels. It is determined that your organization can achieve this new capability by adding a new column to the INVENTORY table. The ADD clause is used in the ALTER TABLE statement to define the new column, INVENTORY_STATUS, and its attributes. The new column's purpose is to identify the following inventory status values: "In-Stock," "Out-of-Stock," and "Back Ordered."

SQL Code

```
PROC SQL;
  ALTER TABLE INVENTORY
    ADD inventory_status char(12);
QUIT;
```

Once the new column is added, the SAS log indicates that 6 columns exist in the INVENTORY table.

SAS Log Results

```

PROC SQL;
  ALTER TABLE INVENTORY
    ADD inventory_status char(12);
NOTE: Table WORK.INVENTORY has been modified, with 6
columns.
  QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.44 seconds

```

The POSITION option as specified in PROC CONTENTS shows the new column, INVENTORY_STATUS, has been added at the end of the INVENTORY table.

Results

Alphabetic List of Variables and Attributes						
#	Variable	Type	Len	Format	Informat	Label
4	invencst	Num	6	DOLLAR10.2		Inventory Cost
2	invenqty	Num	3			Inventory Quantity
6	inventory_status	Char	12			
5	manunum	Num	3			Manufacturer Number
3	orddate	Num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
1	prodnum	Num	3			Product Number

Variables in Creation Order						
#	Variable	Type	Len	Format	Informat	Label
1	prodnum	Num	3			Product Number
2	invenqty	Num	3			Inventory Quantity
3	orddate	Num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
4	invencst	Num	6	DOLLAR10.2		Inventory Cost
5	manunum	Num	3			Manufacturer Number
6	inventory_status	Char	12			

Controlling the Position of Columns in a Table

Column position is not normally important in relational database processing. But, there are times when a particular column order is desired, for example when `SELECT *` (select all) syntax is specified. To add one or more columns in a designated order, the SQL standard provides a couple of choices.

- You can create a new table with the columns in the desired order and load the data into the new table.
- You can create a view that puts the columns in the desired order and then access the view in lieu of the table (see Chapter 8, “Working with Views,” for a detailed explanation).

Suppose that you want to add the `INVENTORY_STATUS` column so that it is inserted between the `ORDDATE` and `INVENCST` columns and is not just added as the last column in the table. The following example shows how this can be done. As before, begin by adding the `INVENTORY_STATUS` column to the `INVENTORY` table. Then, create a new table called `INVENTORY_COPY` and load the data from the `INVENTORY` table in the following column order: `PRODNUM`, `INVENQTY`, `ORDDATE`, `INVENTORY_STATUS`, `INVENCST`, and `MANUNUM`.

SQL Code

```
PROC SQL;
  ALTER TABLE INVENTORY
    ADD INVENTORY_STATUS CHAR(12);
  CREATE TABLE INVENTORY_COPY AS
    SELECT PRODNUM, INVENQTY, ORDDATE, INVENTORY_STATUS,
      INVENCST, MANUNUM
    FROM INVENTORY;
QUIT;
PROC CONTENTS DATA=INVENTORY_COPY POSITION;
RUN;
```

The `PROC CONTENTS` output below shows the positioning of the columns in the new `INVENTORY_COPY` table including the new `INVENTORY_STATUS` column that was added.

Results

Alphabetic List of Variables and Attributes						
#	Variable	Type	Len	Format	Informat	Label
4	INVENTORY_STATUS	Char	12			
5	invencst	Num	6	DOLLAR10.2		Inventory Cost
2	invenqty	Num	3			Inventory Quantity
6	manunum	Num	3			Manufacturer Number
3	orddate	Num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
1	prodnum	Num	3			Product Number

Variables in Creation Order						
#	Variable	Type	Len	Format	Informat	Label
1	prodnum	Num	3			Product Number
2	invenqty	Num	3			Inventory Quantity
3	orddate	Num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
4	INVENTORY_STATUS	Char	12			
5	invencst	Num	6	DOLLAR10.2		Inventory Cost
6	manunum	Num	3			Manufacturer Number

Another way of controlling a table's column order is to create a view or virtual table (for more information on views, see Chapter 8, “Working with Views”) from an existing table by specifying the desired column order. Using a CREATE VIEW statement and a SELECT query, you can construct a new view so that the columns appear in a specific order. Essentially, the view contains only the PROC SQL query’s instructions that were used to create it. It does not contain data. The biggest advantage of creating a view to reorder the columns defined in a table is that a view not only avoids the creation of a physical table, but a view also hides sensitive data from unauthorized viewing. In the next example, a new view called INVENTORY_VIEW is created from the INVENTORY table with selected columns appearing in a specific order.

SQL Code

```
PROC SQL;
  CREATE VIEW INVENTORY_VIEW AS
    SELECT PRODNUM, INVENQTY, INVENTORY_STATUS
      FROM INVENTORY;
QUIT;
```

The PROC CONTENTS output below shows the positioning of the columns in the new view including the new INVENTORY_STATUS column that was added earlier.

Results

Variables in Creation Order				
#	Variable	Type	Len	Label
1	prodnum	Num	3	Product Number
2	invenqty	Num	3	Inventory Quantity
3	INVENTORY_STATUS	Char	12	

Changing a Column's Length

Column definitions (length, informat, format, and label) can be modified with the MODIFY clause in the ALTER TABLE statement. PROC SQL enables a character or numeric column to have its length changed. In the next example, suppose that you want to reduce the length of the character column MANUCITY in the MANUFACTURERS table from 20 bytes to a length of 15 bytes in order to conserve space. The CHAR column-definition is used in the MODIFY clause in the ALTER TABLE statement to redefine the length of the column.

SQL Code

```
PROC SQL;
  ALTER TABLE MANUFACTURERS
    MODIFY MANUCITY CHAR(15);
QUIT;
```

SAS Log Results

```
PROC SQL;
  ALTER TABLE MANUFACTURERS
    MODIFY MANUCITY CHAR(15);
NOTE: Table WORK.MANUFACTURERS has been modified, with 4
columns.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.50 seconds
```

The following PROC CONTENTS output illustrates the changed column length made to the MANUCITY column in the MANUFACTURERS table.

Results

Variables in Creation Order				
#	Variable	Type	Len	Label
1	manunum	Num	3	Manufacturer Number
2	manuname	Char	25	Manufacturer Name
3	manucity	Char	15	Manufacturer City
4	manustat	Char	2	Manufacturer State

The column length can also be changed using the PROC SQL LENGTH= option in the SELECT clause of the CREATE TABLE statement. With this construct, you do not need to use the ALTER TABLE statement, as illustrated in the previous example, as well as using a DATA step. The next example shows the LENGTH= option to reduce the length of the MANUCITY column from 20 bytes to 15 bytes.

SQL Code

```
PROC SQL;
  CREATE TABLE MANUFACTURERS_MODIFIED AS
    SELECT MANUNUM, MANUNAME, MANUCITY LENGTH=15, MANUSTAT
      FROM MANUFACTURERS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE MANUFACTURERS_MODIFIED AS
    SELECT MANUNUM, MANUNAME, MANUCITY LENGTH=15, MANUSTAT
      FROM MANUFACTURERS;
NOTE: Table WORK.MANUFACTURERS_MODIFIED created, with 6 rows
and 4 columns.

QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.12 seconds
      cpu time           0.01 seconds
```

A column that is initially defined as numeric can also have its length changed in PROC SQL. The SQL procedure ignores a field width in these situations and defines all numeric columns with a maximum width of 8 bytes. The reason is that numeric columns are always defined with the maximum precision allowed by SAS. To override this limitation, it is recommended that you use a LENGTH= option in the SELECT clause of the CREATE TABLE statement, or use the LENGTH statement in a DATA step to assign (or reassign) any numeric column

lengths to the desired size. You can also improve query results by assigning indexes only to those columns that have many unique values or that you use regularly in joins.

In the next example, the numeric column MANUNUM has its length changed (or redefined) from 3 bytes to 4 bytes using the LENGTH= option in the SELECT clause of the CREATE TABLE statement.

Note: Recursive references in the target table can create data integrity problems. For this reason, you should refrain from specifying the same table name in the CREATE TABLE statement as specified in the FROM clause.

SQL Code

```
PROC SQL;
  CREATE TABLE MANUFACTURERS_MODIFIED AS
    SELECT MANUNUM LENGTH=4, MANUNAME, MANUCITY, MANUSTAT
      FROM MANUFACTURERS;
QUIT;
```

The PROC CONTENTS output illustrates the changed column length that was assigned to the numeric MANUNUM column in the MANUFACTURERS_MODIFIED table.

Results

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	manucity	Char	15	Manufacturer City
2	manuname	Char	25	Manufacturer Name
1	manunum	Num	4	Manufacturer Number
4	manustat	Char	2	Manufacturer State

Variables in Creation Order				
#	Variable	Type	Len	Label
1	manunum	Num	4	Manufacturer Number
2	manuname	Char	25	Manufacturer Name
3	manucity	Char	15	Manufacturer City
4	manustat	Char	2	Manufacturer State

In the next example, the numeric column MANUNUM has its length changed (or redefined) from 3 bytes to 4 bytes using the LENGTH statement in a DATA step. To avoid truncation or data problems, you should verify that a column that has a shorter length can handle existing

data. Because PROC SQL does not produce any notes or warnings if numeric values are truncated, you need to know your data.

DATA Step Code

```
DATA MANUFACTURERS;
  LENGTH MANUNUM 4.;
  SET MANUFACTURERS;
RUN;
```

SAS Log Results

```
DATA MANUFACTURERS;
  LENGTH MANUNUM 4.;
  SET MANUFACTURERS;
RUN;

NOTE: There were 6 observations read from the dataset
WORK.MANUFACTURERS.
NOTE: The data set WORK.MANUFACTURERS has 6 observations and 4
variables.
NOTE: DATA statement used:
      real time          0.44 seconds
```

The following PROC CONTENTS output illustrates the changed column length that was assigned to the numeric MANUNUM column in the MANUFACTURERS table.

Results

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
1	MANUNUM	Num	4	Manufacturer Number
3	manucity	Char	15	Manufacturer City
2	manuname	Char	25	Manufacturer Name
4	manustat	Char	2	Manufacturer State

Variables in Creation Order				
#	Variable	Type	Len	Label
1	MANUNUM	Num	4	Manufacturer Number
2	manuname	Char	25	Manufacturer Name
3	manucity	Char	15	Manufacturer City
4	manustat	Char	2	Manufacturer State

Changing a Column's Format

You can permanently change a column's format with the MODIFY clause of the ALTER TABLE statement—and not just for the duration of the step. Suppose that you want to increase the size of the DOLLARw.d format from DOLLAR9.2 to DOLLAR12.2 to allow larger product cost (PRODCOST) values in the PRODUCTS table to print properly.

SQL Code

```
PROC SQL;
  ALTER TABLE PRODUCTS
    MODIFY PRODCOST FORMAT=DOLLAR12.2;
QUIT;
```

SAS Log Results

```
PROC SQL;
  ALTER TABLE PRODUCTS
    MODIFY PRODCOST FORMAT=DOLLAR12.2;
NOTE: Table WORK.PRODUCTS has been modified, with 5 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.33 seconds
```

Changing a Column's Label

You can modify a column's label information with the ALTER TABLE statement MODIFY clause. Because the label information is part of the descriptor record, changes to this value have no impact on the data itself. Suppose that you want to change the label that corresponds to the product cost (PRODCOST) column in the PRODUCTS table so that when printed it displays “Retail Product Cost”.

SQL Code

```
PROC SQL;
  ALTER TABLE PRODUCTS
    MODIFY PRODCOST LABEL="Retail Product Cost";
QUIT;
```

SAS Log Results

```
PROC SQL;
  ALTER TABLE PRODUCTS
    MODIFY PRODCOST LABEL="Retail Product Cost";
NOTE: Table WORK.PRODUCTS has been modified, with 5 columns.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

Renaming a Column

The SQL procedure provides an ANSI approach to renaming columns in a table. By specifying the SELECT clause in the CREATE TABLE statement, you can rename columns,

although it can be tedious if a large number of columns exist in the table. The next example illustrates a SELECT clause in a CREATE TABLE statement being used to rename the ITEM column to ITEM_PURCHASED in the PURCHASES table. As the following example illustrates, you should refrain from specifying the same table name in the CREATE TABLE statement as is specified in the FROM clause. Recursive references to the target table can cause data integrity problems.

SQL Code

```
PROC SQL;
  CREATE TABLE PURCHASES AS
    SELECT CUSTNUM, ITEM AS ITEM_PURCHASED, UNITS, UNITCOST
      FROM PURCHASES;
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE PURCHASES AS
    SELECT CUSTNUM, ITEM AS ITEM_PURCHASED, UNITS, UNITCOST
      FROM PURCHASES;
WARNING: This CREATE TABLE statement recursively references the
target table. A consequence of this is a possible data
integrity
problem.
NOTE: Table WORK.PURCHASES created, with 7 rows and 4 columns.

QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.41 seconds
      cpu time           0.02 seconds
```

An alternative approach to renaming columns in a table consists of using the RENAME=SAS data set option in a SELECT statement's FROM clause. Suppose that you want to rename ITEM in the PURCHASES table to ITEM_PURCHASED. In the next example, the RENAME= SAS data set option can be specified in one of two ways, as illustrated below. Either approach is syntactically correct.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PURCHASES (RENAME=ITEM=ITEM_PURCHASED);
QUIT;

< or >

PROC SQL;
  SELECT *
    FROM PURCHASES (RENAME=(ITEM=ITEM_PURCHASED));
QUIT;
```

SAS Log Results

```

PROC SQL;
  SELECT *
    FROM PURCHASES (RENAME=ITEM=ITEM_PURCHASED);
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.31 seconds
      cpu time          0.02 seconds

```

Renaming a Table

The SQL procedure does not provide a standard ANSI approach to renaming a table in a SAS library. Consequently, the DATASETS procedure is the recommended method to accomplish this relatively simple task. Suppose that you want to rename the PRODUCTS table in the WORK library to MANUFACTURED_PRODUCTS.

SAS Code

```

PROC DATASETS LIBRARY=WORK DETAILS;
  CHANGE PRODUCTS = MANUFACTURED_PRODUCTS;
QUIT;

```

SAS Results

Directory							
Libref	WORK						
Engine	V9						
Physical Name	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD3004_KPL-PC_						
Filename	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files_TD3004_KPL-PC_						

#	Name	Member Type	Obs, Entries or Indexes	Vars	Label	File Size	Last Modified
1	CUSTOMERS	DATA	18	3		5120	01Sep13:20:09:10
2	INVENTORY	DATA	7	5		5120	01Sep13:20:09:10
3	INVOICE	DATA	7	6		5120	01Sep13:20:09:10
4	MANUFACTURED_PRODUCTS	DATA	10	5		17408	01Sep13:20:09:10
	MANUFACTURED_PRODUCTS	INDEX	2			13312	01Sep13:20:09:10
5	MANUFACTURERS	DATA	6	4		5120	01Sep13:20:09:10
6	PURCHASES	DATA	57	4		5120	01Sep13:20:09:10

An assortment of novel approaches has been used to rename tables. One approach, which is shown below, uses the CREATE TABLE statement with the SELECT query to create a new table with the desired table name, followed by the DROP TABLE statement to delete the old table. You should be aware, however, that this is not an efficient method to rename a table.

SQL Code

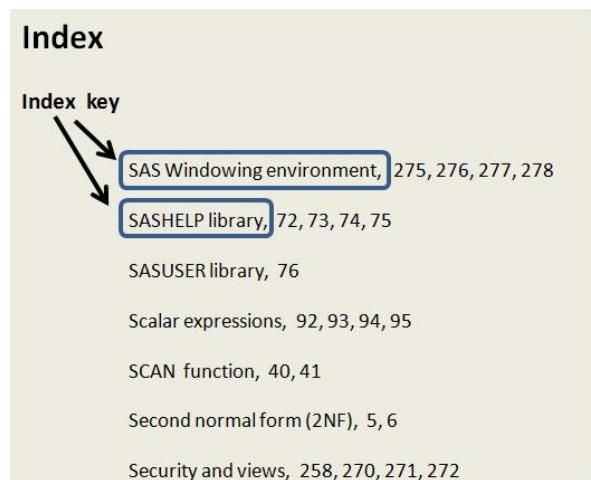
```
PROC SQL;
  CREATE TABLE MANUFACTURED_PRODUCTS AS
  SELECT *
    FROM PRODUCTS;
  DROP TABLE PRODUCTS;
QUIT;
```

Indexes

In database systems, an index is a data structure that is used to locate specific rows of data in a table. In SAS, an index exists as a member type of INDEX and processes a keyword or other identifier to search the index for the specific rows of interest. For years SAS and SQL users have constructed indexes on key variables in their tables to help query processing performance by avoiding a full scan through the data. Whitcher (2008, 10) offers the following advice about index processing, “For PROC SQL to consider using an index, the index must contain all the variables being referenced in the query, and all the variables in the index must also be used in the query.” And for finding and processing unique (distinct) values in a category variable, the index must have been constructed using the UNIQUE keyword with the CREATE UNIQUE INDEX statement.

To better understand how an index works, it is useful to imagine an index located at the end of a book. A book’s index contains keywords that are listed in alphabetical order along with the corresponding page numbers displayed in ascending order, as shown in Figure 6.1. In its purest form, an index in a book is typically made available to enable readers to skip around to different pages or locations in a book. Navigating a book’s contents with an index, commonly referred to as a direct access, is contrasted with the more traditional approach of reading a book in a page-by-page sequential access manner. It is also important to recognize that for an index to retain its intrinsic value, the keyword and page number must correctly access the desired page(s) of interest at all times.

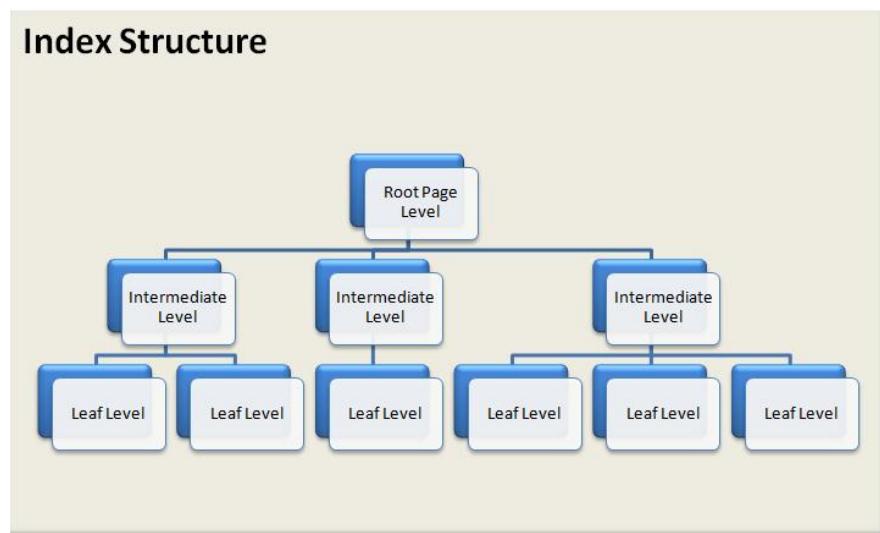
Figure 6.1: Keyword Index in Book



Indexes provide SAS users with an access method that avoids large-table scans and disk sorts, which are frequently used when the optimizer is not able to find an efficient way to service a query. An index consists of one or more columns, representing a key, to uniquely identify rows within a table. When an index is used, it finds the storage location of the rows requested by the query's search criteria and retrieves just these rows of data.

An index is best represented as an inverted tree structure, which is sometimes referred to as a balanced tree (or b-tree), and is displayed in the following diagram. As shown in Figure 6.2, an index consists of a single root page level (starting point), zero or more intermediate levels (or nodes), and a leaf level where the actual rows of data are stored.

Figure 6.2: Inverted Tree Structure Known as a B-Tree



Because indexes are frequently too large to fit completely into primary storage (memory), they are stored on a secondary type of storage such as disk. In b-tree data structures, records or rows of data are stored in locations called *leaves*. B-trees are best suited to handle situations where some or part of the data resides in secondary storage. This provides an environment where the number of secondary disk accesses is reduced (fewer I/O operations), which results in a more cost-effective process.

An index typically is used to improve the speed in which subsets of data are accessed and is composed of one or more character, numeric, or mixed (alphanumeric) types of columns. Rather than sequentially accessing rows of data or physically sorting a table (as performed with an ORDER BY clause or a BY statement in PROC SORT), an index is designed to set up a logical data arrangement without the need to physically sorting it. This has a distinct advantage of reducing CPU and memory requirements, as well as reducing data access time when using WHERE clause processing.

For example, once a specific part number is known, an index can be used to look up the location of the part along with its manufacturer, cost, and availability far more efficiently than with other methods. For more information about using indexes in SAS, see *The Complete Guide to Using SAS Indexes* by Michael Raithel.

Designing Indexes

There is no rule that says a table has to have an index, but when an index is available it can make information access and retrieval more efficient and considerably faster.

When a query is executed in SAS software, the SQL optimizer evaluates the costs associated with the available methods and uses the most efficient method to process data. Processing can occur using a sequential table scan, or with an index if one exists. When a sequential table scan is performed, SAS starts at the beginning of the table, steps row-by-row through all of the rows in the table, and processes (retrieves) rows that match the selection criteria that is specified in the WHERE clause of the query.

Indexes should be designed to provide efficient database processing when triggered (or referenced) in a query. To better understand the impact that an index has on a database application, the following things should be kept in mind:

- Indexes can improve the performance of queries that do not modify data because the optimizer has more choices to select from to in order to determine the fastest way to access data.
- A table with too many indexes might actually experience a performance degradation when using INSERT, MODIFY, or DELETE operations because all indexes must be adjusted to correspond to the changes made to the data in the table.
- A query that specifies an exact match comparison can benefit from an index. For example,


```
WHERE prodnum = 5001;
      < or >
WHERE prodtype = "Laptop";
```
- A query that specifies a value between a range of values can benefit from an index. For example,


```
WHERE invenqty BETWEEN 3 AND 10;
      < or >
WHERE invenqty >= 3 AND invenqty <= 10;
```
- Queries that produce sorted output without specifying an explicit sort operation.
- Queries that use a LIKE comparison operator can benefit from an index when the search pattern begins with a specific character string, such as “Lap%”, but not when the search pattern begins with a wildcard, such as “%ware”.
- Forcing the SQL optimizer to use an index in a query (with an IDXWHERE= or IDXNAME= data set option) with a small table can impede performance because SAS would traverse the index looking for matches instead of allowing the software to process data using a sequential table scan.

Cardinality

In the context of a database table, cardinality refers to the uniqueness, or lack thereof, of data values contained in a specific column of a table. The cardinality of a set of values refers to the uniqueness of a number of elements in a set. For example, the set CUSTNUM= {101, 201, 301, 401} contains four unique elements, and has a cardinality of four. In contrast, the set

`PRODTYPE = {Workstation, Laptop, Software, Software}` contains four elements, three of which are unique, which results in a cardinality of three.

An understanding of the rules of cardinality and how cardinality affects a database application can help determine an optimal query plan. Table 6.1 illustrates and describes the three types of cardinality: low, normal, and high. The cardinality of a set becomes higher as the more unique elements are contained in a column. Conversely, the lower the cardinality, the more duplicate elements are contained in a column.

Table 6.1: Rules of Cardinality

Data Values Contain Large Number of Duplicate Values	Data Values Contain Some Unique/Some Duplicate Values	Data Values Contain Large Number of Unique Values
Low cardinality contains few, if any, unique values. Typical examples of column values with low cardinality include flags and switches, Boolean values, or gender. Columns of this type typically contain a larger number of duplicate values and make for poor indexes.	Normal cardinality contains a limited number of possible values. Typical examples of column values with normal cardinality include product type, manufacturer state, customer name, or customer home city. Columns of this type generally contain some unique values as well as some duplicate values.	High cardinality contains values that are highly unique. Column values with high cardinality are typically used for identification purposes and include manufacturer numbers, invoice numbers, or email addresses. Columns of this type generally consist of large numbers of unique values and make the best indexes.

Index Selectivity

An index is most effective when it is highly selective. This means that a column with high cardinality (as was presented in the previous section) is most selective when the ratio of distinct values divided by the number of rows in the table is as close to 1 as possible. The selectivity formula for an index can be quantified as follows:

$$\text{Selectivity} = \frac{\text{Unique (Distinct) Rows}}{\text{Total Number Rows}}$$

Note: Perfect selectivity can only be reached on NOT NULL columns.

To determine the degree of selectivity, the number of distinct rows, and the total rows in a table for any column in a table, use the following SQL statement:

```
SELECT      COUNT (DISTINCT (column-name)) / COUNT (*) AS
SELECTIVITY,
           COUNT (DISTINCT (column-name)) AS DISTINCT_VALUES,
           COUNT (*) AS TOTAL_ROWS
      FROM    table-name;
```

<or>

```
SELECT      COUNT (DISTINCT (column-name)) / COUNT (*) AS SELECTIVITY
      FROM    table-name;
```

To illustrate an example of good (and in this case perfect) selectivity, the following query selects the CUSTNUM column divided by the total number of rows in the CUSTOMERS table to compute the selectivity ratio:

SQL Code

```
PROC SQL;
  SELECT COUNT(DISTINCT(CUSTNUM)) / COUNT(*) AS SELECTIVITY
    FROM CUSTOMERS;
QUIT;
```

SAS Results

SELECTIVITY
1

A result of 1 (18 distinct values / 18 total rows) indicates perfect selectivity and is an ideal candidate for an index because it is 100% selective of the rows in a table.

To illustrate an example of poor (and in this case bad) selectivity, the following query selects the PRODTYPE column divided by the total number of rows in the PRODUCTS table to compute the selectivity ratio:

SQL Code

```
PROC SQL;
  SELECT COUNT(DISTINCT(PRODTYPE)) / COUNT(*) AS SELECTIVITY
    FROM PRODUCTS;
QUIT;
```

SAS Results

SELECTIVITY
0.4

A result of 0.4 (4 distinct values / 10 total rows) represents low cardinality as well as less than good selectivity. Consequently, this column might not be an ideal candidate for an index because a sequential full table scan might be a more efficient way to process rows in a table.

Note: A column's selectivity should be recalculated from time-to-time because row inserts, updates, and deletions could change the selectivity ratio.

Defining Indexes

When defining an index, first understand the purpose the index is to serve. An important thing to remember about indexes is that they should be created only when absolutely necessary. Too many or unnecessary indexes use up computer resources and impede performance. Although the typical index takes up less space than is required by the table itself, it still represents a copy of some part of a table, and therefore requires storage space to store its contents. For this reason, care should be used when deciding when and what indexes to create.

To help determine when indexes are necessary, consider existing data as well as the way the base table(s) will be used. Acquaint yourself with queries classified as mission critical and/or essential to the success of the organization or a process. Then, determine how the data is dispersed (or the variability of the data) in the underlying base table(s) by using analytical tools such as the FREQ procedure.

If an index is used to specify some order within a table, such as manufacturer number or product number in the PRODUCTS table, you should fully assess what the impact of that index will be.

Sometimes the column(s) making up an index is obvious, and other times it is not. An index should permit the greatest flexibility so that every column in a table can be accessed and retrieved. Improvements with query results can also be achieved by assigning indexes to the most discriminating columns in a table (or columns that have many unique values), as well as to the columns that are used regularly in queries.

When an index is specified for one or more tables, a join process might actually process faster. The SQL optimizer might decide to use an index when certain conditions permit its use. Here are a few things to consider prior to creating an index:

- If the table is small, sequential processing might be just as fast, or faster, than processing with an index.
- If the page count, as displayed in the CONTENTS procedure, is less than three pages, then an index might serve little or no value.
- Avoid creating more indexes than are absolutely necessary.
- If the data subset for the index is large, then sequential access might be more efficient than using the index.
- If the percentage of matches is approximately 15% or less (referred to as the 15% rule) of the overall population, then index usage might be beneficial.
- The costs associated with maintaining an index can outweigh its performance value, because an index is updated each time a row in a table is added, deleted, or modified.

Two types of indexes can be defined and used in PROC SQL: *simple* and *composite*. When a simple index is created, it references only a single column. In contrast, a composite index references two or more columns in a table.

Creating a Simple Index

A simple index is specifically defined for one column in a table and must be the same name as the column. Suppose that you want to create an index that consists of product type (PRODTYPE) in the PRODUCTS table. Once created, the index becomes a separate object located in the SAS library.

SQL Code

```
PROC SQL;          ①
  CREATE INDEX PRODTYPE ON PRODUCTS (PRODTYPE);
QUIT;
```

② ③

SAS Log Results

```

PROC SQL;          ❶          ❷          ❸
      CREATE INDEX PRODTYPE ON PRODUCTS(PRODTYPE);
NOTE: Simple index PRODTYPE has been defined.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time           0.37 seconds

```

- ❶ The simple index is assigned a name of PRODTYPE, which must be the same as the column name.
- ❷ The simple index is defined on the PRODUCTS table.
- ❸ The PRODTYPE column in the PRODUCTS table is designated as the column to be used by the index.

Creating a Composite Index

A composite index is specifically defined for two or more columns in a table and must have a different name from the column names. Suppose that you want to create an index that consists of manufacturer number (MANUNUM) and product type (PRODTYPE) located in the PRODUCTS table. You should be aware that only one composite index is allowed per set of columns, but more than one composite index is allowed. The composite index, as with the simple index, becomes a separate object located in the SAS library.

SQL Code

```

PROC SQL;
  CREATE INDEX ❶          ❷          ❸
    MANNUM_PRODTYPE ON PRODUCTS(MANNUM, PRODTYPE);
QUIT;

```

SAS Log Results

```

PROC SQL;
  CREATE INDEX ❶          ❷          ❸
    MANNUM_PRODTYPE ON PRODUCTS(MANNUM, PRODTYPE);
NOTE: Composite index MANNUM_PRODTYPE has been defined.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time           0.00 seconds

```

- ❶ The composite index is assigned a name of MANNUM_PRODTYPE, which is used to represent the MANNUM and PRODTYPE column names.
- ❷ The composite index is defined on the PRODUCTS table.
- ❸ The MANNUM and PRODTYPE columns in the PRODUCTS table are designated as the columns to be used by the index.

Preventing Duplicate Values in an Index

The UNIQUE keyword prevents the entry of a duplicate value in an index. You should use this keyword with care because there might be times when more than one occurrence of a data value in a table is necessary. When multiple occurrences of the same value appear in a table, the UNIQUE keyword is rejected and the index is not created for that particular column.

Modifying Columns Containing Indexes

Altering the attributes of a column that contains an associated index (simple or composite) does NOT prohibit the values in the altered column from using the index. But, if a column that contains an index is dropped, then the index is also dropped. Accordingly, when a column is dropped, any data in that index is also lost.

Indexes and Function Calls

Readers should use caution when constructing WHERE clause expressions with the use of indexes. The SQL optimizer might prevent the use of an index and the optimization of a WHERE clause expression that contains and uses a function call. One specific function that should be avoided in a WHERE clause with index processing is the UPCASE function.

The next example shows two simple queries – the first query that specifies an UPCASE function in the WHERE clause expression and the second query that excludes the UPCASE function in the WHERE clause expression. The SAS log shows that the SQL optimizer did not optimize the first SQL query's WHERE clause expression to use the index, but the second query did. The INFO: message showed that the simple index Prodbname was selected for WHERE clause optimization. Note: To learn which, if any, of the available indexes is triggered by the WHERE clause and the SQL optimizer the MSGLEVEL=I System option is specified.

SQL Code

```

proc sql;
  create table work.Products as
    select *
      from mydata.Products;
  create index Prodbname
    on work.Products;
quit;

proc contents data=work.Products;
run;

options msglevel=i;

proc sql;
  select *
    from work.Products
    where UPCASE(Prodbname) CONTAINS 'SOFTWARE';

  select *
    from work.Products

```

```
      where Prodname CONTAINS 'Software';
quit;
```

SAS Log Results

```
proc sql;
  create table work.Products as
    select *
      from mydata.Products;
NOTE: Table WORK.PRODUCTS created, with 10 rows and 5 columns.

  create index Prodname
    on work.Products;
NOTE: Simple index Prodname has been defined.
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.01 seconds
      cpu time          0.02 seconds

proc contents data=work._all_;
run;

NOTE: PROCEDURE CONTENTS used (Total process time):
      real time          0.11 seconds
      cpu time          0.11 seconds
```

```
options msglevel=i;

proc sql;
  select *
    from work.Products
      where UPCASE(Prodname) CONTAINS 'SOFTWARE';
```

```
select *
  from work.Products
    where Prodname CONTAINS 'Software';
```

```
INFO: Index prodname selected for WHERE clause optimization.
```

```
quit;
```

Results

#	Name	Member Type	File Size	Last Modified
1	PRODUCTS	DATA	192KB	12/11/2018 14:45:15
	PRODUCTS	INDEX	24KB	12/11/2018 14:45:15
2	REGISTRY	ITEMSTOR	32KB	12/11/2018 14:13:54
3	SASGOPT	CATALOG	12KB	12/11/2018 14:15:36
4	SASMAC1	CATALOG	208KB	12/11/2018 14:13:55
5	SASMAC2	CATALOG	20KB	12/11/2018 14:13:56
6	SASMAC3	CATALOG	20KB	12/11/2018 14:13:56
7	SASMAC4	CATALOG	20KB	12/11/2018 14:45:15
8	SASMAC5	CATALOG	20KB	12/11/2018 14:13:57
9	SASMAC6	CATALOG	20KB	12/11/2018 14:13:57
10	SASMAC7	CATALOG	20KB	12/11/2018 14:13:57
11	SASMAC8	CATALOG	20KB	12/11/2018 14:13:58
12	SASMAC9	CATALOG	20KB	12/11/2018 14:13:58
13	SASMACR	CATALOG	20KB	12/11/2018 14:15:36

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	2
First Data Page	1
Max Obs per Page	1281
Obs in First Data Page	10
Index File Page Size	8192
Number of Index File Pages	2
Number of Data Set Repairs	0

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
3	manunum	Num	3		Manufacturer Number
5	prodcost	Num	5	DOLLAR9.2	Product Cost
2	prodname	Char	25		Product Name
1	prodnum	Num	3		Product Number
4	prodtype	Char	15		Product Type

Alphabetic List of Indexes and Attributes		
#	Index	# of Unique Values
1	prodname	10

Index Processing Costs

When processing observations in a sequential manner without the use of an index, SAS reads and processes all the observations from a page of disk into memory continuing this process until the end of file. In some scenarios sequential access can be considerably costlier since the SQL optimizer will need to perform a full scan through the data.

A common assumption about an index is that as the size of the subset, based on the WHERE clause expression, becomes smaller, the index performance gains become larger. Olson (2000) describes how index processing often incurs additional computing resources. In the creation, maintenance, and usage of an index, CPU, disk space, and I/O operations will almost certainly increase so time spent understanding your query's requirements should result in the construction of better and more efficient indexes. With index processing, SAS determines the location of the next observation using the index, and reads the observations on the page, and if necessary from a new page, satisfying the WHERE clause expression. One way to reduce the possible index processing costs is to first sort the data table in ascending order by the key column or columns and then index the table by that sorted key column or columns. As a result, performance costs using the index tend to be better since fewer reads are often performed.

Deleting (Dropping) Indexes

When one or more indexes are no longer needed, the DROP INDEX statement can be used to remove them. Suppose that you no longer need the composite index MANUNUM_PRODTYPE (which was created earlier) because processing requirements have changed. The next example illustrates a single composite index being deleted from the SAS library.

SQL Code

```
PROC SQL;
  DROP INDEX MANUNUM_PRODTYPE
    FROM PRODUCTS;
QUIT;
```

SAS Log Results

<pre>PROC SQL; DROP INDEX MANUNUM_PRODTYPE FROM PRODUCTS; NOTE: Index MANUNUM_PRODTYPE has been dropped. QUIT; NOTE: PROCEDURE SQL used: real time 0.00 seconds</pre>
--

According to the ANSI SQL standard, two or more indexes can also be deleted in a DROP INDEX statement. The next example illustrates the MANUNUM and PRODTYPE indexes being deleted from the SAS library in a single DROP INDEX statement.

SQL Code

```
PROC SQL;
  DROP INDEX MANUNUM, PRODTYPE
    FROM PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  DROP INDEX MANUNUM, PRODTYPE
    FROM PRODUCTS;
NOTE: Index MANUNUM has been dropped.
NOTE: Index PRODTYPE has been dropped.
      QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.00 seconds
      cpu time           0.01 seconds
```

Updating Data in a Table

Once a table is populated with data, you might need to update values in one or more of its rows. Column values in existing rows in a table can be updated with the UPDATE statement. The key to successful row updates is the creation of a well-constructed SET clause and WHERE expression. If the WHERE expression is not constructed correctly, the possibility of an update error is great.

Suppose that all laptops in the PRODUCTS table have just been discounted by 20 percent and the new price is to take effect immediately. The update would compute the discounted product cost for “Laptop” computers only. For example, the discounted price for a laptop computer would be reduced to \$2,720.00 from \$3,400.00.

SQL Code

```
PROC SQL;
  UPDATE PRODUCTS
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2)
    WHERE UPCASE(PRODTYPE) = 'LAPTOP';

  SELECT *
    FROM PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  UPDATE PRODUCTS
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2)
    WHERE UPCASE(PRODTYPE) = 'LAPTOP';
NOTE: 1 row was updated in WORK.PRODUCTS.
      SELECT *
        FROM PRODUCTS;
QUIT;
```

NOTE: PROCEDURE SQL used:	
real time	0.00 seconds

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
1700	Travel Laptop	170	Laptop	\$2,720.00
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00
5001	Spreadsheet Software	500	Software	\$299.00
5002	Database Software	500	Software	\$399.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00

Summary

1. Data Definition Language (DDL) statements provide programmers with a way to redefine the definition of one or more existing tables (see the “Modifying Tables” section).
2. As one or more new columns are added to a table, each is automatically added at the end of a table’s descriptor record (see the “Adding New Columns” section).
3. To add one or more columns in a designated order, the SQL standard provides a couple of choices to choose from (see the “Controlling the Position of Columns in a Table” section).
4. PROC SQL enables a character column (but not a numeric column) to have its length changed (see the “Changing a Column’s Length” section).
5. A column’s format and label information can be modified with a MODIFY clause (see the “Changing a Column’s Format” and “Changing a Column’s Label” sections).
6. The RENAME= SAS data set option must be used in a FROM clause to rename column names (see the “Renaming a Column” section).
7. The DATASETS procedure is the recommended way to rename tables (see the “Renaming a Table” section).
8. An index consists of one or more columns used to uniquely identify each row within a table (see the “Indexes” section).

9. An understanding of the rules of cardinality and how it affects a database application can help determine an optimal query plan (see the “Creating a Simple Index” section).
10. An index is most effective when it is highly selective (see the “Creating a Composite Index” section).
11. Column values in existing rows in a table can be modified with the UPDATE statement (see the “Updating Data in a Table” section).

Chapter 7: Coding Complex Queries

Introduction	222
Introducing Complex Queries.....	222
Joins.....	222
Why Joins Are Important.....	223
Information Retrieval Based on Relationships.....	223
DATA Step Merges versus PROC SQL Joins	223
Types of Complex Queries.....	224
Demystifying Join Algorithms.....	226
Influencing Joins with a Little Magic.....	227
Cartesian Product Joins	229
Inner Joins.....	230
Equijoins	230
Non-Equijoins.....	232
Reflexive or Self Joins	233
Using Table Aliases in Joins	235
Performing Computations in Joins	236
Joins with Three Tables	237
Joins with More Than Three Tables	238
Outer Joins	240
Left Outer Joins	241
Right Outer Joins	244
Full Outer Joins	245
Subqueries	246
Alternate Approaches to Subqueries	247
Passing a Single Value with a Subquery.....	248
Passing More Than One Row with a Subquery.....	251
Comparing a Set of Values	252
Correlated Subqueries	254
Set Operations.....	256
Rules for Set Operators.....	256
Set Operators and Precedence	257
Accessing Rows from the Intersection of Two Queries.....	257
Accessing Rows from the Combination of Two Queries	259
Concatenating Rows from Two Queries.....	261
Comparing Rows from Two Queries.....	263
Data Structure Transformations	265
Types of Transformations	265
Concatenating Tables of Data	266
Interleaving Tables of Data	268
Splitting a Table into Multiple Tables.....	269
Complex Query Applications	272
One-to-One, One-to-Many, Many-to-One, and Many-to-Many Relationships	272
Processing First, Last, and Between Rows for BY-and Groups.....	277
Determining the Number of Rows in an Input Table.....	281

Identifying Tables with the Most Indexes.....	282
Nearest Neighbor	283
Summary	287

Introduction

In previous chapters, our discussion of queries was confined to a single table referenced with a SELECT statement. The real strength of the relational approach is the ability it gives to constructing queries that refer to several tables or even to other queries. These types of queries are referred to as *complex queries*. PROC SQL supports coding constructs for inner and outer joins consisting of multiple tables, implementing queries that control other queries through a process known as nesting, and combining output as a single table from multiple queries.

Introducing Complex Queries

Let's look at queries of a more complex nature that utilize all the features of the SQL procedure. Four complex query constructs are illustrated in this chapter.

Inner Joins

A maximum of 256 tables can be referenced in a FROM and optional WHERE clause of a SELECT statement.

Outer Joins

A maximum of two tables are referenced in a FROM and ON clause of a SELECT statement.

Subqueries

A query is embedded (nested) in the WHERE clause of a main query.

Set Operations

Results are created from two or more separate queries.

Joins

Joining two or more tables of data is a powerful feature in the relational model. The SQL procedure enables you to join tables of information quickly and easily. Linking one piece of information with another piece of information is made possible when at least one column is common to each table. A maximum of 256 tables can be combined using conventional (inner) join techniques, as opposed to a maximum of two tables at a time using outer join techniques.

This chapter discusses a number of join topics including why joins are important, the differences between the various join techniques, the importance of the WHERE clause in creating joins, creating and using table aliases, joining three or more tables of data, outer (left, right, and full) joins, subqueries, and set operations. It is important to recognize that many of these techniques can be accomplished using DATA step programming techniques, but the

simplicity and flexibility found in the SQL procedure makes it especially useful, if not indispensable, as a tool for the practitioner.

Why Joins Are Important

As relational database systems continue to grow in popularity, the need to access normalized data stored in separate tables becomes increasingly important. By relating matching values in key columns in one table with key columns in the other table(s), you can retrieve information as if the data were stored in one huge file. The results can provide new and exciting insights into possible data relationships.

Information Retrieval Based on Relationships

Being able to define relationships between multiple tables and retrieve information based on these relationships is a powerful feature of the relational model. A join of two or more tables provides a means of gathering and manipulating data in a single SELECT statement. You join two or more tables by specifying the table names in a SELECT statement. Joins are specified on a minimum of two tables at a time, where a column from each table is used for the purpose of connecting the two tables. Connecting columns should have “*like*” values and the same column attributes because the join’s success is dependent on these values.

In a typical join, you name the relevant columns in the SELECT statement, specify the tables to be joined in the FROM clause, and in the WHERE clause you specify the relationship that you want revealed. That is, you describe the data subset that you want to produce. To be of use (and of a manageable size), your join needs a WHERE clause to constrain the results and ensure their utility and relevance.

Note: When you create a join without a WHERE clause, you are creating an internal, virtual table called a *Cartesian product*. This table can be extremely large because it represents all possible combinations of rows and columns in the joined tables.

DATA Step Merges versus PROC SQL Joins

The purpose of this section is to briefly explain the differences (and similarities) between DATA step merges and PROC SQL joins. The reason for addressing this topic is that over the years questions have been asked about which approach is superior, inferior, faster, slower, better, worse, easier, harder, more efficient, less efficient, more demanding on a programmer’s time, less demanding on a programmer’s time, more supportable, less supportable, etc. My usual reply mentions that, “Only you and your organization will be able to answer this question completely, and here’s why.”

Besides the obvious syntax and implementation method differences with the merge using a DATA step construct and the join using a procedure supplied by SAS, both approaches achieve essentially the same results. In fact, both approaches combine two or more data sets (or tables) horizontally by matching keys, and are able to process one-to-one, one-to-many, many-to-one, and many-to-many matched combinations. Table 7.1 provides a few considerations that every SAS and SQL user should ask and answer before deciding which approach is most appropriate to use.

Table 7.1: Merge versus Join Features/Considerations

Features / Considerations	DATA Step Merge	PROC SQL Join
Is there a limit to the number of data sets (or tables) that can be processed, other than disk space?	No	Yes (Limit of 256 tables)
Is the code portable to other database implementations?	No	Yes
Is the approach a standardized method for specifying database requests?	No	Yes
Does a sort need to be performed for processing BY-groups?	Yes	No
Is a common variable name required when combining data sets (or tables)?	Yes	No
When combining data sets (or tables) is the duplicate matching column automatically overlaid?	Yes	No
Are the results automatically printed after combining data sets (or tables)?	No	Yes

Types of Complex Queries

The SQL procedure supports a number of complex query constructs (sometimes referred to as join types). From inner joins to left, right, and full outer joins, this chapter provides a comprehensive look at the various forms of SELECT statements that can be constructed to manipulate and transform multiple tables. The SQL procedure supports four categories of data relationships: one-to-one, one-to-many, many-to-one, and many-to-many, where each category is classified by how the rows in each table relate to one another.

One-to-one relationships represent the simplest of join operations. It's characterized by the tables that have a BY-group column with no repeats of BY-values (a unique BY-column) in any of the tables. It assumes that the rows of data in each table are in the exact same relative order, where the combined results will comprise row one in table one with row one in table two, row two in table one with row two in table two, row three in table one with row three in table two, and so on.

One-to-many and many-to-one relationships are no more complicated than one-to-one relationships, and are easily represented by the SQL procedure. These types of relationships are characterized by one table having no repeats of BY-values and the other table having one or more repeats of BY-values.

Many-to-many relationships can be a bit more complicated than one-to-one, one-to-many, and many-to-one relationships. Many-to-many relationships are characterized by all tables having one or more repeats of by-values. There are notable advantages of using the SQL procedure to perform these types of joins. First, the column names from the source tables do not have to be identically named. Second, pre-sorting the tables is not necessary because the SQL optimizer will determine whether any indexes are available (see Chapter 6, “Modifying and Updating Tables and Indexes”) or if ordering the data is necessary using an implied ORDER BY clause.

Additional topics and examples include subqueries and set operations such as UNION, INTERSECT, and EXCEPT operators. Table 7.2 presents the various types of complex queries that are available in the SQL procedure.

Table 7.2: Types of Complex Queries

Query Type	Description
Cartesian Product or Cross Join	A join that creates a table that represents all of the combinations of rows and columns from two or more tables. It is represented by the absence of a WHERE clause.
Inner Joins	A join that only retrieves rows with matching values from two or more tables (maximum of 256 tables). This type of join is referred to as a conventional type of join.
Equijoin	A join with an equality condition (for example, equal sign “=”) specified between columns in two or more tables.
Non-Equijoin	A join with an inequality condition (for example, NE, >, <) specified between columns in two or more tables.
Reflexive or Self Join	A join that combines a table with itself.
Outer Joins	A join that retrieves rows with matching values while preserving some or all of the unmatched rows from one or both tables (maximum of 2 tables).
Left Outer Join	A join that preserves unmatched rows from the left table.
Right Outer Join	A join that preserves unmatched rows from the right table.
Full Outer Join	A join that preserves unmatched rows from the left and right tables.
Subqueries	A query within another query, which is sometimes referred to as a nested query, that retrieves rows from one table based on values in another table.
Simple Subquery	A self-contained and independent query within another query that returns single or multiple values from an inner query.
Correlated Subquery	An outer query that passes value(s) to an inner query that after execution passes the results back to the outer query.

Query Type	Description
Set Operations	These operators combine or concatenate query results vertically.
UNION	Combines all unique (non-duplicate) rows from both queries.
INTERSECT	Combines all matched rows from the first query with rows in the second query.
EXCEPT	Produces rows from the first query that do not appear in the second query.
OUTER UNION	Concatenates (appends) the results from both queries.

Demystifying Join Algorithms

The SQL procedure contains an optimizer whose purpose is to use the best plan for query execution. A join algorithm locates, for each distinct value of the join attributes, an ordered list of elements in each relation that display that value. The SQL optimizer uses criteria to determine which join algorithm to use to optimize the query. The join construct, the structure of the data, the definition of indexes, the availability of real memory, and other factors can influence the join algorithm that is selected by the optimizer. The specific details of each of the join algorithms are described below.

Nested Loop

A nested loop join algorithm might be selected by the SQL optimizer when processing small tables of data where one table is considerably smaller than the other table, the join condition does not contain an equality condition, first row matching is optimized, or using a sort-merge or hash join has been eliminated. This algorithm operates by looping through the smaller of the tables looking for a matching key in the larger table. A nested loop join algorithm is extremely sensitive to the contents of the right table because it processes the right table for each row of the left table. For this reason, this join algorithm tends to be more CPU intensive than other choices, particularly as the table sizes increase.

Sort-Merge

A sort-merge join algorithm might be selected by the SQL optimizer when the tables are small to medium size. This algorithm operates by first sorting the tables of data (if necessary) using one or more key columns, and then, for each row in the left table, the algorithm reads all matching rows in the right table. The SQL optimizer considers using a sort-merge join algorithm when the conditions for using an index or hash join algorithm have been eliminated from consideration.

Index

An index join algorithm might be selected by the SQL optimizer when indexes created on each of the columns participating in the join relationship will improve performance. This algorithm operates by looking up each row of the smaller table by accessing the index of the larger table. The SQL optimizer considers using an index join algorithm when the larger table has an associated index with all the join keys, the tables are related

with an equality condition, and/or join conditions use the AND operator between multiple expressions.

Hash

A hash join algorithm might be selected by the SQL optimizer when sufficient memory is available to the system, and the BUFFERSIZE option is large enough to store the smaller of the tables into memory. This algorithm operates by sequentially scanning the larger table and performing row-by-row lookup against the smaller table. The SQL optimizer considers using a hash join algorithm when an index join algorithm has been eliminated from consideration.

Influencing Joins with a Little Magic

The SQL procedure supports various options to influence the execution of specific join algorithms. The following SQL procedure options are available:

Table 7.3: SQL Procedure Options

Option	Description
MAGIC=101	Influence the SQL optimizer to select the Nested Loop join algorithm.
MAGIC=102	Influence the SQL optimizer to select the Sort-Merge join algorithm.
MAGIC=103	Influence the SQL optimizer to select the Hash join algorithm.

In the next example, the PROC SQL option MAGIC=101 is specified to influence the optimizer to select a nested loop join algorithm for query execution.

SQL Code

```
proc sql magic=101;
  select c.custname,
         p1.prodnum, p1.prodname, p2.units, p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products  p1,
         purchases  p2,
         customers  c
   where p1.prodnum = p2.prodnum  AND
         p2.custnum = c.custnum
  order by c.custname, p1.prodname;
quit;
```

The log results that follow show that the SQL optimizer (or Planner) chose the nested loop (sequential loop) join algorithm when the MAGIC=101 option was specified.

SAS Log Results

```
proc sql magic=101;
  select c.custname,
         p1.prodnum, p1.prodname, p2.units, p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products  p1,
         purchases  p2,
         customers  c
```

```

        where p1.prodnum = p2.prodnum  AND
              p2.custnum = c.custnum
          order by c.custname, p1.prodname;
NOTE: PROC SQL planner chooses sequential loop join.
NOTE: PROC SQL planner chooses sequential loop join.
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time            0.08 seconds
      cpu time             0.07 seconds

```

In the next example, the PROC SQL option MAGIC=102 is specified to influence the optimizer to select a sort-merge join algorithm for query execution.

SQL Code

```

proc sql magic=102;
  select c.custname,
         p1.prodnum, p1.prodname, p2.units, p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products  p1,
         purchases  p2,
         customers  c
   where p1.prodnum = p2.prodnum  AND
         p2.custnum = c.custnum
      order by c.custname, p1.prodname;
quit;

```

The log results that follow show that the SQL optimizer (or Planner) chose the sort-merge (merge) join algorithm when the MAGIC=102 option was specified.

SAS Log Results

```

proc sql magic=102;
  select c.custname,
         p1.prodnum, p1.prodname, p2.units, p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products  p1,
         purchases  p2,
         customers  c
   where p1.prodnum = p2.prodnum  AND
         p2.custnum = c.custnum
      order by c.custname, p1.prodname;
NOTE: PROC SQL planner chooses merge join.
NOTE: PROC SQL planner chooses merge join.
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time            0.14 seconds
      cpu time             0.09 seconds

```

In the next example, the PROC SQL option MAGIC=103 is specified to influence the optimizer to select a hash join algorithm for query execution.

SQL Code

```

proc sql magic=103;
  select c.custname,
         p1.prodnum, p1.prodname, p2.units, p2.unitcost,

```

```

      p2.units * p2.unitcost as Total_Cost format=dollar12.2
  from products  p1,
       purchases p2,
       customers c
 where p1.prodnum = p2.prodnum  AND
       p2.custnum = c.custnum
   order by c.custname, p1.prodname;
quit;

```

The log results show that the SQL optimizer (or Planner) initially chose a merge join algorithm, but transformed the merge join to a hash join algorithm when the MAGIC=103 option was specified.

SAS Log Results

```

proc sql magic=103;
  select c.custname,
         p1.prodnum, p1.prodname, p2.units, p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products  p1,
         purchases p2,
         customers c
   where p1.prodnum = p2.prodnum  AND
         p2.custnum = c.custnum
   order by c.custname, p1.prodname;
NOTE: PROC SQL planner chooses merge join.
NOTE: PROC SQL planner chooses merge join.
NOTE: A merge join has been transformed to a hash join.
NOTE: A merge join has been transformed to a hash join.
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time            0.08 seconds
      cpu time             0.07 seconds

```

Cartesian Product Joins

As mentioned previously, the Cartesian product (or cross join) represents all possible combinations of rows and columns from the joined tables. To be exact, it represents the sum of the number of columns of the input tables plus the product of the number of rows of the input tables. Put another way, it represents each row from the first table matched with each possible row from the second table, and so on. For example, if you performed a join operation on one table that consists of 100,000 rows and on a second table that consists of 10,000 rows, you would get a Cartesian product that consists of 10 million rows.

Although the Cartesian product serves a very useful purpose in the relational model, it is essentially meaningless for a user to intentionally produce it as a final table. Besides being large, Cartesian products contain too much information and make it difficult, if not impossible, for the practitioner to select what is salient. It is only when you subset the Cartesian product by using a WHERE clause that your data becomes quantifiable and manageable. For more information on Cartesian product joins and examples that illustrate the results of these joins, go to the Author Page for this book at support.sas.com/lafler.

Inner Joins

As mentioned previously, inner joins can handle a maximum of 256 tables at a time, and are the most recognized and widely used type of join. They are principally used to restrict rows where the specific search condition is not met. As a result, only rows that satisfy the conditions specified in the WHERE clause are kept. This is in direct contrast with outer joins (which will be discussed in a later section).

Equijoins

The most common form of inner join, which is often referred to as an *equijoin*, uses an equal sign “=” in the WHERE clause to indicate equality between the columns in two or more tables. Suppose that you want to match products with their corresponding manufacturers so that all products from each manufacturer would be listed. An equijoin is performed to equate the manufacturer number from tables PRODUCTS and MANUFACTURERS.

SQL Code

```
PROC SQL;
  SELECT prodname, prodcost,
         manufacturers.manunum, manuname
    ①          ②
   FROM PRODUCTS, MANUFACTURERS
 WHERE products.manunum = ③
       manufacturers.manunum;
QUIT;
```

- ① The PRODUCTS table is the first table specified in the FROM clause.
- ② The MANUFACTURERS table is the second table specified in the FROM clause.
- ③ The specification of an equal sign “=” in a WHERE clause between the columns in the tables indicates an equality type of join.

Results

Product Name	Product Cost	Manufacturer Number	Manufacturer Name
Dream Machine	\$3,200.00	111	Cupid Computer
Business Machine	\$3,300.00	120	Storage Devices Inc
Analog Cell Phone	\$35.00	210	Global Comm Corp
Digital Cell Phone	\$175.00	210	Global Comm Corp
Spreadsheet Software	\$299.00	500	KPL Enterprises
Database Software	\$399.00	500	KPL Enterprises
Wordprocessor Software	\$299.00	500	KPL Enterprises
Graphics Software	\$299.00	500	KPL Enterprises

The previous example can be further qualified by adding another condition in the WHERE clause. For example, suppose that you want to display only those products from the manufacturer KPL Enterprises. The following join identifies all of the products that are manufactured by KPL Enterprises as specified in the WHERE clause (all rows that do not meet the condition of the WHERE clause are automatically excluded from the results of the join).

Note: This join assumes that you know KPL Enterprises' unique manufacturer number.

SQL Code

```
PROC SQL;
  SELECT proddname, prodcost,
         manufacturers.manunum, manuname
    FROM PRODUCTS, MANUFACTURERS
   WHERE products.manunum =
        manufacturers.manunum      ①
          AND
        products.manunum = 500;
QUIT;
```

- ① The specification of the AND logical operator in the WHERE clause indicates that both conditions must be true in order to retrieve rows from both tables.

Results

Product Name	Product Cost	Manufacturer Number	Manufacturer Name
Spreadsheet Software	\$299.00	500	KPL Enterprises
Graphics Software	\$299.00	500	KPL Enterprises
Wordprocessor Software	\$299.00	500	KPL Enterprises
Database Software	\$399.00	500	KPL Enterprises

Let's extend our knowledge of equijoins by identifying how much money is tied up with products that are manufactured by KPL Enterprises. To accomplish this, you need to do two things. First, you need to sum the product cost (PRODCOST) column across all rows that match the WHERE clause condition. Because the objective of the equijoin is to compute a total amount for products that are manufactured by KPL Enterprises, you need to prevent duplicate rows from displaying in the results. To prevent duplicate rows, you need to specify the DISTINCT keyword.

SQL Code

```
PROC SQL;
  SELECT DISTINCT SUM(prodcost) AS Total_Cost      ①
    FORMAT=DOLLAR10.2,
    manufacturers.manunum
   FROM PRODUCTS, MANUFACTURERS
  WHERE products.manunum =
    manufacturers.manunum AND
    manufacturers.manuname = 'KPL Enterprises';
QUIT;
```

① The DISTINCT keyword prevents duplicate rows from appearing in the result.

Results

Total_Cost	Manufacturer Number
\$1,296.00	500

Non-Equijoins

Another type of inner join is known as a non-equijoin. As you might guess from its name, a non-equijoin does not have an equal sign “=” specified in its WHERE clause. For example, suppose that you want to display products that are manufactured by KPL Enterprises that cost more than \$299.00. The use of the greater than “>” operator gives this type of join its name.

Note: When the SQL procedure optimizer is unable to optimize a join query by reducing the Cartesian product, a message is displayed in the SAS log that indicates that the join requires performing one or more Cartesian product joins and cannot be optimized.

SQL Code

```
PROC SQL;
  SELECT prodname, prodtype, prodcost,
         manufacturers.manunum, manufacturers.manuname
    FROM PRODUCTS, MANUFACTURERS
   WHERE manufacturers.manunum = 500 AND
         prodtype = 'Software' AND
         prodcost > 299.00;      ①
QUIT;
```

- ① The specification of the greater than “>” operator in the WHERE clause indicates a non-equijoin scenario.

SAS Log Results

```
PROC SQL;
  SELECT prodname, prodtype, prodcost,
         manufacturers.manunum, manufacturers.manuname
    FROM PRODUCTS, MANUFACTURERS
   WHERE manufacturers.manunum = 500 AND
         prodtype = 'Software' AND
         prodcost > 299.00;
NOTE: The execution of this query involves performing one or
more
Cartesian product joins that cannot be optimized.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.01 seconds
      cpu time          0.01 seconds
```

Results

Product Name	Product Type	Product Cost	Manufacturer Number	Manufacturer Name
Database Software	Software	\$399.00	500	KPL Enterprises

Reflexive or Self Joins

The final type of inner join is referred to as a reflexive join, which is also known as a self join. As its name implies, a self join makes an internal copy of a table, and then joins the copy to itself. Essentially, a join of this type joins one copy of a table to itself for the purpose of exploiting and illustrating comparisons between table values. For example, suppose that you want to compare the prices of products side-by-side by product type with the less expensive product appearing first (as shown in the first three columns of the example results).

SQL Code

```
PROC SQL;
  SELECT products.prodname, products.prodtype,
         products.prodcost,
         products_copy.prodname, products_copy.prodtype,
         products_copy.prodcost
```

```

①          ②
FROM PRODUCTS, PRODUCTS PRODUCTS_COPY
    WHERE products.prodtype =
          products_copy.prodtype AND
          products.prodcost <
          products_copy.prodcost;
③
QUIT;

```

- ① The PRODUCTS table is the primary table that is specified in the FROM clause.
- ② A copy of the PRODUCTS table called PRODUCTS_COPY is joined with the PRODUCTS table.
- ③ The WHERE clause requests the same type of products to be compared side-by-side with the less expensive product appearing first.

Results

Product Name	Product Type	Product Cost	Product Name	Product Type	Product Cost
Dream Machine	Workstation	\$3,200.00	Business Machine	Workstation	\$3,300.00
Analog Cell Phone	Phone	\$35.00	Office Phone	Phone	\$130.00
Analog Cell Phone	Phone	\$35.00	Digital Cell Phone	Phone	\$175.00
Office Phone	Phone	\$130.00	Digital Cell Phone	Phone	\$175.00
Spreadsheet Software	Software	\$299.00	Database Software	Software	\$399.00
Wordprocessor Software	Software	\$299.00	Database Software	Software	\$399.00
Graphics Software	Software	\$299.00	Database Software	Software	\$399.00

Let's look at another example. Suppose that you want to find out the names and invoice amounts where, for each customer, you list the names and invoice amounts of each customer with larger invoice amounts. The next example illustrates a useful application of a self join.

SQL Code

```

PROC SQL;
  SELECT invoice.custnum, invoice.invprice,
         invoice_copy.custnum, invoice_copy.invprice
  ①          ②
  FROM INVOICE, INVOICE INVOICE_COPY
    WHERE invoice.invprice <      ③
          invoice_copy.invprice;
QUIT;

```

- ① The INVOICE table is the primary table that is specified in the FROM clause.
- ② A copy of the INVOICE table called INVOICE_COPY is joined with the INVOICE table.
- ③ The WHERE clause produces the names of customers with larger invoice amounts.

Results

Customer Number	Invoice Unit Price	Customer Number	Invoice Unit Price
201	\$1,495.00	1301	\$1,598.00
201	\$1,495.00	501	\$9,600.00
201	\$1,495.00	401	\$23,100.00
1301	\$1,598.00	501	\$9,600.00
1301	\$1,598.00	401	\$23,100.00
101	\$245.00	201	\$1,495.00
101	\$245.00	1301	\$1,598.00
101	\$245.00	501	\$9,600.00
101	\$245.00	801	\$798.00
101	\$245.00	901	\$396.00
101	\$245.00	401	\$23,100.00
501	\$9,600.00	401	\$23,100.00
801	\$798.00	201	\$1,495.00
801	\$798.00	1301	\$1,598.00
801	\$798.00	501	\$9,600.00
801	\$798.00	401	\$23,100.00
901	\$396.00	201	\$1,495.00
901	\$396.00	1301	\$1,598.00
901	\$396.00	501	\$9,600.00
901	\$396.00	801	\$798.00
901	\$396.00	401	\$23,100.00

Using Table Aliases in Joins

Every table in a SAS library must have a unique name to reference it. Table names must conform to valid SAS naming conventions, which means that table names can have a maximum length of 32 characters and must start with a letter or underscore (see *SAS Language Reference: Concepts* for further details).

To minimize the number of keystrokes that are needed to reference the tables that are specified in a join query, you can assign an *alias* or temporary table name reference to each table. When assigned, these arbitrary aliases provide a shortcut method to the tables themselves and are in effect for the duration of the join query *but no longer*. In the next example, the table alias “P” is assigned to the PRODUCTS table and the alias “M” is assigned to the MANUFACTURERS table in the FROM clause. Table name references in the SELECT statement and WHERE clause are made easier as well.

SQL Code

```
PROC SQL;
  SELECT prodnum, prodname, prodtype, M.manunum
    FROM PRODUCTS P, MANUFACTURERS M      ①
      WHERE P.manunum = M.manunum AND
            M.manuname = 'KPL Enterprises';
QUIT;
```

- ① The assignment of the table alias “P” and the table alias “M” in the FROM clause provides a shortcut method to refer to the longer table names PRODUCTS and MANUFACTURERS.

Results

Product Number	Product Name	Product Type	Manufacturer Number
5001	Spreadsheet Software	Software	500
5002	Database Software	Software	500
5003	Wordprocessor Software	Software	500
5004	Graphics Software	Software	500

Performing Computations in Joins

Join queries, as with simpler queries, can take full advantage of the power of the SQL procedure. Logical and arithmetic operators, predicates, and summary functions are all available for you to use. The join query is an essential component because stored information is not always available in the form that you need.

PROC SQL provides the ability to perform basic arithmetic operations such as addition, subtraction, multiplication, and division with columns that contain numeric values. Essentially, this enables any query to perform column addition, subtraction, multiplication, and division. Suppose that you want to compute the sales tax of 7.75% for all manufactured products that are sold in the state of California. In the next example, the SELECT statement shows the California sales tax (using the product cost column and the fixed sales tax percentage) computation assigns a column alias to the result column as well as a format and label to enhance the readability of the results.

SQL Code

```
PROC SQL;
  SELECT prodname, prodtype, prodcost,
         prodcost * .0775 AS SalesTax      ①
         FORMAT=dollar10.2 LABEL='California Sales Tax'
    FROM PRODUCTS P, MANUFACTURERS M
      WHERE P.manunum = M.manunum AND
            M.manustat = 'CA';
QUIT;
```

- ① The ability to perform basic arithmetic operations in a SELECT statement as well as assign a column alias to the result is part of the SQL ANSI standard.

Results

Product Name	Product Type	Product Cost	California Sales Tax
Business Machine	Workstation	\$3,300.00	\$255.75
Analog Cell Phone	Phone	\$35.00	\$2.71
Digital Cell Phone	Phone	\$175.00	\$13.56
Spreadsheet Software	Software	\$299.00	\$23.17
Database Software	Software	\$399.00	\$30.92
Wordprocessor Software	Software	\$299.00	\$23.17
Graphics Software	Software	\$299.00	\$23.17

Joins with Three Tables

Up to this point, our examples have been limited to two-table joins. But what if more information is needed than the two tables can provide? To extract the required information, access to a third table might be necessary. A join with three tables is a fairly simple extension of a two-table join.

As before, each joinable column must possess the same column attributes and contain the same type of information. In addition to listing all of the required tables in the FROM clause, the WHERE clause would need to include any and all restrictions in order to subset only the rows desired. For example, suppose that you want to display only those products along with their invoice quantity that appear in the INVOICE table for the manufacturer KPL Enterprises (manunum=500).

SQL Code

```
PROC SQL;
  SELECT P.prodname,
         P.prodcost,
         M.manuname,
         I.invqty
    FROM PRODUCTS  P,
         MANUFACTURERS  M,
         INVOICE  I
   WHERE P.manunum = M.manunum AND
         P.prodnum = I.prodnum AND
         M.manunum = 500;
QUIT;
```

Results

Product Name	Product Cost	Manufacturer Name	Invoice Quantity - Units Sold
Spreadsheet Software	\$299.00	KPL Enterprises	5
Database Software	\$399.00	KPL Enterprises	2

Let's examine the construction of the WHERE clause for this three-way join a bit further. The column that contains the manufacturer number from the PRODUCTS, MANUFACTURERS, and INVOICE tables is joined by using an AND logical operator in the WHERE clause. Additionally, the WHERE clause restricts the resulting table to only product invoices for manufacturer (manunum=500). In the next example, a three-way join lists the product names and costs, along with the customer who bought each product.

SQL Code

```
PROC SQL;
  SELECT P.prodname,
         P.prodcost,
         C.custname,
         I.invprice
    FROM PRODUCTS  P,
         INVOICE    I,
         CUSTOMERS C
   WHERE P.prodnum = I.prodnum AND
         I.custnum = C.custnum;
QUIT;
```

Results

Product Name	Product Cost	Customer Name	Invoice Unit Price
Analog Cell Phone	\$35.00	La Mesa Computer Land	\$245.00
Spreadsheet Software	\$299.00	Vista Tech Center	\$1,495.00
Business Machine	\$3,300.00	La Jolla Computing	\$23,100.00
Dream Machine	\$3,200.00	Alpine Technical Center	\$9,600.00
Database Software	\$399.00	Jamul Hardware & Software	\$798.00

Joins with More Than Three Tables

Occasionally, information needs to be extracted from four, five, or more tables (up to a maximum of 256 tables). Joins of four or more tables can be constructed just like those accessing two or three tables. The only difference is the number of table references in the FROM clause and the level of complexity in the WHERE clause to restrict what rows are kept. Suppose that you want to know, based on invoices, the number of products that were ordered before September 1, 2000. One way to find this information is to perform a join with four tables.

SQL Code

```
PROC SQL;
  SELECT sum(inventory.invenqty)
    AS Products_Ordered_Before_09012000
  FROM PRODUCTS,
       INVOICE,
       CUSTOMERS,
       INVENTORY
 WHERE inventory.orddate < mdy(09,01,00) AND
       products.prodnum = invoice.prodnum AND
       invoice.custnum = customers.custnum AND
       invoice.prodnum = inventory.prodnum;
QUIT;
```

Results

Products_Ordered_Before_09012000
8

If you were wondering whether this result could have been derived another way, you would be correct. You could also determine, based on invoices, the number of products that were ordered before September 1, 2000, with the following two-way join code. As with this example, there is often more than one way to construct a join to extract the information that you want.

SQL Code

```
PROC SQL;
  SELECT sum(inventory.invenqty)
    AS Products_Ordered_Before_09012000
  FROM INVOICE I,
       INVENTORY I2
 WHERE inventory.orddate < mdy(09,01,00) AND
       invoice.prodnum = inventory.prodnum;
QUIT;
```

Results

Products_Ordered_Before_09012000
8

To expand your understanding of joins with more than three tables, the following example illustrates a four-table join. Suppose that you want to know which products are being purchased and who is purchasing them. The next example shows a four-way inner join that combines data from the MANUFACTURERS, PRODUCTS, INVOICE, and CUSTOMERS tables.

SQL Code

```

PROC SQL;
  SELECT products.prodname,
         products.prodtype,
         customers.custname,
         manufacturers.manuname
    FROM MANUFACTURERS,
         PRODUCTS,
         INVOICE,
         CUSTOMERS
   WHERE manufacturers.manunum = products.manunum AND
         manufacturers.manunum = invoice.manunum AND
         products.prodnum      = invoice.prodnum AND
         invoice.custnum        = customers.custnum;
QUIT;

```

Results

Product Name	Product Type	Customer Name	Manufacturer Name
Analog Cell Phone	Phone	La Mesa Computer Land	Global Comm Corp
Spreadsheet Software	Software	Vista Tech Center	KPL Enterprises
Dream Machine	Workstation	Alpine Technical Center	Cupid Computer
Database Software	Software	Jamul Hardware & Software	KPL Enterprises

Outer Joins

As the previous examples in this chapter have shown, an inner join disregards any rows where the search condition is not met. This differs significantly from the way an *outer join* groups tables. In contrast with an inner join, an outer join keeps rows that match the ON (search) condition, as well as preserving some or all of the unmatched data from one or both of the tables. Essentially, an outer join retains rows from one table even when they do not match rows in the second table. This distinction is critical because this is what truly differentiates an outer join from an inner join.

Next, an outer join is capable of processing a maximum of two tables at a time, whereas (under the SAS implementation) an inner join is able to process a maximum of 256 tables.

Another difference has to do with how you specify outer join syntax. The comma that is used to designate or delimit one table from the other table in the FROM clause of inner joins is replaced with one of the following keywords: LEFT JOIN, RIGHT JOIN, or FULL JOIN in outer joins. Additionally, the WHERE clause expression that is used to restrict what rows are kept in the result table is replaced with the ON keyword.

Finally, an outer join is considered to be an asymmetric join (Lorie and Daudenarde, 1991, 87). Unlike inner joins, an outer join does not select rows proportionally from its parts or tables.

Left Outer Joins

Let's look at how a left join is applied in a real-world situation. Suppose that you want to see a list of all manufacturers, their city locations, their manufacturer numbers, their product types, and their product costs (if available) without leaving out those manufacturers that do not have products yet. This means that the MANUFACTURERS table (left table) acts as the master table having its rows preserved while the PRODUCTS table (right table) acts as the contributing table (subordinate table). The following left outer join example effectively retains those matched rows from both tables as well as retaining those rows from the left table that have no match in the right table.

SQL Code

```
PROC SQL;
  SELECT manuname, manucity, manufacturers.manunum,
         products.prodtype, products.prodcost
    FROM MANUFACTURERS LEFT JOIN PRODUCTS      ❶
      ON manufacturers.manunum =          ❷
         products.manunum;
QUIT;
```

- ❶ The LEFT JOIN specification preserves all of the rows in the left table (MANUFACTURERS) even when there are no matching rows in the right table (PRODUCTS).
- ❷ The ON clause acts as a WHERE clause to select the desired rows in the join results.

As the results from the left outer join illustrate, the rows in the left (MANUFACTURERS) table that match rows in the right (PRODUCTS) table are included in the result table. As a result, eight rows match as evidenced by the value assigned to product type and product cost. Additionally, two rows from the left table that do not match rows in the right table (based on the search condition) are also retained (bolded). Therefore, each row from the MANUFACTURERS table that does not have a matching value in the PRODUCTS table is added to the resulting virtual table, accompanied by null values in the product type and product cost columns.

Results

Manufacturer Name	Manufacturer City	Manufacturer Number	Product Type	Product Cost
Cupid Computer	Houston	111	Workstation	\$3,200.00
Storage Devices Inc	San Mateo	120	Workstation	\$3,300.00
Global Comm Corp	San Diego	210	Phone	\$175.00
Global Comm Corp	San Diego	210	Phone	\$35.00
KPL Enterprises	San Diego	500	Software	\$299.00
KPL Enterprises	San Diego	500	Software	\$299.00
KPL Enterprises	San Diego	500	Software	\$299.00
KPL Enterprises	San Diego	500	Software	\$399.00
World Internet Corp	Miami	600		.
San Diego PC Planet	San Diego	700		.

Specifying a WHERE Clause

To provide greater subsetting capabilities as well as added flexibility, the SQL procedure also permits the specification of an optional WHERE clause in addition to an ON clause when constructing outer joins. The ability to specify a WHERE clause in conjunction with an ON clause permits greater control over the subsetting of rows. An example will help illustrate how a WHERE clause is used in an outer join. Suppose that you want to limit the results from the previous left outer join to only those products that cost less than \$300.00. In this example, the left outer join syntax uses a WHERE clause to subset row results to nonmissing products that cost less than \$300.00.

SQL Code

```
PROC SQL;
  SELECT manuname, manucity, manufacturers.manunum,
         products.prodtype, products.prodcost
    FROM MANUFACTURERS LEFT JOIN PRODUCTS
      ON manufacturers.manunum =
         products.manunum
     WHERE prodcost < 300 AND      ①
           prodcost NE .;
QUIT;
```

- ① The optional WHERE clause that is specified in addition to an ON clause in an outer join further subsets the joined results.

Results

Manufacturer Name	Manufacturer City	Manufacturer Number	Product Type	Product Cost
Global Comm Corp	San Diego	210	Phone	\$175.00
Global Comm Corp	San Diego	210	Phone	\$35.00
KPL Enterprises	San Diego	500	Software	\$299.00
KPL Enterprises	San Diego	500	Software	\$299.00
KPL Enterprises	San Diego	500	Software	\$299.00

Specifying Aggregate Functions

Suppose that you need to produce a monthly report that consists of a total invoice amount by manufacturer. An aggregate function can be specified with outer join syntax to perform a group computation using a GROUP BY clause. In the next example, a left join computes the total invoice amount for each manufacturer with a SUM function and GROUP BY clause.

SQL Code

```
PROC SQL;
  SELECT manuname,
    SUM(invoice.invprice) AS Total_Invoice_Amt      ①
    FORMAT=DOLLAR10.2
  FROM MANUFACTURERS LEFT JOIN INVOICE
  ON manufacturers.manunum =
    invoice.manunum
  GROUP BY MANUNAME;      ②
QUIT;
```

- ① The SUM function computes the total invoice amount for each manufacturer.
- ② The GROUP BY clause groups all of the rows associated with a manufacturer into a single row.

The results show that manufacturers with no activity have a null or missing value in the aggregated Total_Invoice_Amt column.

Results

Manufacturer Name	Total_Invoice_Amt
Cupid Computer	\$9,600.00
Global Comm Corp	\$245.00
KPL Enterprises	\$25,789.00
San Diego PC Planet	.
Storage Devices Inc	.
World Internet Corp	\$1,598.00

Right Outer Joins

Right joins are similar to left joins, except that the rows in the right (second) table are preserved. Consequently, the results contain the rows of the symmetric join plus a row for each unmatched row in the right table. Nulls are automatically substituted for values from the left table. Suppose that you want to see all manufacturers and their respective products. In the next example, a simple report that contains products, product type, manufacturer number, and manufacturer name is produced from the PRODUCTS and MANUFACTURERS tables using a right outer join construct.

SQL Code

```
PROC SQL;
  SELECT prodname, prodtype,
         products.manunum, manuname
    FROM PRODUCTS RIGHT JOIN MANUFACTURERS ①
      ON products.manunum =
        manufacturers.manunum;
QUIT;
```

- ① The RIGHT JOIN specification preserves all of the rows in the right table (MANUFACTURERS) even when there are no matching rows in the left table (PRODUCTS).

The results show that manufacturers that appear in the MANUFACTURERS table with no products listed in the PRODUCTS table have null or missing values in the Product Name, Product Type, and Manufacturer Number columns.

Note: To remove rows with missing values in the results, a WHERE clause could be specified.

Results

Product Name	Product Type	Manufacturer Number	Manufacturer Name
Dream Machine	Workstation	111	Cupid Computer
Business Machine	Workstation	120	Storage Devices Inc
Digital Cell Phone	Phone	210	Global Comm Corp
Analog Cell Phone	Phone	210	Global Comm Corp
Spreadsheet Software	Software	500	KPL Enterprises
Graphics Software	Software	500	KPL Enterprises
Wordprocessor Software	Software	500	KPL Enterprises
Database Software	Software	500	KPL Enterprises
		.	World Internet Corp
		.	San Diego PC Planet

Full Outer Joins

Full outer joins combine the power of left and right joins by preserving rows from both the left and right tables. Although a full join is not used as frequently as left join or right join constructs, it can be useful when information from both tables is missing. In the next example, a full outer join is specified to produce a report that contains manufacturers with no products and products with no known manufacturers.

SQL Code

```
PROC SQL;
  SELECT prodname, prodtype,
         products.manunum, manuname
    FROM PRODUCTS FULL JOIN MANUFACTURERS      ①
      ON products.manunum =
          manufacturers.manunum;
QUIT;
```

- ① The full join specification preserves all of the rows in the left table (PRODUCTS) as well as all of the rows in the right table (MANUFACTURERS) even when there are no matching rows.

Results

Product Name	Product Type	Manufacturer Number	Manufacturer Name
Dream Machine	Workstation	111	Cupid Computer
Business Machine	Workstation	120	Storage Devices Inc
Travel Laptop	Laptop	170	
Digital Cell Phone	Phone	210	Global Comm Corp
Analog Cell Phone	Phone	210	Global Comm Corp
Office Phone	Phone	220	
Spreadsheet Software	Software	500	KPL Enterprises
Graphics Software	Software	500	KPL Enterprises
Wordprocessor Software	Software	500	KPL Enterprises
Database Software	Software	500	KPL Enterprises
		.	World Internet Corp
		.	San Diego PC Planet

Subqueries

Now that you have seen how two or more tables can be combined in a join query, turn your attention to another type of complex query known as a subquery. A subquery is a query expression that is nested within another query expression. Its purpose is to have the inner query produce a single value or multiple values that can then be passed into the outer query for processing. You achieve this by embedding a SELECT statement inside a WHERE clause of an outer query's SELECT statement, INSERT statement, DELETE statement, or HAVING clause.

Note: You should avoid nesting more than two subqueries deep because of the conceptual and processing complexities this introduces.

The typical subquery consists of a (inner) query combined inside the predicate of another (outer or main) query. When processed, the inner query passes a Boolean value to the outer query consisting of either *True* if it returns a minimum of one row or *False* if no rows are returned by the subquery. The results of the inner query are stored in a temporary results table and used as input to the main query. Our exploration of subqueries will involve using them with comparison operators, the IN predicate, and the ANY and ALL keywords, and will conclude with a look at a special type of subquery called a *correlated subquery*.

Alternate Approaches to Subqueries

A subquery is a very useful construct, especially when information from multiple tables needs to be interrelated. Unfortunately, a subquery is not always easy to construct and might be even more difficult to understand. So before constructing every table relation with a subquery, consider your options carefully.

When all of the information is available in a single table, a simple query is probably all that needs to be constructed. Suppose that you want to produce a report that consists of the invoice information for Global Comm Corp. Let's further assume that you know the specific manufacturer number for Global Comm Corp. Knowing this means that you don't have to go into the MANUFACTURERS table to find it. In the next example, a simple query is constructed to retrieve all of the invoice information from the INVOICE table.

Simple Query

```
PROC SQL;
  SELECT *
    FROM INVOICE
   WHERE manunum = 210;
QUIT;
```

Results

Invoice Number	Manufacturer Number	Customer Number	Invoice Quantity - Units Sold	Invoice Unit Price	Product Number
1003	210	101	7	\$245.00	2101

But, what if all the information is not in a single table? And what if the manufacturer number for Global Comm Corp is not known? As shown previously, a join can be constructed just as easily as a subquery. Some users prefer joins to subqueries because joins can be easier to understand and maintain. In fact, a join frequently performs better than a subquery. In the next example, the manufacturer number for Global Comm Corp is not known. Consequently, a simple inner join is needed to retrieve all related rows from the MANUFACTURERS and INVOICE tables for Global Comm Corp.

Simple Join

```
PROC SQL;
  SELECT M.manunum, M.manuname, I.invnum,
         I.invqty, I.invprice
    FROM MANUFACTURERS M, INVOICE I
   WHERE M.manunum = I.manunum AND
         M.manuname = 'Global Comm Corp';
QUIT;
```

Results

Manufacturer Number	Manufacturer Name	Invoice Number	Invoice Quantity - Units Sold	Invoice Unit Price
210	Global Comm Corp	1003	7	\$245.00

Passing a Single Value with a Subquery

Let's see how a subquery could be constructed to provide the same results as with the join. As before, suppose that you want to pull all of the invoices for the manufacturer Global Comm Corp but know only the manufacturer name (or at least part of the name), but not the manufacturer number (MANUNUM). The following subquery uses an = (equal sign) in its outer query WHERE clause to accomplish this.

Because the manufacturer number is not known, a subquery is constructed to first search for the manufacturer number in the MANUFACTURERS table. Actually, the subquery approach is more versatile than the previous query approach, because it does not require a unique manufacturer number, which is often more difficult to remember than a manufacturer names. It also enables quick searches even if the manufacturer number changes for a given manufacturer.

When the entire query is executed, SQL first evaluates the inner query (or subquery) within the outer query's WHERE clause. It executes the inner query the same way as if it were a standalone query. It searches the MANUFACTURERS table for any row where the manufacturer name equals the character string Global Comm Corp, and then pulls the MANUNUM values for this row. SQL then substitutes the derived MANUNUM value of 210 from the inner query inside the predicate of the main query (outer query). As a result of this substitution, the SQL query looks identical to the query mentioned previously.

SQL Code

```
PROC SQL FEEDBACK;
  SELECT invnum, INVOICE.manunum, custnum, invqty, invprice, prodnum
    FROM INVOICE,
        (SELECT manunum          ①
         FROM MANUFACTURERS
           WHERE manuname = 'Global Comm Corp')
      WHERE INVOICE.manunum = MANUFACTURERS.manunum;
QUIT;
```

Result of Inner Query

```
PROC SQL;
  SELECT *
    FROM INVOICE
      WHERE manunum = 210;          ②
QUIT;
```

- ① PROC SQL evaluates the inner query within the outer query's WHERE clause to search for the manufacturer number for manufacturer Global Comm Corp.
- ② The resulting query after substituting the derived manufacturer number value from the inner query evaluates to a single value, and is then executed as the main (outer) query.

Results

Invoice Number	Manufacturer Number	Customer Number	Invoice Quantity - Units Sold	Invoice Unit Price	Product Number
1003	210	101	7	\$245.00	2101

Let's look at another subquery. Suppose that you want to retrieve the invoice from the INVOICE table for the manufacturer that manufactures the Dream Machine workstation. The following subquery (inner query) extracts the product number (PRODNUM) that is associated with the Dream Machine, and passes the single value to the outer query for processing.

SQL Code

```
PROC SQL FEEDBACK;
  SELECT invnum, manunum, custnum, invqty, invprice,
         INVOICE.prodnum
    FROM INVOICE
      (SELECT prodnum          ①
       FROM PRODUCTS
      WHERE prodname LIKE 'Dream%');
QUIT;
```

Result of Inner Query

```
PROC SQL;
  SELECT *
    FROM INVOICE
   WHERE prodnum = 1110;          ②
QUIT;
```

- ① PROC SQL evaluates the inner query within the outer query's WHERE clause to search for the product number for the Dream Machine product.
- ② The resulting inner query after substituting the derived product number value evaluates to a single value, and is then executed as the main (outer) query.

Results

Invoice Number	Manufacturer Number	Customer Number	Invoice Quantity - Units Sold	Invoice Unit Price	Product Number
1004	111	501	3	\$9,600.00	1110

It is fortunate that the subquery in the previous example passed only one row or value to the main (outer) query. Had it returned more than one value from the PRODUCTS table, it would have made it impossible for the SQL procedure to evaluate the condition as true or false and would have produced an error in the outer query. Let's look at another example where more than one value is returned by the subquery.

In the next example, more than one row is returned by the inner query making it impossible for the main query to evaluate as true or false. As a result, an error is produced and the

subquery does not execute. In general, it is best to avoid using the = (equal sign) and other comparison operators (<, >, <=, >=, and <>) in a subquery expression unless you know in advance that the result of the subquery is a table with a single row of data (although it might not always be possible to know this beforehand). In the “Passing More Than One Row with a Subquery” section in this chapter, you will see this problem alleviated by using the IN predicate.

SQL Code

```
PROC SQL;
  SELECT *
    FROM INVOICE
   WHERE manunum =
     (SELECT manunum
      FROM MANUFACTURERS
     WHERE UPCASE(manucity) LIKE 'SAN DIEGO%');
QUIT;
```

SAS Log Result

```
PROC SQL;
  SELECT *
    FROM INVOICE
   WHERE manunum =
     (SELECT manunum
      FROM MANUFACTURERS
     WHERE UPCASE(manucity) LIKE 'SAN DIEGO%');
ERROR: Subquery evaluated to more than one row.
      QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

Let’s look at another subquery example that uses the comparison operator < (less than). A summary function specified in an inner query forces a single row to result. In the next example, the subquery uses the AVG summary (aggregate) function to determine which products (based on their invoice quantities) were purchased in lower quantities than the average product purchase.

SQL Code

```
PROC SQL;
  SELECT prodnum, invnum, invqty, invprice
    FROM INVOICE
   WHERE invqty <
     (SELECT AVG(invqty)      ①
      FROM INVOICE);
QUIT;
```

Result of Inner Query

```
PROC SQL;
  SELECT prodnum, invnum, invqty, invprice
    FROM INVOICE
   WHERE invqty < 4.285714;      ②
QUIT;
```

- ① PROC SQL evaluates the inner query within the outer query's WHERE clause to produce an average invoice quantity.
- ② The resulting inner query passes the derived average invoice quantity of 4.285714 as a single value to the main (outer) query for execution.

Results

Product Number	Invoice Number	Invoice Quantity - Units Sold	Invoice Unit Price
6001	1002	2	\$1,598.00
1110	1004	3	\$9,600.00
5002	1005	2	\$798.00
6000	1006	4	\$396.00

Passing More Than One Row with a Subquery

PROC SQL does not permit a subquery to select more than one column. To prevent this problem, which is associated with passing more than one value to the main (outer) query, you can specify the IN predicate in a subquery. Similar to the IN operator in the DATA step, the IN predicate permits the SQL procedure to pass multiple row values from the (inner) subquery to the main (outer) query without producing an error.

The next example shows how multiple row values are passed from the subquery to the main (outer) query using the IN predicate for San Diego manufacturers.

SQL Code

```
PROC SQL;
  SELECT *
    FROM INVOICE
   WHERE manunum IN      ①
        (SELECT manunum
          FROM MANUFACTURERS
         WHERE UPCASE(manucity) LIKE 'SAN DIEGO%');  ②
QUIT;
```

Result of Inner Query

```
PROC SQL;
  SELECT prodnum, invnum, invqty, invprice
  FROM INVOICE
  WHERE manunum IN (210, 500, 700);      ❸
QUIT;
```

- ❶ PROC SQL's IN predicate is specified in the outer query to process a list of values that are passed from the inner query.
- ❷ PROC SQL evaluates the inner query within the outer query's WHERE clause to produce a list of manufacturer numbers for San Diego manufacturers.
- ❸ The resulting inner query passes multiple row values to the main (outer) query for execution.

Results

Invoice Number	Manufacturer Number	Customer Number	Invoice Quantity - Units Sold	Invoice Unit Price	Product Number
1001	500	201	5	\$1,495.00	5001
1003	210	101	7	\$245.00	2101
1005	500	801	2	\$798.00	5002
1006	500	901	4	\$396.00	6000
1007	500	401	7	\$23,100.00	1200

Comparing a Set of Values

A subquery can have multiple values returned for a single column to the outer query. But there are special keywords that permit comparison operators to be used in subqueries to process multiple values. The special keywords ANY and ALL can be used to compare a set of values returned by a subquery. Let's see how these keywords work.

Suppose that you want to view the products whose inventory quantity is greater than or equal to the lowest average inventory quantity. The following example illustrates a subquery with the ANY keyword specified in the WHERE clause of the main query expression. When ANY is specified, the entire WHERE clause is true if the subquery returns at least one value.

SQL Code

```
PROC SQL;
  SELECT manunum, prodnum, invqty, invprice
  FROM INVOICE
  WHERE invprice GE ANY      ❶
    (SELECT invprice
     FROM INVOICE
     WHERE prodnum IN (5001,5002)); ❷
QUIT;
```

Result of Inner Query

```
PROC SQL;
  SELECT manunum, prodnum, invqty, invprice
    FROM INVOICE
   WHERE invprice > ANY ($1,495.,$798.); ③
QUIT;
```

- ① PROC SQL retrieves any invoices from the outer query where the invoice price is greater than or equal to the row values that are passed from the inner query.
- ② The WHERE clause of the inner query retrieves any invoice prices for product numbers 5001 and 5002 and passes them to the outer query.
- ③ The resulting inner query passes multiple row values to the main (outer) query for execution.

Results

Manufacturer Number	Product Number	Invoice Quantity - Units Sold	Invoice Unit Price
500	5001	5	\$1,495.00
600	6001	2	\$1,598.00
111	1110	3	\$9,600.00
500	5002	2	\$798.00
500	1200	7	\$23,100.00

The ALL keyword works very differently from the ANY keyword. When you specify ALL before a subquery expression, the subquery is true only if the comparison is true for values that are returned by the subquery. For example, suppose that you want to view the products whose inventory quantity is less than the average inventory quantity?

SQL Code

```
PROC SQL;
  SELECT manunum, prodnum, invqty, invprice
    FROM INVOICE
   WHERE invprice < ALL ①
     (SELECT invprice
      FROM INVOICE
     WHERE prodnum IN (5001,5002)); ②
QUIT;
```

Result of Inner Query

```
PROC SQL;
  SELECT manunum, prodnum, invqty, invprice
  FROM INVOICE
  WHERE invprice < ALL ($1,495.,$798.); ⑧
QUIT;
```

- ① PROC SQL retrieves all invoices from the outer query where the invoice price is less than the row values that are passed from the inner query.
- ② The WHERE clause of the inner query retrieves all invoice prices for product numbers 5001 and 5002 and passes them to the outer query.
- ③ The resulting inner query passes multiple row values to the main (outer) query for execution.

Results

Manufacturer Number	Product Number	Invoice Quantity - Units Sold	Invoice Unit Price
210	2101	7	\$245.00
500	6000	4	\$396.00

Correlated Subqueries

In the subquery examples shown previously, the subquery (inner query) operates independently from the main (outer) query. Essentially, the subquery's results are evaluated and used as input to the main (outer) query. Although this is a common way that subqueries execute, it is not the only way. SQL also permits a subquery to accept one or more values from its outer query. Once the subquery executes, the results are then passed to the outer query. Subqueries of this variety are called *correlated subqueries*. The ability to construct subqueries in this manner provides a powerful extension to SQL.

The difference between the subqueries discussed previously and correlated subqueries is in the way the WHERE clause is constructed. Correlated subqueries relate a column in the subquery with a column in the outer query to determine the rows that match or in certain cases don't match the expression. Suppose, for example, that you want to view products in the PRODUCTS table that do not appear in the INVOICE table. One way to do this is to construct a correlated subquery.

In the next example, the subquery compares the product number column in the PRODUCTS table with the product number column in the INVOICE table. If at least one match is found (the product appears in both the PRODUCTS and INVOICE tables), then the resulting table from the subquery will not be empty, and the NOT EXISTS condition will be false. However, if no matches are found, then the subquery returns an empty table that results in the NOT EXISTS condition being true, which causes the product number, product name, and product type of the current row in the main (outer) query to be selected.

SQL Code

```

PROC SQL;
  SELECT prodnum, prodname, prodtype
  FROM PRODUCTS
  WHERE NOT EXISTS      ①
    (SELECT *
     FROM INVOICE
     WHERE PRODUCTS.prodnum = INVOICE.prodnum);      ②
QUIT;

```

- ① The (inner) subquery receives its value(s) from the main (outer) query. With the value(s), the subquery runs and passes the results back to the main query where the WHERE clause and the NOT EXISTS condition are processed.
- ② The inner query selects matching product and invoice information and passes it to the outer query.

Results

Product Number	Product Name	Product Type
1700	Travel Laptop	Laptop
2102	Digital Cell Phone	Phone
2200	Office Phone	Phone
5003	Wordprocessor Software	Software
5004	Graphics Software	Software

Correlated subqueries are useful for placing restrictions on the results of an entire query with a HAVING clause (or, when combined with a GROUP BY clause, of an entire group).

Suppose that you want to know which manufacturers have more than one invoiced product.

In the next example, the subquery compares the manufacturer number in the PRODUCTS table with the manufacturer number in the INVOICE table. A HAVING clause and a COUNT function are specified to select all manufacturers with two or more invoices. Because an aggregate (summary) function is used in an optional HAVING clause, a GROUP BY clause is not needed to select the manufacturers with two or more invoices. An EXISTS condition is specified in the outer query's WHERE clause to capture only those manufacturers that match the subquery.

SQL Code

```
PROC SQL;
  SELECT prodnum, prodname, prodtype
  FROM PRODUCTS
  WHERE EXISTS      ①
    (SELECT *
     FROM INVOICE
     WHERE PRODUCTS.manunum = INVOICE.manunum
     HAVING COUNT(*) > 1);  ②
QUIT;
```

- ① The (inner) subquery receives its value(s) from the main (outer) query. With the value(s), the subquery runs and passes the results back to the main query where the WHERE clause and the EXISTS condition are processed.
- ② The inner query specifies a HAVING clause in order to subset manufacturers with two or more invoices.

Results

Product Number	Product Name	Product Type
5001	Spreadsheet Software	Software
5002	Database Software	Software
5003	Wordprocessor Software	Software
5004	Graphics Software	Software

Set Operations

Now that you have seen how tables are combined with join queries and subqueries, let's look at another type of complex query. The SQL procedure provides users with several table operators: INTERSECT, UNION, OUTER UNION, and EXCEPT, which are commonly referred to as *set operators*. In contrast to joins and subqueries where query results are combined horizontally, the purpose of each set operator is to combine or concatenate query results vertically. Essentially, set operators construct compound queries by combining the result sets of two or more queries.

Rules for Set Operators

Set operators adhere to basic rules of operation.

1. If a SELECT statement consists of more than one set operator, set operators will be applied in the order specified.
2. By default, duplicate rows are eliminated from the results.
3. To allow duplicates, the ALL option must be specified with a set operator.
4. Arguments are evaluated from left to right.
5. Set operators can be used in

- a. Queries
- b. Subqueries
- c. Derived tables
- d. View definitions
- e. INSERT with SELECT clause

Set Operators and Precedence

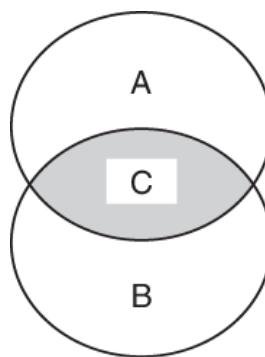
Set operators adhere to an order of precedence. The following precedence rules apply:

1. When more than one set operator is specified, each is applied in the order specified:
 - a. Top to bottom
 - b. Left to right
2. The default order of precedence for processing set operators follows:
 - a. INTERSECT
 - b. UNION and/or EXCEPT
 - c. When parentheses are specified, the default order of precedence can be altered.

Accessing Rows from the Intersection of Two Queries

The INTERSECT operator creates query results that consist of all the unique rows from the intersection of the two queries. Put another way, the intersection of two queries (A and B) is represented by C, which indicates that the rows that are produced occur in both A and in B. As Figure 7.1 shows, the intersection of both queries is represented in the shaded area (C).

Figure 7.1: Intersection of Two Queries



To see all products that cost less than \$300.00 and product types classified as “phone”, you could construct a simple query with a WHERE clause or specify the intersection of two separate queries. The next example illustrates a simple query that specifies a WHERE clause to display phones that cost less than \$300.00.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE prodcost < 300.00 AND
         prodtype = 'Phone';
QUIT;
```

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00

The INTERSECT approach can be constructed to produce the same results as in the previous example. The INTERSECT process assumes that the tables in each query are structurally identical to each other. It overlays the columns from both queries based on position in the SELECT statement. Should you attempt to intersect two queries with different table structures, the process might fail due to differing column types, or the results might contain data integrity issues.

The most significant distinction between the two approaches, and one that might affect large table processing, is that the first query example (using the AND operator) takes less time to process: 0.05 seconds versus 0.17 seconds for the second approach (using the INTERSECT operator). The next example shows how the INTERSECT operator achieves the same result less efficiently.

SQL Code

```
PROC SQL;
  SELECT *      ①
    FROM PRODUCTS
   WHERE prodcost < 300.00
INTERSECT      ②
  SELECT *      ①
    FROM PRODUCTS
   WHERE prodtype = "Phone";
QUIT;
```

- ① It is assumed that the tables in both queries are structurally identical because the wildcard character “*” is specified in the SELECT statement.
- ② The INTERSECT operator produces rows that are common to both queries.

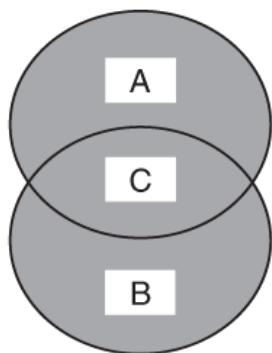
Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00

Accessing Rows from the Combination of Two Queries

The UNION operator preserves all of the unique rows from the combination of queries. The result is the same as if an OR operator is used to combine the results of each query. Put another way, the union of two queries (A and B) represents rows in A or in B or in both A and B. As illustrated in Figure 7.2, the union represents the entire shaded area (A, B, and C).

Figure 7.2: Union of Two Queries



The UNION operator automatically eliminates duplicate rows from the results, unless the ALL keyword is specified as part of the UNION operator. The column names assigned to the results are derived from the names in the first query.

In order for the union of two or more queries to be successful, each query must specify the same number of columns of the same or compatible types. Type compatibility means that column attributes are defined the same way. Because column names and attributes are derived from the first table, data types must be of the same type. The data types of the result columns are derived from the source table(s).

To see all products that cost less than \$300.00 or products that are classified as a workstation, you have a choice between using OR as shown in the following query or UNION as shown in the next query. As illustrated in the output from both queries, the results are identical no matter which query is used.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE prodcost < 300.00 OR
         prodtype = "Workstation";
  QUIT;
```

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00
5001	Spreadsheet Software	500	Software	\$299.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00

In the next example, the UNION operator is specified to combine the results of both queries.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE prodcost < 300.00
UNION          ①
  SELECT *
    FROM PRODUCTS
   WHERE prodtype = 'Workstation';
  QUIT;
```

- ① The UNION operator combines the results of two queries.

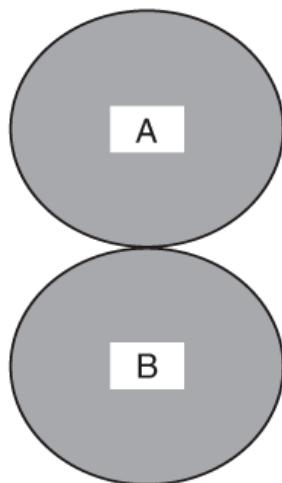
Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00
5001	Spreadsheet Software	500	Software	\$299.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00

Concatenating Rows from Two Queries

The OUTER UNION operator concatenates the results of two queries. As with a DATA step or PROC APPEND concatenation, the results consist of rows that are combined vertically. Put another way, the outer union of two queries (A and B) represents all rows in both A and B with no overlap. As illustrated in Figure 7.3, the outer union represents the entire shaded area (A and B).

Figure 7.3: Outer Union of Two Queries



The next example concatenates the results of two queries. As illustrated in the results, the rows from both queries are concatenated.

SQL Code

```
PROC SQL;
  SELECT prodnum, prodname, prodtype, prodcost
    FROM PRODUCTS
  OUTER UNION      ①
  SELECT prodnum, prodname, prodtype, prodcost
    FROM PRODUCTS;
QUIT;
```

- ① The OUTER UNION operator concatenates the results of both queries.

Results

Product Number	Product Name	Product Type	Product Cost	Product Number	Product Name	Product Type	Product Cost
1110	Dream Machine	Workstation	\$3,200.00	.			.
1200	Business Machine	Workstation	\$3,300.00	.			.
1700	Travel Laptop	Laptop	\$3,400.00	.			.
2101	Analog Cell Phone	Phone	\$35.00	.			.
2102	Digital Cell Phone	Phone	\$175.00	.			.
2200	Office Phone	Phone	\$130.00	.			.
5001	Spreadsheet Software	Software	\$299.00	.			.
5002	Database Software	Software	\$399.00	.			.
5003	Wordprocessor Software	Software	\$299.00	.			.
5004	Graphics Software	Software	\$299.00	.			.
.	.	.	.	1110	Dream Machine	Workstation	\$3,200.00
.	.	.	.	1200	Business Machine	Workstation	\$3,300.00
.	.	.	.	1700	Travel Laptop	Laptop	\$3,400.00
.	.	.	.	2101	Analog Cell Phone	Phone	\$35.00
.	.	.	.	2102	Digital Cell Phone	Phone	\$175.00
.	.	.	.	2200	Office Phone	Phone	\$130.00
.	.	.	.	5001	Spreadsheet Software	Software	\$299.00
.	.	.	.	5002	Database Software	Software	\$399.00
.	.	.	.	5003	Wordprocessor Software	Software	\$299.00
.	.	.	.	5004	Graphics Software	Software	\$299.00

The OUTER UNION operator automatically concatenates rows from two queries with no overlap, unless the CORRESPONDING (CORR) keyword is specified as part of the operator. The column names that are assigned to the results are derived from the names in the first query. In the next example, the CORR keyword enables columns with the same name and attributes to be overlaid.

SQL Code

```
PROC SQL;
  SELECT prodnum, prodname, prodtype, prodcost
    FROM PRODUCTS
  OUTER UNION CORR      ①
  SELECT prodnum, prodname, prodtype, prodcost
    FROM PRODUCTS;
QUIT;
```

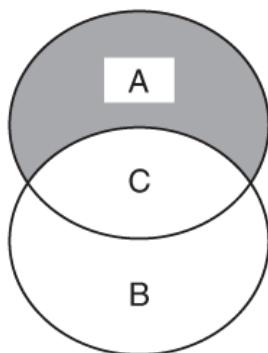
- ① The OUTER UNION operator with the CORR keyword concatenates and overlays the results of both queries.

Results

Product Number	Product Name	Product Type	Product Cost
1110	Dream Machine	Workstation	\$3,200.00
1200	Business Machine	Workstation	\$3,300.00
1700	Travel Laptop	Laptop	\$3,400.00
2101	Analog Cell Phone	Phone	\$35.00
2102	Digital Cell Phone	Phone	\$175.00
2200	Office Phone	Phone	\$130.00
5001	Spreadsheet Software	Software	\$299.00
5002	Database Software	Software	\$399.00
5003	Wordprocessor Software	Software	\$299.00
5004	Graphics Software	Software	\$299.00
1110	Dream Machine	Workstation	\$3,200.00
1200	Business Machine	Workstation	\$3,300.00
1700	Travel Laptop	Laptop	\$3,400.00
2101	Analog Cell Phone	Phone	\$35.00
2102	Digital Cell Phone	Phone	\$175.00
2200	Office Phone	Phone	\$130.00
5001	Spreadsheet Software	Software	\$299.00
5002	Database Software	Software	\$399.00
5003	Wordprocessor Software	Software	\$299.00
5004	Graphics Software	Software	\$299.00

Comparing Rows from Two Queries

The EXCEPT operator compares rows from two queries to determine the changes made to the first table that are not present in the second table. The following results show new and changed rows in the first table that are not in the second table, but do not show rows that have been deleted from the second table. As illustrated in Figure 7.4, the results of specifying the EXCEPT operator represent the shaded area (A).

Figure 7.4: Compare Two Tables to Determine Additions and Changes

When working with two tables that consist of similar information, you can use the EXCEPT operator to determine new and modified rows. The EXCEPT operator is used to identify rows in the first table (or query), but is not used to identify rows in the second table (or query). It also uniquely identifies rows that have changed from the first table to the second table. Columns are compared in the order that they appear in the SELECT statement.

If the wildcard character “*” is specified in the SELECT statement, it is assumed that the tables are structurally identical to one another. Let's look at an example.

Suppose that you have master and backup tables of the CUSTOMERS file, and you want to compare them to identify the new and changed rows. The EXCEPT operator as illustrated in the next example returns all new or changed rows from the CUSTOMERS table that do not appear in the CUSTOMERS_BACKUP table. As illustrated in the results, three new customer rows are added to the CUSTOMERS table that had not previously existed in the CUSTOMERS_BACKUP table.

SQL Code

```
PROC SQL;
  SELECT *
    FROM CUSTOMERS_BACKUP
EXCEPT      ①
  SELECT *
    FROM CUSTOMERS;
QUIT;
```

- ① The EXCEPT operator compares rows in both tables to identify the rows existing in the first table but not the second table.

Results

Customer Number	Customer Name	Customer's Home City
1302	Software Intelligence Cor	Spring Valley
1901	Shipp Consulting	San Pedro
1902	Gupta Programming	Simi Valley

Data Structure Transformations

Data structure transformations involve the process of taking a table's observations and/or columns of data in one format and converting them to another format. This is a common step that many, if not most, users need to perform when extracting, transforming, and loading (ETL) data in preparation for conducting data analysis. It's also a common practice that organizations implement one or more business rules to safeguard the data transformation process from data accuracy and integrity from occurring. For example, a business rule might be implemented to prevent products from being purchased by customers when a product is never actually sold by a manufacturer. Should a business rule like this be violated then a WTF (Where's This From) error handling routine would be triggered to help alert the organization of the situation.

The big question on many users' minds is why anyone should care about transforming data structures. The simple answer is in many database environments data is not always stored in the way it's needed for conducting analysis. In fact, it's frequently found that data in one database table is not stored in the same way as it is in another table. Due to the propensity of inconsistent data mappings of key (or other important) data values, users are frequently faced with finding ways to prep and transform data structures to a desired format for analysis purposes.

Types of Transformations

There are many types of data structure transformations that users should be familiar with. The following table illustrates the leading types of data structure transformations along with a brief description of each.

Transformation	Description
Data Cleaning	The process of detecting and correcting incomplete, inaccurate, or incorrect data.
Filtering	The process of selecting and/or subsetting desired rows and/or columns of data.
Sorting	The process of arranging data into a meaningful order for understanding, analysis, reporting, and/or visualization.
Data Integration	The process of assigning each column and/or data element a consistent and standard definition.
De-duplication	The process of identifying and removing duplicate observations based on the key or all the columns representing the observation.
Format Revision	The process of converting character, numeric, date/time, and/or units of measurement values for input, process, and output.
Derived Column	The process of applying business rules associated with the creation of a new column.
Data Validation	The process of checking the accuracy and quality of source data, the processing of data, and/or the output data and results.
Joining	The process of linking or connecting data from two or more sources using one of the various join approaches (e.g., Cartesian Product, inner, left outer, right outer, or full outer join).

Transformation	Description
Concatenation	The process of combining or stacking query results or tables, one after the other, into a single table or result.
Interleaving	The process of combining two or more sorted results or tables of data into a single sorted table.
Splitting	The process of breaking a table into multiple tables.
Summarization	The process of computing down rows and across columns for the purpose of deriving statistics such as the count, average value, maximum value, and minimum value.
Aggregation	The process of combining separate items or units for statistical analysis purposes.
Convert Long to Wide	The process of reshaping long rows of data into wide tables of columns.
Convert Wide to Long	The process of reshaping wide columns of data into long rows of data with fewer columns.

Concatenating Tables of Data

A popular concatenation approach for PROC SQL users is to specify the OUTER UNION set operator. The next example shows the use of the OUTER UNION set operator to process the concatenated results of the first query with the results of the second query without overlaying columns.

SQL Code

```
PROC SQL;
  SELECT *
    FROM MANUFACTURERS
```

OUTER UNION

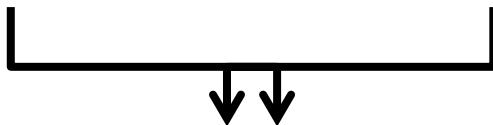
```
SELECT *
  FROM PRODUCTS;
QUIT;
```

The results of the concatenation operation show the two input tables: MANUFACTURERS and PRODUCTS concatenated together. Note: The results from the OUTER UNION operation contain two columns with the same name: MANUNUM (Manufacturer Number).

Results

Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State
111	Cupid Computer	Houston	TX
210	Global Comm Corp	San Diego	CA
600	World Internet Corp	Miami	FL
120	Storage Devices Inc	San Mateo	CA
500	KPL Enterprises	San Diego	CA
700	San Diego PC Planet	San Diego	CA

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
1700	Travel Laptop	170	Laptop	\$3,400.00
2101	Analog Cell Phone	210	Phone	\$35.00
2102	Digital Cell Phone	210	Phone	\$175.00
2200	Office Phone	220	Phone	\$130.00
5001	Spreadsheet Software	500	Software	\$299.00
5002	Database Software	500	Software	\$399.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00



Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State	Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
111	Cupid Computer	Houston	TX	-	-	-	-	-
210	Global Comm Corp	San Diego	CA	-	-	-	-	-
600	World Internet Corp	Miami	FL	-	-	-	-	-
120	Storage Devices Inc	San Mateo	CA	-	-	-	-	-
500	KPL Enterprises	San Diego	CA	-	-	-	-	-
700	San Diego PC Planet	San Diego	CA	-	-	-	-	-
-	-	-	-	1110	Dream Machine	111	Workstation	\$3,200.00
-	-	-	-	1200	Business Machine	120	Workstation	\$3,300.00
-	-	-	-	1700	Travel Laptop	170	Laptop	\$3,400.00
-	-	-	-	2101	Analog Cell Phone	210	Phone	\$35.00
-	-	-	-	2102	Digital Cell Phone	210	Phone	\$175.00
-	-	-	-	2200	Office Phone	220	Phone	\$130.00
-	-	-	-	5001	Spreadsheet Software	500	Software	\$299.00
-	-	-	-	5002	Database Software	500	Software	\$399.00
-	-	-	-	5003	Wordprocessor Software	500	Software	\$299.00
-	-	-	-	5004	Graphics Software	500	Software	\$299.00

To overlay or prevent the display of the duplicate column, MANUNUM, in the results, the CORR (CORRESPONDING) keyword can be specified in the OUTER UNION set operator. The next example shows the CORR keyword being specified in the OUTER UNION set operator to process the concatenated results and overlay the duplicate column.

SQL Code

```
PROC SQL;
  SELECT *
  FROM MANUFACTURERS
```

OUTER UNION CORR

```
SELECT *
  FROM PRODUCTS;
QUIT;
```

Results

Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State	Product Number	Product Name	Product Type	Product Cost
111	Cupid Computer	Houston	TX
210	Global Comm Corp	San Diego	CA
600	World Internet Corp	Miami	FL
120	Storage Devices Inc	San Mateo	CA
500	KPL Enterprises	San Diego	CA
700	San Diego PC Planet	San Diego	CA
111				1110	Dream Machine	Workstation	\$3,200.00
120				1200	Business Machine	Workstation	\$3,300.00
170				1700	Travel Laptop	Laptop	\$3,400.00
210				2101	Analog Cell Phone	Phone	\$35.00
210				2102	Digital Cell Phone	Phone	\$175.00
220				2200	Office Phone	Phone	\$130.00
500				5001	Spreadsheet Software	Software	\$299.00
500				5002	Database Software	Software	\$399.00
500				5003	Wordprocessor Software	Software	\$299.00
500				5004	Graphics Software	Software	\$299.00

Interleaving Tables of Data

As we saw in the previous example, the results of the duplicate column, MANUNUM, are overlaid with the OUTER UNION CORR keyword. In the next example, the results are interleaved and displayed in ascending order with the ORDER BY clause. Note: Columns that do not have corresponding columns are automatically retained in the result set.

SQL Code

```
PROC SQL;
  CREATE TABLE Interleaving_MANU_PROD AS
    SELECT *
      FROM MYDATA.MANUFACTURERS
```

OUTER UNION CORRESPONDING

```
SELECT *
  FROM MYDATA.PRODUCTS
 ORDER BY MANUNUM;
```

```
SELECT * FROM Interleaving_MANU_PROD;
QUIT;
```

Results

Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State	Product Number	Product Name	Product Type	Product Cost
111	Cupid Computer	Houston	TX
111				1110	Dream Machine	Workstation	\$3,200.00
120	Storage Devices Inc	San Mateo	CA
120				1200	Business Machine	Workstation	\$3,300.00
170				1700	Travel Laptop	Laptop	\$3,400.00
210				2102	Digital Cell Phone	Phone	\$175.00
210				2101	Analog Cell Phone	Phone	\$35.00
210	Global Comm Corp	San Diego	CA
220				2200	Office Phone	Phone	\$130.00
500				5002	Database Software	Software	\$399.00
500				5001	Spreadsheet Software	Software	\$299.00
500				5003	Wordprocessor Software	Software	\$299.00
500	KPL Enterprises	San Diego	CA
500				5004	Graphics Software	Software	\$299.00
600	World Internet Corp	Miami	FL
700	San Diego PC Planet	San Diego	CA

Splitting a Table into Multiple Tables

As experienced users are well aware, SAS gives us many ways to perform a great number of tasks. The ability to split a table into multiple tables is one of these. For many users the DATA step is the go-to approach for splitting a table into multiple tables. But, this approach is not always the first choice for SQL users. Often, SQL users integrate PROC SQL with the macro language to emulate DATA step processing in the creation of multiple tables from a table.

The next example shows a simple and less than sophisticated approach to splitting a table into multiple tables. Although the code, as shown, is not as efficient and flexible as using other approaches, primarily because of the hardcoded logic, it does serve to illustrate the general approach of splitting a table into multiple tables with multiple CREATE TABLE statements.

SQL Code

```
PROC SQL;
```

```

CREATE TABLE WORK.Prodtype_Laptop AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Laptop")) ;

CREATE TABLE WORK.Prodtype_Phone AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Phone")) ;

CREATE TABLE WORK.Prodtype_Software AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Software")) ;

CREATE TABLE WORK.Prodtype_Workstation AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Workstation")) ;

QUIT;

```

SAS Log Results

```

PROC SQL;
  CREATE TABLE WORK.Prodtype_Laptop AS
    SELECT *
      FROM PRODUCTS (WHERE=(PRODTYPE="Laptop"));
NOTE: Table WORK.PRODTYPE_LAPTOP created, with 1 rows and 5 columns.
CREATE TABLE WORK.Prodtype_Phone AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Phone"));
NOTE: Table WORK.PRODTYPE_PHONE created, with 3 rows and 5 columns.
CREATE TABLE WORK.Prodtype_Software AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Software"));
NOTE: Table WORK.PRODTYPE_SOFTWARE created, with 4 rows and 5 columns.
CREATE TABLE WORK.Prodtype_Workstation AS
  SELECT *
    FROM PRODUCTS (WHERE=(PRODTYPE="Workstation"));
NOTE: Table WORK.PRODTYPE_WORKSTATION created, with 2 rows and 5
columns.
QUIT;

```

In the next example, a more efficient and flexible approach is used to split a table into multiple tables. The first query derives a count of the distinct (unique) values for the PRODTYPE (Product Type) categorical variable storing the result to a single-value (aggregate) macro variable using the SELECT – INTO clause. The second query derives the unique PRODTYPE values and saves the result to a value-list macro variable separating each value with a ‘~’ (tilde). Finally, a user-defined macro routine called, LOOP_CREATE_TABLE, is specified to control the process of splitting the PRODUCTS table into separate tables using iterative %DO – %END logic to conditionally execute multiple CREATE TABLE statements. Finally, a WHERE clause is specified for subsetting purposes along with a %SCAN function to derive and subset PRODTYPE values for the naming of each separate table.

SQL Code

```
PROC SQL NOPRINT;
```

```

SELECT COUNT(DISTINCT PRODTYPE) AS ProdType_Cnt
  INTO :mProdtype_Cnt
   FROM PRODUCTS;
SELECT DISTINCT PRODTYPE
  INTO :mProdtype_Lst SEPARATED BY '~'
   FROM PRODUCTS;

```

```

QUIT;
%PUT mProdtype_Cnt = &mProdtype_Cnt;
%PUT mProdtype_Lst = &mProdtype_Lst;
%MACRO LOOP_CREATE_TABLE;

```

```

%DO I = 1 %TO &mProdtype_Cnt;
  PROC SQL;
    CREATE TABLE WORK.Prodtype_%SCAN(&mProdtype_Lst,&I,~) AS
      SELECT *
        FROM

```

```

PRODUCTS (WHERE=(PRODTYPE="%SCAN(&mProdtype_Lst,&I,~)"));

QUIT;
%END;

%MEND LOOP_CREATE_TABLE;

%LOOP_CREATE_TABLE;

```

SAS Log Results

```

PROC SQL;
  SELECT COUNT(DISTINCT PRODTYPE) AS ProdType_Cnt
    INTO :mProdtype_Cnt
      FROM PRODUCTS;
  SELECT DISTINCT PRODTYPE
    INTO :mProdtype_Lst SEPARATED BY '~'
      FROM PRODUCTS;
QUIT;

```

NOTE: PROCEDURE SQL used (Total process time):
 real time 0.04 seconds
 cpu time 0.04 seconds

```

%PUT mProdtype_Cnt = &mProdtype_Cnt;
mProdType_Cnt = 4

%PUT mProdtype_Lst = &mProdtype_Lst;
mProdType_Lst = Laptop~Phone~Software~Workstation

```

```

%MACRO LOOP_CREATE_TABLE;
%DO I = 1 %TO &mProdtype_Cnt;
  PROC SQL;
    CREATE TABLE WORK.Prodtype_%SCAN(&mProdtype_Lst,&I,~)
  AS
    SELECT *
      FROM
PRODUCTS (WHERE=(PRODTYPE="%SCAN(&mProdtype_Lst,&I,~)"));
    QUIT;
  %END;
%MEND LOOP_CREATE_TABLE;

```

```
%LOOP_CREATE_TABLE;
```

NOTE: Table WORK.PRODTYPE_LAPTOP created, with 1 rows and 5 columns.
 NOTE: Table WORK.PRODTYPE_PHONE created, with 3 rows and 5 columns.
 NOTE: Table WORK.PRODTYPE_SOFTWARE created, with 4 rows and 5 columns.
 NOTE: Table WORK.PRODTYPE_WORKSTATION created, with 2 rows and 5 columns.

Complex Query Applications

Query applications come in all forms and can typically be classified into three distinct categories:

- **Production-oriented queries** rarely change, are run as needed (e.g., daily, weekly, monthly, etc.), and consist of SQL statements, parameter lists, and/or action queries.
- **Ad-hoc queries** are typically constructed as needed and are often used once or in some unpredictable way to solve a particular need or problem.
- **Custom queries** are classified as falling somewhere in-between production-oriented and ad-hoc queries, where the query is essentially the same each time its run, but conditional, parameter, action processing (e.g., create a new table, update one or more rows in a table, delete one or more rows in a table, and append one or more rows to an existing table), and data differences exist and must be handled.

In this section, a few complex query applications are presented to show how the SQL procedure can be used to satisfy specific processing and/or data management requirements. The applications have been selected, in part, based on the needs many SAS users have, the emulation of popular DATA step techniques as SQL queries, and to share interesting SQL procedure coding techniques and approaches.

One-to-One, One-to-Many, Many-to-One, and Many-to-Many Relationships

Input tables are frequently characterized by the way rows in one table relate to one or more rows in another table. This process, referred to as the data relationship between two or more tables, consists of four categories:

- one-to-one
- one-to-many
- many-to-one
- many-to-many

To better understand how data sources can be processed for producing desirable results, it helps to be able to differentiate between the four data relationship categories. The following example illustrates coding conventions for application of one-to-one, one-to-many, many-to-one, and many-to-many data relationships in the SQL procedure.

SQL Code

```
*****  
/** PROGRAM NAME: DATA-RELATIONSHIPS.SAS */  
/** PURPOSE.....: Derive one-to-one, one-to-many, many-to-one, */  
/**           and many-to-many data relationships using */  
/**           complex queries. */  
/** AUTHOR.....: Kirk Paul Lafler */  
/** DATE WRITTEN: August 30, 2012 */  
*****  
proc sql noprint;  
*****  
/** ROUTINE.....: ONE-TO-ONE */
```

```
/** PURPOSE.....: Produce a one-to-one data relationship using */
/**           a base table with a unique key and a lookup   */
/**           table with a unique key in an EQUIJOIN.      */
/*********************************************************/
create table one_to_one as
  select m.manunum,
         m.manuname,
         p.prodtype,
         p.prodcost
    from manufacturers  m,
         products       p
   where m.manunum=p.manunum;
/*********************************************************/
/** ROUTINE.....: ONE-TO-MANY                           */
/** PURPOSE.....: Produce a one-to-many data relationship */
/**           using the table with the unique key as a      */
/**           lookup table and a LEFT JOIN.                 */
/*********************************************************/
create table one_to_many as
  select p1.prodnum,
         p1.prodname,
         p2.units,
         p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products     p1
      LEFT JOIN
      purchases     p2
   on p1.prodnum=p2.prodnum;
/*********************************************************/
/** ROUTINE.....: MANY-TO-ONE                           */
/** PURPOSE.....: Produce a many-to-one data relationship */
/**           using a LEFT OUTER JOIN.                   */
/*********************************************************/
create table many_to_one as
  select p2.prodnum,
         p2.prodname,
         p1.units,
         p1.unitcost,
         p1.units * p1.unitcost as Total_Cost format=dollar12.2
    from purchases   p1
      LEFT JOIN
      products     p2
   on p1.prodnum=p2.prodnum;
/*********************************************************/
/** ROUTINE.....: MANY-TO-MANY                           */
/** PURPOSE.....: Produce a many-to-many data relationship */
/**           using the table with the unique key and a      */
/**           lookup table in an EQUIJOIN construct.        */
/*********************************************************/
create table many_to_many as
  select p1.prodnum,
         p1.prodname,
         p2.units,
         p2.unitcost,
         p2.units * p2.unitcost as Total_Cost format=dollar12.2
    from products   p1,
         purchases  p2
```

```
where p1.prodnum=p2.prodnum;  
quit;
```

Results

The resulting tables for a one-to-one, one-to-many, many-to-one, and many-to-many data relationship are displayed in Figure 7.5 through Figure 7.8.

Figure 7.5: One-to-One Data Relationship

	Manufacturer Number	Manufacturer Name	Product Type	Product Cost
1		111 Cupid Computer	Workstation	\$3,200.00
2		120 Storage Devices Inc	Workstation	\$3,300.00
3		210 Global Comm Corp	Phone	\$35.00
4		210 Global Comm Corp	Phone	\$175.00
5		500 KPL Enterprises	Software	\$299.00
6		500 KPL Enterprises	Software	\$399.00
7		500 KPL Enterprises	Software	\$299.00
8		500 KPL Enterprises	Software	\$299.00

Figure 7.6: One-to-Many Data Relationship

	Product Number	Product Name	Units	Unitcost	Total_Cost
1		1110 Dream Machine	2	\$3,200.00	\$6,400.00
2		1110 Dream Machine	1	\$3,200.00	\$3,200.00
3		1110 Dream Machine	5	\$3,200.00	\$16,000.00
4		1200 Business Machine	3	\$3,300.00	\$9,900.00
5		1200 Business Machine	3	\$3,300.00	\$9,900.00
6		1200 Business Machine	3	\$3,300.00	\$9,900.00
7		1200 Business Machine	6	\$3,300.00	\$19,800.00
8		1200 Business Machine	8	\$3,300.00	\$26,400.00
9		1700 Travel Laptop	2	\$3,400.00	\$6,800.00
10		1700 Travel Laptop	2	\$3,400.00	\$6,800.00
11		1700 Travel Laptop	3	\$3,400.00	\$10,200.00
12		1700 Travel Laptop	7	\$3,400.00	\$23,800.00
13		1700 Travel Laptop	3	\$3,400.00	\$10,200.00
14		1700 Travel Laptop	4	\$3,400.00	\$13,600.00
15		1700 Travel Laptop	5	\$3,400.00	\$17,000.00
16		1700 Travel Laptop	3	\$3,400.00	\$10,200.00
17		1700 Travel Laptop	4	\$3,400.00	\$13,600.00
18		1700 Travel Laptop	2	\$3,400.00	\$6,800.00
19		1700 Travel Laptop	5	\$3,400.00	\$17,000.00
20		2101 Analog Cell Phone	.		
21		2102 Digital Cell Phone	11	\$175.00	\$1,925.00
• • • • •					
47		5003 Wordprocessor Software	9	\$299.00	\$2,691.00
48		5003 Wordprocessor Software	5	\$299.00	\$1,495.00
49		5003 Wordprocessor Software	5	\$299.00	\$1,495.00
50		5003 Wordprocessor Software	7	\$299.00	\$2,093.00
51		5003 Wordprocessor Software	3	\$299.00	\$897.00
52		5003 Wordprocessor Software	8	\$299.00	\$2,392.00
53		5003 Wordprocessor Software	8	\$299.00	\$2,392.00
54		5004 Graphics Software	3	\$299.00	\$897.00
55		5004 Graphics Software	1	\$299.00	\$299.00
56		5004 Graphics Software	3	\$299.00	\$897.00
57		5004 Graphics Software	2	\$299.00	\$598.00
58		5004 Graphics Software	2	\$299.00	\$598.00

Figure 7.7: Many-to-One Data Relationship

	Product Number	Product Name	Units	Unitcost	Total_Cost
1	1110	Dream Machine	2	\$3,200.00	\$6,400.00
2	1110	Dream Machine	1	\$3,200.00	\$3,200.00
3	1110	Dream Machine	5	\$3,200.00	\$16,000.00
4	1200	Business Machine	3	\$3,300.00	\$9,900.00
5	1200	Business Machine	3	\$3,300.00	\$9,900.00
6	1200	Business Machine	3	\$3,300.00	\$9,900.00
7	1200	Business Machine	6	\$3,300.00	\$19,800.00
8	1200	Business Machine	8	\$3,300.00	\$26,400.00
9	1700	Travel Laptop	2	\$3,400.00	\$6,800.00
10	1700	Travel Laptop	2	\$3,400.00	\$6,800.00
11	1700	Travel Laptop	3	\$3,400.00	\$10,200.00
12	1700	Travel Laptop	7	\$3,400.00	\$23,800.00
13	1700	Travel Laptop	3	\$3,400.00	\$10,200.00
14	1700	Travel Laptop	4	\$3,400.00	\$13,600.00
15	1700	Travel Laptop	5	\$3,400.00	\$17,000.00
16	1700	Travel Laptop	3	\$3,400.00	\$10,200.00
17	1700	Travel Laptop	4	\$3,400.00	\$13,600.00
18	1700	Travel Laptop	2	\$3,400.00	\$6,800.00
19	1700	Travel Laptop	5	\$3,400.00	\$17,000.00
20	2102	Digital Cell Phone	11	\$175.00	\$1,925.00
21	2102	Digital Cell Phone	9	\$175.00	\$1,575.00
...
47	5003	Wordprocessor Software	5	\$299.00	\$1,495.00
48	5003	Wordprocessor Software	5	\$299.00	\$1,495.00
49	5003	Wordprocessor Software	7	\$299.00	\$2,093.00
50	5003	Wordprocessor Software	3	\$299.00	\$897.00
51	5003	Wordprocessor Software	8	\$299.00	\$2,392.00
52	5003	Wordprocessor Software	8	\$299.00	\$2,392.00
53	5004	Graphics Software	3	\$299.00	\$897.00
54	5004	Graphics Software	1	\$299.00	\$299.00
55	5004	Graphics Software	3	\$299.00	\$897.00
56	5004	Graphics Software	2	\$299.00	\$598.00
57	5004	Graphics Software	2	\$299.00	\$598.00

Figure 7.8: Many-to-Many Data Relationship

	Product Number	Product Name	Units	Unitcost	Total_Cost
1	1110	Dream Machine	1	\$3,200.00	\$3,200.00
2	5001	Spreadsheet Software	7	\$299.00	\$2,093.00
3	5001	Spreadsheet Software	11	\$299.00	\$3,289.00
4	5003	Wordprocessor Software	8	\$299.00	\$2,392.00
5	5002	Database Software	4	\$399.00	\$1,596.00
6	5004	Graphics Software	3	\$299.00	\$897.00
7	1700	Travel Laptop	2	\$3,400.00	\$6,800.00
8	1200	Business Machine	3	\$3,300.00	\$9,900.00
9	1110	Dream Machine	2	\$3,200.00	\$6,400.00
10	5001	Spreadsheet Software	3	\$299.00	\$897.00
11	5003	Wordprocessor Software	5	\$299.00	\$1,495.00
12	5002	Database Software	2	\$399.00	\$798.00
13	1700	Travel Laptop	2	\$3,400.00	\$6,800.00
14	1200	Business Machine	3	\$3,300.00	\$9,900.00
15	1110	Dream Machine	5	\$3,200.00	\$16,000.00
16	5001	Spreadsheet Software	9	\$299.00	\$2,691.00
17	5002	Database Software	5	\$399.00	\$1,995.00
18	5003	Wordprocessor Software	8	\$299.00	\$2,392.00
19	5004	Graphics Software	2	\$299.00	\$598.00
20	5001	Spreadsheet Software	11	\$299.00	\$3,289.00
21	5002	Database Software	5	\$399.00	\$1,995.00
• • • • •					
47	1700	Travel Laptop	7	\$3,400.00	\$23,800.00
48	1700	Travel Laptop	4	\$3,400.00	\$13,600.00
49	1700	Travel Laptop	5	\$3,400.00	\$17,000.00
50	1700	Travel Laptop	2	\$3,400.00	\$6,800.00
51	1200	Business Machine	8	\$3,300.00	\$26,400.00
52	5001	Spreadsheet Software	3	\$299.00	\$897.00
53	5003	Wordprocessor Software	5	\$299.00	\$1,495.00
54	5004	Graphics Software	1	\$299.00	\$299.00
55	1700	Travel Laptop	4	\$3,400.00	\$13,600.00
56	5001	Spreadsheet Software	6	\$299.00	\$1,794.00
57	2102	Digital Cell Phone	9	\$175.00	\$1,575.00

Processing First, Last, and Between Rows for BY-and Groups

Occasionally, SAS users might find that the SQL procedure falls short in its ability to offer an “out of the box” solution to a SAS programming technique. As a result, and often out of frustration, users abandon their search for a possible SQL solution turn instead to a familiar DATA step technique or to one of the many procedures (which often is outside the SQL procedure).

As a case in point, the ability to perform BY-group processing using a DATA step BY statement for identifying FIRST, LAST, and BETWEEN observations is a popular technique with SAS users. Unfortunately, an equivalent process using the SQL procedure either doesn’t exist, or isn’t easy to find. After years of researching and looking unsuccessfully for an SQL technique that would emulate this DATA step processing stalwart, I decided to develop a

solution of my own. The following SQL code emulates the behavior of the FIRST, LAST, and BETWEEN processing.

SQL Code

```
*****
/** PROGRAM NAME: FIRST-BETWEEN-LAST-ROWS.SAS          */
/** PURPOSE.....: Derive the first (min) row, last (max) row   */
/**               and between rows for each by-group using    */
/**               subqueries.                                     */
/** AUTHOR.....: Kirk Paul Lafler                         */
/** DATE WRITTEN: June 4, 2012                            */
*****
***** ROUTINE.....: FIRST-BY-GROUP-ROWS                */
/** PURPOSE.....: Derive the first (min) row within each   */
/**               by-group using a subquery.                   */
*****
proc sql;
  create table first_bygroup_rows as
    select custnum,
           prodnum,
           units,
           unitcost,
           'FirstRow' as ByGroup
    from purchases P1
    where prodnum =
      (select min(prodnum)
       from purchases P2
       where P1.custnum = P2.custnum)
    order by custnum, prodnum;
*****
/** ROUTINE.....: LAST-BY-GROUP-ROWS                  */
/** PURPOSE.....: Derive the last (max) row within each   */
/**               by-group using a subquery.                 */
*****
create table last_bygroup_rows as
  select custnum,
         prodnum,
         units,
         unitcost,
         'LastRow' as ByGroup
  from purchases P1
  where prodnum =
    (select max(prodnum)
     from purchases P2
     where P1.custnum = P2.custnum)
  order by custnum, prodnum;
*****
/** ROUTINE.....: BETWEEN-BY-GROUP-ROWS               */
/** PURPOSE.....: Derive all rows between the first (min) row, */
/**               and the last (max) row within each by-group   */
/**               using a subquery.                           */
*****
create table between_bygroup_rows as
  select custnum,
         prodnum,
         units,
```

```

unitcost,
min(prodnum) as Min_Prodnum,
max(prodnum) as Max_Prodnum,
'BetweenRow' as ByGroup
from purchases
group by custnum
having CALCULATED min_Prodnum NOT =
       CALCULATED max_Prodnum AND
       CALCULATED min_Prodnum NOT =
          prodnum           AND
       CALCULATED max_Prodnum NOT = prodnum
order by custnum, prodnum;
/*****************************************************************/
/** ROUTINE.....: CONCATENATE-FIRST-BETWEEN-LAST               */
/** PURPOSE.....: Concatenate the results from the first (min) */
/**                row, between rows, and last (max) row within */
/**                each by-group using UNION ALL set operators. */
/*****************************************************************/
create table first_between_last_rows as
select custnum,
       prodnum,
       units,
       unitcost,
       bygroup
  from first_bygroup_rows
UNION ALL
  select custnum,
         prodnum,
         units,
         unitcost,
         bygroup
    from between_bygroup_rows
UNION ALL
  select custnum,
         prodnum,
         units,
         unitcost,
         bygroup
    from last_bygroup_rows;
/*****************************************************************/
/** ROUTINE.....: PRINT-FIRST-BETWEEN-LAST                      */
/** PURPOSE.....: Print the results from the first (min) row, */
/**                between rows, and last (max) row within each */
/**                by-group using a select query.                  */
/*****************************************************************/
reset number;
select *
  from first_between_last_rows;
quit;

```

Results

The results for the FIRST., LAST., and BETWEEN. rows are displayed in Figure 7.9 through Figure 7.11.

Figure 7.9: FIRST. Rows Results

	Custnum	Prodnum	Units	Unitcost	ByGroup
1	101	1200	3	\$3,300.00	First Row
2	201	1700	3	\$3,400.00	First Row
3	301	2102	8	\$175.00	First Row
4	401	1200	6	\$3,300.00	First Row
5	501	1700	4	\$3,400.00	First Row
6	701	1110	2	\$3,200.00	First Row
7	801	2102	5	\$175.00	First Row
8	901	1110	5	\$3,200.00	First Row
9	1001	1700	5	\$3,400.00	First Row
10	1101	1700	2	\$3,400.00	First Row
11	1201	1200	8	\$3,300.00	First Row
12	1301	1700	3	\$3,400.00	First Row
13	1401	2102	7	\$175.00	First Row
14	1601	1700	7	\$3,400.00	First Row
15	1701	1110	1	\$3,200.00	First Row
16	1801	1700	4	\$3,400.00	First Row

Figure 7.10: LAST. Rows Results

	Custnum	Prodnum	Units	Unitcost	ByGroup
1	101	5004	2	\$299.00	Last Row
2	201	5003	9	\$299.00	Last Row
3	301	5001	6	\$299.00	Last Row
4	401	5004	3	\$299.00	Last Row
5	501	5004	1	\$299.00	Last Row
6	701	5004	3	\$299.00	Last Row
7	801	2102	5	\$175.00	Last Row
8	901	5004	2	\$299.00	Last Row
9	1001	1700	5	\$3,400.00	Last Row
10	1101	2200	3	\$130.00	Last Row
11	1201	1200	8	\$3,300.00	Last Row
12	1301	5003	5	\$299.00	Last Row
13	1401	2102	7	\$175.00	Last Row
14	1601	1700	7	\$3,400.00	Last Row
15	1701	1110	1	\$3,200.00	Last Row
16	1801	1700	4	\$3,400.00	Last Row

Figure 7.11: BETWEEN. Rows Results

	Custnum	Prodnum	Units	Unitcost	Min_Prodnum	Max_Prodnum	ByGroup
1	101	1700	5	\$3,400.00	1200	5004	BetweenRow
2	101	2102	9	\$175.00	1200	5004	BetweenRow
3	101	5001	7	\$299.00	1200	5004	BetweenRow
4	101	5003	3	\$299.00	1200	5004	BetweenRow
5	201	2102	5	\$175.00	1700	5003	BetweenRow
6	201	5001	6	\$299.00	1700	5003	BetweenRow
7	201	5001	2	\$299.00	1700	5003	BetweenRow
8	201	5001	6	\$299.00	1700	5003	BetweenRow
9	201	5002	4	\$399.00	1700	5003	BetweenRow
10	401	1700	3	\$3,400.00	1200	5004	BetweenRow
11	401	5001	11	\$299.00	1200	5004	BetweenRow
12	401	5002	5	\$399.00	1200	5004	BetweenRow
13	401	5003	7	\$299.00	1200	5004	BetweenRow
14	501	2102	9	\$175.00	1700	5004	BetweenRow
15	501	2102	12	\$175.00	1700	5004	BetweenRow
16	501	5001	3	\$299.00	1700	5004	BetweenRow
17	501	5003	5	\$299.00	1700	5004	BetweenRow
18	701	1200	3	\$3,300.00	1110	5004	BetweenRow
19	701	1700	2	\$3,400.00	1110	5004	BetweenRow
20	701	5001	11	\$299.00	1110	5004	BetweenRow
21	701	5002	4	\$399.00	1110	5004	BetweenRow
22	701	5003	8	\$299.00	1110	5004	BetweenRow
23	901	1200	3	\$3,300.00	1110	5004	BetweenRow
24	901	1700	2	\$3,400.00	1110	5004	BetweenRow
25	901	5001	9	\$299.00	1110	5004	BetweenRow
26	901	5001	2	\$299.00	1110	5004	BetweenRow
27	901	5002	5	\$399.00	1110	5004	BetweenRow
28	901	5003	8	\$299.00	1110	5004	BetweenRow
29	1101	2102	9	\$175.00	1700	2200	BetweenRow
30	1301	2102	11	\$175.00	1700	5003	BetweenRow
31	1301	5001	3	\$299.00	1700	5003	BetweenRow
32	1301	5002	2	\$399.00	1700	5003	BetweenRow

Determining the Number of Rows in an Input Table

The SQL procedure provides users with the ability to determine the number of rows contributed by one or more input tables using the VERBOSE option. The VERBOSE option serves to provide important “need to know” information in our quest for learning more about our input data, our query’s processing requirements, and resource utilization (e.g., the number of input rows and the table’s logical record length). The next example shows a four table join query with the SQL procedure VERBOSE option specified.

SQL Code

```
PROC SQL VERBOSE;
  SELECT products.prodname,
         products.prodtype,
         customers.custname,
         manufacturers.manuname
  FROM MANUFACTURERS,
       PRODUCTS,
       INVOICE,
```

```

CUSTOMERS
WHERE manufacturers.manunum = products.manunum AND
      manufacturers.manunum = invoice.manunum AND
      products.prodnum      = invoice.prodnum AND
      invoice.custnum       = customers.custnum;
QUIT;

```

As illustrated in the following SAS log, information pertaining to the number of input rows along with the logical record length (LRECL) is produced for each table when the VERBOSE SQL procedure option is specified.

SAS Log Results

```

PROC SQL VERBOSE;
SELECT P.prodname,
       P.prodtype,
       C.custname,
       M.manuname
  FROM MANUFACTURERS  M,
       PRODUCTS        P,
       INVOICE         I,
       CUSTOMERS      C
 WHERE M.manunum = P.manunum    AND
       M.manunum = I.manunum    AND
       P.prodnum = I.prodnum   AND
       I.custnum = C.custnum;

Data Set Tags.
Data Set WORK.MANUFACTURERS is num=1 and tag=0001. NOBS=6,
lrecl=50.
Data Set WORK.PRODUCTS is num=2 and tag=0002. NOBS=10, lrecl=51.
Data Set WORK.INVOICE is num=3 and tag=0004. NOBS=7, lrecl=20.
Data Set WORK.CUSTOMERS is num=4 and tag=0008. NOBS=18,
lrecl=48.
QUIT;

```

Identifying Tables with the Most Indexes

Database administrators and user support staff are concerned with the installation, configuration, administration, monitoring, and maintenance of the database environment, and consequently take special interest in a variety of activities including the performance and health of database applications. It is particularly important for them to be aware of issues that could impact future expansion requirements. One thing that can affect I/O and storage performance is a large number of indexes in a database environment. For this reason, I often find it valuable to know how many indexes exist as well as which tables have the most indexes defined. The next example illustrates an SQL query that accesses DICTIONARY table content (see Chapter 2, “Working with Data in PROC SQL”) to identify the tables in a database environment with the most indexes.

SQL Code

```
*****
/** PROGRAM NAME: TABLES-WITH-THE-MOST-INDEXES.SAS          */
/** PURPOSE.....: Perform an index analysis to identify the   */
/**               tables with the most indexes.                  */
/** AUTHOR.....: Kirk Paul Lafler                            */
/** DATE WRITTEN: August 22, 2012                           */
*****
```

```
proc sql;
  create table Tables_with_most_Indexes as
    select i.libname,
           i.memname,
           i.name,
           i.idxusage,
           i.idxname,
           count(i.memname) as ctr_memname
              label='Number of Defined Indexes',
           t.nobs
      from dictionary.indexes i,
           dictionary.tables t
     where upcase(i.libname) = upcase(t.libname) and
           upcase(i.memname) = upcase(t.memname) and
           upcase(i.idxusage) IN ('SIMPLE','COMPOSITE','BOTH')
   group by i.libname, i.memname
  order by ctr_memname desc, t.nobs;
quit;
```

Results

The results for tables with the most indexes are displayed in Figure 7.12.

Figure 7.12: TABLES_WITH_MOST_INDEXES

	Library Name	Member Name	Column Name	Column Index Type	Index Name	Number of Defined Indexes	Number of Physical Observations
1	WORK	PRODUCTS	manunum	COMPOSITE	MANNUM_PRODTYP	3	10
2	WORK	PRODUCTS	prodtype	COMPOSITE	MANNUM_PRODTYP	3	10
3	WORK	PRODUCTS	prodtype	SIMPLE	prodtype	3	10
4	MYDATA	PRODUCTS	manunum	COMPOSITE	MANNUM_PRODTYP	3	12
5	MYDATA	PRODUCTS	prodtype	COMPOSITE	MANNUM_PRODTYP	3	12
6	MYDATA	PRODUCTS	prodtype	SIMPLE	prodtype	3	12
7	SASUSER	MYFUNCTIONS	_Key_	SIMPLE	_Key_	1	111

When processing observations in a sequential manner without the use of an index, SAS reads and processes all the observations from a page of disk into memory continuing this process until the end of file. In some scenarios sequential access can be considerably costlier since the SQL optimizer will need to perform a full scan through the data. For more information, see the “Index Processing Costs” section in Chapter 6.

Nearest Neighbor

As a general rule, SQL queries are designed to perform operations on a row-by-row basis. For most processing requests this does not present any real issues. But, occasionally a problem comes along where performing operations on a row-by-row basis is not only inadequate – it will not work. One particular example where this occurs is when a query needs to access data

from different rows at the same time. In this case, the LAG and LEAD functions, which are used by DATA step users to simplify the process, are not available to PROC SQL users. Consequently, a query that accesses data from different rows at the same time from an input table must be constructed to perform extra work.

SQL Code

```
proc sql nonumber;
  select Prodnum, Prodname,
    LAG(Prodname) AS Lag_Prodname,
    LEAD(Prodname) AS Lead_Prodname
  from Products;
quit;
```

Since the LAG and LEAD functions are only valid within the DATA step and not supported in PROC SQL, processing stops and the following ERROR messages are produced and displayed on the SAS log.

SAS Log Results

```
proc sql nonumber;
  select Prodnum, Prodname,
    LAG(Prodname) AS Lag_Prodname,
    LEAD(Prodname) AS Lead_Prodname
  from Products;
```

```
ERROR: The LAG function is not supported in PROC SQL, it is only valid
within the
      DATA step.
ERROR: Function LEAD could not be located.
```

```
NOTE: PROC SQL set option NOEXEC and will continue to check the
syntax of statements.
```

```
quit;
```

```
NOTE: The SAS System stopped processing this step because of errors.
```

The next example illustrates a query that accesses data from different rows at the same time from an input table in two parts. The first part specifies the ODS OUTPUT statement to create a new sequenced (numbered) table from the PRODUCTS table and specifies the NUMBER parameter in the PROC SQL statement. After running this step, the resulting table contains a sequenced table with a newly derived ROW column, and the Prodnum and Prodname columns.

SQL Code

```
ods output sql_results=Products_with_Row_Numbers;
proc sql number;
  select Prodnum, Prodname
  from Products;
quit;
```

Results

Row	Product Number	Product Name
1	1110	Dream Machine
2	1200	Business Machine
3	1700	Travel Laptop
4	2101	Analog Cell Phone
5	2102	Digital Cell Phone
6	2200	Office Phone
7	5001	Spreadsheet Software
8	5002	Database Software
9	5003	Wordprocessor Software
10	5004	Graphics Software

The second part of this example uses a query to process the value of the current row's product name and with that information derives the value of the previous (LAG) row's product name in a subquery with a WHERE clause by specifying Row = M.row - 1 relative positioning and derives the value of the next (LEAD) row's product name in a subquery with a WHERE clause by specifying M.row + 1 relative positioning from the PRODUCTS_with_Row_Numbers table created in the previous step. With each row's current product name identified the values of the previous product name and the next product name can then be identified. This operation is repeated for every row in the input table.

SQL Code

```
proc sql nonumber;
  select Prodnnum, Prodname,
    (select Prodname
      from Products_with_Row_Numbers
      where Row = M.row - 1) AS Previous_prodbname,
    (select Prodname
      from Products_with_Row_Numbers
      where Row = M.row + 1) AS Next_prodbname
    from Products_with_Row_Numbers M;
quit;
```

The value of each current row's product name along with the product name one level back and the product name one level forward is displayed below.

Results

Product Number	Product Name	Previous_prodname	Next_prodname
1110	Dream Machine		Business Machine
1200	Business Machine	Dream Machine	Travel Laptop
1700	Travel Laptop	Business Machine	Analog Cell Phone
2101	Analog Cell Phone	Travel Laptop	Digital Cell Phone
2102	Digital Cell Phone	Analog Cell Phone	Office Phone
2200	Office Phone	Digital Cell Phone	Spreadsheet Software
5001	Spreadsheet Software	Office Phone	Database Software
5002	Database Software	Spreadsheet Software	Wordprocessor Software
5003	Wordprocessor Software	Database Software	Graphics Software
5004	Graphics Software	Wordprocessor Software	

The next example uses a query to process the value of the current row's product name and with that information derives the value of the previous (LAG) row's product name two levels back in a subquery with a WHERE clause by specifying Row = M.row - 2 relative positioning and derives the value of the next (LEAD) row's product name by specifying M.row + 2 relative positioning from the PRODUCTS_with_Row_Numbers table. By processing each row's current product name, the values of the product name two levels back and the product name two levels forward can then be identified. This operation is repeated for every row in the input table.

SQL Code

```
proc sql nonumber;
  select Prodnum, Prodname,
    (select Prodname
      from Products_with_Row_Numbers
      where Row = M.row - 2) AS Previous_2_prodname,
    (select Prodname
      from Products_with_Row_Numbers
      where Row = M.row + 2) AS Next_2_prodname
    from Products_with_Row_Numbers M;
quit;
```

The value of each current row's product name along with the product name two levels back and two levels forward is displayed below.

Results

Product Number	Product Name	Previous_2_prodname	Next_2_prodname
1110	Dream Machine		Travel Laptop
1200	Business Machine		Analog Cell Phone
1700	Travel Laptop	Dream Machine	Digital Cell Phone
2101	Analog Cell Phone	Business Machine	Office Phone
2102	Digital Cell Phone	Travel Laptop	Spreadsheet Software
2200	Office Phone	Analog Cell Phone	Database Software
5001	Spreadsheet Software	Digital Cell Phone	Wordprocessor Software
5002	Database Software	Office Phone	Graphics Software
5003	Wordprocessor Software	Spreadsheet Software	
5004	Graphics Software	Database Software	

Summary

- When one or more relationships or connections between disparate pieces of data are needed, the PROC SQL join construct is used (see the “Why Joins Are Important” section).
- You use a join to relate one table with another table through a process known as column matching (see the “Introducing Complex Queries” section).
- You can assign table aliases to tables to minimize the number of keystrokes needed to reference a table in a join query (see the “Using Table Aliases in Joins” section).
- When a query is placed inside the predicate of another query, it is called a subquery. Put another way, a subquery is a SELECT statement that is embedded in the WHERE clause of another SELECT statement (see the “Subqueries” section).
- The IN predicate permits PROC SQL to pass multiple values from the subquery to the main query without producing an error (see the “Passing More Than One Row with a Subquery” section).
- A subquery can also be constructed to evaluate multiple times, once for each row of data accessed by the main (outer) query (see the “Comparing a Set of Values” section).
- The INTERSECT operator creates an output table that consists of all the unique rows from the intersection of two query expressions (see the “Accessing Rows from the Intersection of Two Queries” section).
- The UNION operator creates an output table that consists of all of the unique rows from the combination of query expressions (see the “Accessing Rows from the Combination of Two Queries” section).

Chapter 8: Working with Views

Introduction	289
Views—Windows to Your Data	289
What Views Aren't.....	290
Types of Views	291
Creating Views	292
Displaying a View's Contents	293
Describing View Definitions	294
Creating and Using Views in SAS.....	295
Views and SAS Procedures	296
Views and DATA Steps.....	297
Eliminating Redundancy	299
Restricting Data Access—Security.....	299
Hiding Logic Complexities	300
Nesting Views	302
Updatable Views	304
Inserting New Rows of Data	304
Updating Existing Rows of Data.....	308
Deleting Rows of Data.....	311
Deleting Views	311
Summary	312

Introduction

In previous chapters, the examples assumed that each table had a physical existence, that is, the data stored in each table occupied storage space. In this chapter, let's turn our attention to a different type of table structure that has no real physical existence. This structure, known as a *virtual table* or *view*, offers users and programmers an incredible amount of flexibility and control. This makes views an ideal way to look at data from a variety of perspectives and according to different users' needs. Unlike tables, views store no data and have only a "virtual" existence. You will learn how to create, access, and delete views as you examine the many examples in this chapter.

Views—Windows to Your Data

Views are one of the more powerful features available in the SQL procedure. They are commonly referred to as "virtual tables" to distinguish them from base tables. The simple difference is that views are not tables, but instead are files that consist of executable instructions. As a query, a view appears to behave as a table with one striking difference—it does not store any data. When referenced, a view always produces up-to-date results just like a table does. So how does a view get its data? Views access data from one or more underlying

tables (base tables) or other views, provide you with your own personal access to data, and can be used in DATA steps as well as by SAS procedures.

Views can be made to extend the security capabilities in dynamic environments where data duplication or data redundancy, logic complexities, and data security are an issue. When properly designed, views can be made to adhere to row-level and column-level security requirements by allowing access to only those columns and/or rows of data with relevance to the information needs of the application. Any columns and/or rows of data that are deemed “off limits,” or are classified as “restricted” can be eliminated from the view’s selection list. Consequently, views can be constructed to allow access to only those portions of an underlying table (or tables) that each user is permitted to access.

Another important feature of views is that they ensure consistently derived data by creating calculated (computed) columns that are based on some arithmetic formula (or algorithm). Database purists often design tables to be free of calculated columns, thereby relying on the view to create computed columns. By allowing views to perform data aggregation, instead of the tables themselves, processing costs can be postponed until needed or requested.

As a means of shielding users from complex logic constructs, views can be designed to look as though a database were designed specifically for a single user as well as for a group of users, each having different needs. Data references are coded once and, only when fully tested and ready for production, can be conveniently stored in common shareable libraries for all to access. Views ensure that the most current input data is used without the need for replicating partial or complete copies of the input data. They also require very little storage space because a view contains only its definition, and does not contain a copy of the data that it presents.

Views are also beneficial when queries or subqueries are repeated a number of times throughout an application. In these situations, the addition of a view enables a change to be made only once, which improves your productivity through a reduction in time and resources. The creation of view libraries should be considered so that users throughout an organization have an easily accessible array of productivity routines as they would a macro.

What Views Aren’t

Views are not tables, but they are file constructs that contain compiled code that access one or more underlying tables. Because views do not physically store data, they are referred to as “virtual” tables. Unlike tables, views do not physically contain or store rows of data. Views, however, do have a physical presence and take up space. Storage demands for views are minimal because the only portion saved is the SELECT statement or query itself. Tables, on the other hand, store one or more rows of data and their attributes within their structure.

Views are created with the CREATE VIEW statement while tables are created with the CREATE TABLE statement. Because you use one or more underlying tables to create a virtual (derived) table, views provide you with a powerful method for accessing data sources.

Although views have many unique and powerful features, they also have pitfalls. First, views generally take longer to process than tables. Each time a view is referenced, the current underlying table or tables are accessed and processed. Because a view is not physically materialized until it is accessed, higher utilization costs can be expected, particularly for

larger views. Also, because a view is not a data file (table) and does not contain data, an index cannot be created for a view. However, if a view is created from a data file that has a defined index, then the SQL optimizer might attempt to use the index when processing its associated WHERE clause expression. Finally, if the results of a view are used several times in the same program, it might be more efficient to save and reuse a copy of the results in a table, thereby avoiding the re-execution (materialization) of the view multiple times.

Types of Views

Views can be designed to achieve a number of objectives:

- Reference a single table
- Produce summary data across a row
- Conceal sensitive information
- Create updatable views
- Grouped data based on summary functions or a HAVING clause
- Use set operators
- Combine two or more tables in join operations
- Nest one view within another view

As a way to distinguish the various types of views, Joe Celko introduced a classification system that is based on the type of SELECT statement used. See *SQL for Smarties: Advanced SQL Programming* (Morgan Kaufman, 1999).

To help you understand the different view types, this chapter describes and illustrates view construction as well as how they can be used. A view can also have the characteristics of one or more view types, thereby being classified as a hybrid. A hybrid view, for example, could be designed to reference two or more tables, perform updates, and contain complex computations. Table 8.1 presents the different view types along with a brief description of their purpose.

Table 8.1: A Description of the Various View Types

Types of Views	Description
Single-table view	A single-table view references a single underlying (base) table. It is the most common type of view. Selected columns and rows can be displayed or hidden depending on need.
Calculated column views	A calculated column view provides summary data across a row.
Read-only view	A read-only view prevents data from being updated (as opposed to updatable views) and is used to display data only. This also serves security purposes for the concealment of sensitive information.
Updatable view	An updatable view adds (inserts), modifies, or deletes rows of data.

Types of Views	Description
Grouped view	A grouped view uses query expressions that are based on a query with a GROUP BY clause.
Set operation view	A set operation view includes the union of two tables, the removal of duplicate rows, the concatenation of results, and the comparison of query results.
Joined view	A joined view is based on the joining of two or more base tables. This type of view is often used in table-lookup operations to expand (or translate) coded data into text.
Nested view	A nested view is based on one view being dependent on another view such as with subqueries.
Hybrid view	An integration of one or more view types for the purpose of handling more complex tasks.

Creating Views

You use the CREATE VIEW statement in the SQL procedure to create a view. When the SQL processor reads the words CREATE VIEW, it expects to find a name assigned to the newly created view. The SELECT statement defines the names assigned to the view's columns as well as their order.

Views are often constructed so that the order of the columns is different from the base table. In the next example, a view is created with the columns appearing in a different order from the original MANUFACTURERS base table. The view's SELECT statement does not execute during this step because its only purpose is to define the view in the CREATE VIEW statement.

SQL Code

```
PROC SQL;
  CREATE VIEW MANUFACTURERS_VIEW AS
    SELECT manuname, manunum, manucity, manustat
      FROM MANUFACTURERS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE VIEW MANUFACTURERS_VIEW AS
    SELECT manuname, manunum, manucity, manustat
      FROM MANUFACTURERS;
NOTE: SQL view WORK.MANUFACTURERS_VIEW has been defined.
      QUIT;

NOTE: PROCEDURE SQL used:
      real time           0.44 seconds
```

When you create a view, you can create columns that are not present in the base table from which you built your view. That is, you can create columns that are the result of an operation (addition, subtraction, multiplication, etc.) on one or more columns in the base tables. You can also build a view using one or more unmodified columns of one or more base tables. Columns created this way are referred to as *derived columns* or *calculated columns*. In the next example, suppose that you want to create a view that consists of the product name, inventory quantity, and inventory cost from the INVENTORY base table, and a derived column of average product costs that are stored in inventory.

SQL Code

```
PROC SQL;
  CREATE VIEW INVENTORY_VIEW AS
    SELECT prodnum, invenqty, invencst,
           invencst/invenqty AS AverageAmount
      FROM INVENTORY;
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE VIEW INVENTORY_VIEW AS
    SELECT prodnum, invenqty, invencst,
           invencst/invenqty AS AverageAmount
      FROM INVENTORY;
NOTE: SQL view WORK.INVENTORY_VIEW has been defined.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

Displaying a View's Contents

You would expect the CONTENTS procedure to display information about the physical characteristics of a SAS data library and its tables. But what you might not know is that the CONTENTS procedure can also be used to display information about a view. The output that is generated from the CONTENTS procedure shows that the view contains no rows (observations) by displaying a missing value in the Observations field and a member type of VIEW. The engine used is the SQLVIEW. The following example illustrates the use of the CONTENTS procedure in the Windows environment to display the INVENTORY_VIEW view's contents.

SQL Code

```
PROC CONTENTS DATA=INVENTORY_VIEW;
RUN;
```

SAS Output Results

The CONTENTS Procedure					
Data Set Name	WORK.INVENTORY_VIEW			Observations	.
Member Type	VIEW			Variables	4
Engine	SQLVIEW			Indexes	0
Created	Wed, Sep 04, 2013 02:15:38 AM			Observation Length	32
Last Modified	Wed, Sep 04, 2013 02:15:38 AM			Deleted Observations	0
Protection				Compressed	NO
Data Set Type				Sorted	NO
Label					
Data Representation	Default				
Encoding	Default				

Alphabetic List of Variables and Attributes						
#	Variable	Type	Len	Flags	Format	Label
4	AverageAmount	Num	8	P-		
3	invencst	Num	6	-C-	DOLLAR10.2	Inventory Cost
2	invenqty	Num	3	-C-		Inventory Quantity
1	prodnum	Num	3	--		Product Number

Describing View Definitions

Because views consist of partially compiled executable statements, ordinarily you would not be able to read the code in a view definition. However, the SQL procedure provides a statement to inspect the contents of the executable instructions (stored query expression) that are contained within a view definition. Without this capability, a view's underlying instructions (PROC SQL code) would forever remain a mystery and would make the ability to modify or customize the query expressions next to impossible. Whether your job is to maintain or customize a view, the DESCRIBE VIEW statement is the way you review the statements that make up a view. Let's look at how a view definition is described.

The next example shows the DESCRIBE VIEW statement being used to display the INVENTORY_VIEW view's instructions. It should be noted that results are displayed in the SAS log and not in the Output window.

SQL Code

```
PROC SQL;
  DESCRIBE VIEW INVENTORY_VIEW;
QUIT;
```

SAS Log Results

```

PROC SQL;
  DESCRIBE VIEW INVENTORY_VIEW;
NOTE: SQL view WORK.INVENTORY_VIEW is defined as:
      select prodnum, invenqty, invencst,
             invencst/invenqty as CostQty_Ratio
      from INVENTORY;
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.05 seconds

```

Creating and Using Views in SAS

Views are accessed the same way that tables are accessed. The SQL procedure permits views to be used in SELECT queries, subsets, joins, other views, and DATA and PROC steps. Views can reference other views (as will be described in more detail in a later section of this chapter), but the referenced views must ultimately reference one or more existing base tables.

The only thing that cannot be done is to create a view from a table or view that does not already exist. When this is attempted, an error message is written in the SAS log that indicates that the view is being referenced recursively. An error occurs because the view that is being referenced directly (or indirectly) by it cannot be located or opened successfully. The next example shows the error that occurs when a view called NO_CAN_DO_VIEW is created from a non-existing view by the same name in a SELECT statement FROM clause.

SQL Code

```

PROC SQL;
  CREATE VIEW NO_CAN_DO_VIEW AS
    SELECT *
      FROM NO_CAN_DO_VIEW;
  SELECT *
    FROM NO_CAN_DO_VIEW;
QUIT;

```

SAS Log Results

```

PROC SQL;
  CREATE VIEW NO_CAN_DO_VIEW AS
    SELECT *
      FROM NO_CAN_DO_VIEW;
NOTE: SQL view WORK.NO_CAN_DO_VIEW has been defined.
  SELECT *
    FROM NO_CAN_DO_VIEW;
ERROR: The SQL View WORK.NO_CAN_DO_VIEW is referenced recursively.
  QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.00 seconds
      cpu time           0.01 seconds

```

Views and SAS Procedures

In most cases but not all, views can be used just as input SAS data sets to the universe of available SAS procedures. In the first example, the INVENTORY_VIEW view is used as input to the MEANS procedure to produce simple univariate descriptive statistics for numeric variables. Accessing the INVENTORY_VIEW view is different from accessing the INVENTORY table because the view's internal compiled executable statements are processed providing current data from the underlying table to the view itself. The view statements and the statements and options from the MEANS procedure determine what information is produced.

The next example uses the INVENTORY_VIEW view as input to the MEANS procedure to produce simple univariate descriptive statistics for numeric variables. Accessing the INVENTORY_VIEW view is different from accessing the INVENTORY table because the view derives and provides current data from the underlying table to the view itself. The view statements and the statements and options from the MEANS procedure determine what information is produced.

SAS Code

```
PROC MEANS DATA=INVENTORY_VIEW;
  TITLE1 'Inventory Statistical Report';
  TITLE2 'Demonstration of a View used in PROC MEANS';
RUN;
```

SAS Log Results

```
PROC MEANS DATA=INVENTORY_VIEW;
  TITLE1 'Inventory Statistical Report';
  TITLE2 'Demonstration of a View used in PROC MEANS';
  RUN;
NOTE: There were 7 observations read from the dataset
WORK.INVENTORY.
NOTE: There were 7 observations read from the dataset
WORK.INVENTORY_VIEW.
NOTE: PROCEDURE MEANS used:
      real time          0.32 seconds
```

Results

Inventory Statistical Report Demonstration of a View used in PROC MEANS

The MEANS Procedure

Variable	Label	N	Mean	Std Dev	Minimum	Maximum
prodnum	Product Number	7	3974.43	1763.50	1110.00	5004.00
invenqty	Inventory Quantity	7	10.0000000	7.5055535	2.0000000	20.0000000
invencst	Inventory Cost	7	11357.14	17866.72	900.0000000	45000.00
AverageAmount		7	917.1428571	1121.71	70.0000000	2800.00

The next example uses the INVENTORY_VIEW view as input to the PRINT procedure to produce a detailed listing of the values that are contained in the underlying base table.

Note: It is worth noting that, as with all procedures, all procedure options and statements are available by views.

SAS Code

```
PROC PRINT DATA=INVENTORY_VIEW N NOOBS UNIFORM;
  TITLE1 'Inventory Detail Listing';
  TITLE2 'Demonstration of a View used in PROC PRINT';
  format AverageAmount dollar10.2;
RUN;
```

SAS Log Results

```
PROC PRINT DATA=INVENTORY_VIEW N NOOBS UNIFORM;
  TITLE1 'Inventory Detail Listing';
  TITLE2 'Demonstration of a View used in a Procedure';
  format AverageAmount dollar10.2;
RUN;
NOTE: There were 7 observations read from the dataset
WORK.INVENTORY.
NOTE: There were 7 observations read from the dataset
WORK.INVENTORY_VIEW.
NOTE: PROCEDURE PRINT used:
      real time          0.04 seconds
```

Results

Inventory Detail Listing			
Demonstration of a View used in PROC PRINT			
prodnum	invenqty	invencst	AverageAmount
1110	20	\$45,000.00	\$2,250.00
1700	10	\$28,000.00	\$2,800.00
5001	5	\$1,000.00	\$200.00
5002	3	\$900.00	\$300.00
5003	10	\$2,000.00	\$200.00
5004	20	\$1,400.00	\$70.00
5001	2	\$1,200.00	\$600.00
N = 7			

Views and DATA Steps

As you have already seen, views can be used as input to SAS procedures as if they were data sets. You will now see that views are a versatile component that can be used in a DATA step as well. This gives you a controlled way of using views to access tables of data in custom

report programs. The next example uses the INVENTORY_VIEW view as input to the DATA step as if it were a SAS base table. Notice that the KEEP= data set option reads only two of the variables from the INVENTORY_VIEW view.

SAS Code

```
OPTIONS FORMCHAR="|---|+|---+=|-/\<>*";
DATA _NULL_;
  SET INVENTORY_VIEW (KEEP=PRODNUM AVERAGEAMOUNT);
  FILE PRINT HEADER=H1;
  PUT @10 PRODNUM
    @30 AVERAGEAMOUNT DOLLAR10.2;
RETURN;
H1: PUT @9 'Using a View in a DATA Step'
  /// @5 'Product Number'
  @26 'Average Amount';
RETURN;
RUN;
```

SAS Log Results

```
OPTIONS FORMCHAR="|---|+|---+=|-/\<>*";
DATA _NULL_;
  SET INVENTORY_VIEW (KEEP=PRODNUM AVERAGEAMOUNT);
  FILE PRINT HEADER=H1;
  PUT @10 PRODNUM
    @30 AVERAGEAMOUNT DOLLAR10.2;
RETURN;
H1: PUT @9 'Using a View in a DATA Step'
  /// @5 'Product Number'
  @26 'Average Amount';
RETURN;
RUN;

NOTE: 11 lines were written to file PRINT.
NOTE: There were 7 observations read from the dataset
WORK.INVENTORY.
NOTE: There were 7 observations read from the dataset
      WORK.INVENTORY_VIEW.
NOTE: DATA statement used:
      real time          0.00 seconds
```

Output

Using a View in a DATA Step

Product Number	Average Amount
1110	\$2,250.00
1700	\$2,800.00
5001	\$200.00
5002	\$300.00
5003	\$200.00
5004	\$70.00
5001	\$600.00

Eliminating Redundancy

Data redundancy commonly occurs when two or more users want to see the same data in different ways. To prevent redundancy, organizations should create and maintain a master database environment rather than propagate one or more subsets of data among its user communities. The latter approach creates an environment that is not only problematic for the organization but for its users and customers.

Views provide a way to eliminate or, at least, reduce the degree of data redundancy. Rather than having the same data exist in multiple forms, views create a virtual and shareable database environment for all. Problems that are related to accessing and reporting outdated information, as well as table and program change control, are eliminated.

Restricting Data Access—Security

As data security issues grow increasingly important for organizations around the globe, views offer a powerful alternative in controlling or restricting access to sensitive information. Views, like tables, can prevent unauthorized users from accessing sensitive portions of data. This is important because security breaches pose great risks not only to an organization's data resources but to the customer as well.

Views can be constructed to show a view of data that is different from what physically exists in the underlying base tables. Specific columns can be shown while other columns are hidden. This helps prevent sensitive information such as salary, medical, or credit card data from getting into the wrong hands. Or a view can contain a WHERE clause with any degree of complexity to restrict what rows appear for a group of users while hiding other rows. In the next example, a view called SOFTWARE_PRODUCTS_VIEW is created that displays all columns from the original table except the product cost (PRODCOST) column, and restricts all rows except "Software" from the PRODUCTS table.

SQL Code

```
PROC SQL;
  CREATE VIEW SOFTWARE_PRODUCTS_VIEW AS
    SELECT prodnum, prodname, manunum, prodtype
      FROM PRODUCTS
        WHERE UPCASE (PRODTYPE) IN ('SOFTWARE');
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE VIEW SOFTWARE_PRODUCTS_VIEW AS
    SELECT prodnum, prodname, manunum, prodtype
      FROM PRODUCTS
        WHERE UPCASE (PRODTYPE) IN ('SOFTWARE');
NOTE: SQL view WORK.SOFTWARE_PRODUCTS_VIEW has been defined.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.17 seconds
```

The SOFTWARE_PRODUCTS_VIEW view functions just as if it were a base table, although it contains no rows of data. All of the other columns with the exception of the product cost (PRODCOST) column are inherited from the selected columns in the PRODUCTS table. A view determines what columns and rows are processed from the underlying table and, optionally, the SELECT query that references the view can provide additional criteria during processing. In the next example, the view SOFTWARE_PRODUCTS_VIEW is referenced in a SELECT query and arranged in ascending order by product name (PRODNAME).

SQL Code

```
PROC SQL;
  SELECT *
  FROM SOFTWARE_PRODUCTS_VIEW
  ORDER BY prodname;
QUIT;
```

Results

Product Number	Product Name	Manufacturer Number	Product Type
5002	Database Software	500	Software
5004	Graphics Software	500	Software
5001	Spreadsheet Software	500	Software
5003	Wordprocessor Software	500	Software

Hiding Logic Complexities

Because complex logic constructs such as multi-way table joins, subqueries, or hard-to-understand data relationships might be beyond the skill of other staff in your area, you might want to build or customize views so that others can access the information easily. The next example illustrates how a complex query that contains a two-way join is constructed and saved as a view to simplify its use by other users.

SQL Code

```
PROC SQL;
  CREATE VIEW PROD_MANF_VIEW AS
    SELECT DISTINCT SUM(prodcost) FORMAT=DOLLAR10.2,
      M.manunum,
      M.manuname
    FROM PRODUCTS AS P, MANUFACTURERS AS M
    WHERE P.manunum = M.manunum AND
      M.manuname = 'KPL Enterprises';
QUIT;
```

SAS Log Results

```

PROC SQL;
  CREATE VIEW PROD_MANF_VIEW AS
    SELECT DISTINCT SUM(prodcost) FORMAT=DOLLAR10.2,
      M.manunum,
      M.manuname
    FROM PRODUCTS AS P, MANUFACTURERS AS M
    WHERE P.manunum = M.manunum AND
      M.manuname = 'KPL Enterprises';
NOTE: SQL view WORK.PROD_MANF_VIEW has been defined.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

```

In the next example, the PROD_MANF_VIEW is simply referenced in a SELECT query. Because the view's SELECT statement references the product cost (PRODCOST) column with a summary function but does not contain a GROUP BY clause, the note "The query requires remerging summary statistics back with the original data." appears in the SAS log. This situation causes the sum to be calculated and then remerged with each row in the tables that are being processed.

SQL Code

```

PROC SQL;
  SELECT *
  FROM PROD_MANF_VIEW;
QUIT;

```

SAS Log Results

```

PROC SQL;
  SELECT *
  FROM PROD_MANF_VIEW;
NOTE: The query requires remerging summary statistics back with
the
original data.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.05 seconds

```

Results

	Manufacturer Number	Manufacturer Name
\$1,296.00	500	KPL Enterprises

Nesting Views

An important feature of views is that they can be based on other views. This is called *nesting*. One view can access data that comes through another view. In fact, there isn't a limit to the number of view layers that can be defined. Because of this, views can be a very convenient and flexible way for programmers to retrieve information. Although the number of views that can be nested is virtually unlimited, programmers should use care to avoid nesting views too deeply. Performance-related and maintenance-related issues can result, especially if the views are built many layers deep.

To see how views can be based on other views, two views will be created—one view that references the PRODUCTS table and the other view that references the INVOICE table. In the first example, WORKSTATION_PRODUCTS_VIEW includes only products that are related to workstations and excludes the manufacturer number. The view produces the result displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW WORKSTATION_PRODUCTS_VIEW AS
    SELECT PRODNUM, PRODNAME, PRODTYPE, PRODCOST
      FROM PRODUCTS
        WHERE UPCASE (PRODTYPE) = "WORKSTATION";
QUIT;
```

Results

	Product Number	Product Name	Product Type	Product Cost
1	1110	Dream Machine	Workstation	\$3,200.00
2	1200	Business Machine	Workstation	\$3,300.00

In the next example, INVOICE_1K_VIEW includes rows where the invoice price is \$1,000.00 or greater and excludes the manufacturer number. When accessed from the SAS Windowing environment, the view renders the results displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW INVOICE_1K_VIEW AS
    SELECT INVNUM, CUSTNUM, PRODNUM, INVQTY, INVPRICE
      FROM INVOICE
        WHERE INVPRICE >= 1000.00;
QUIT;
```

Results

	Invoice Number	Customer Number	Product Number	Invoice Quantity - Units Sold	Invoice Unit Price
1	1001	201	5001	5	\$1,495.00
2	1002	1301	6001	2	\$1,598.00
3	1004	501	1110	3	\$9,600.00
4	1007	401	1200	7	\$23,100.00

The next example illustrates how to create a view from the join of the WORKSTATION_PRODUCTS_VIEW and INVOICE_1K_VIEW views. The resulting view is nested two layers deep. When accessed from the SAS Windowing environment, the view renders the results displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW JOINED_VIEW AS
    SELECT V1.PRODNUM, V1.PRODNAME,
           V2.CUSTNUM, V2.INVQTY, V2.INVPRICE
      FROM WORKSTATION_PRODUCTS_VIEW  V1,
           INVOICE_1K_VIEW      V2
     WHERE V1.PRODNUM = V2.PRODNUM;
QUIT;
```

Results

	Product Number	Product Name	Customer Number	Invoice Quantity - Units Sold	Invoice Unit Price
1	1110	Dream Machine	501	3	\$9,600.00
2	1200	Business Machine	401	7	\$23,100.00

In the next example, a third layer of view is nested to the previous view in order to find the largest invoice amount. In the next example, a view is constructed to find the largest invoice amount using the MAX summary function to compute the product of the invoice price (INVPRICE) and invoice quantity (INVQTY) from the JOINED_VIEW view.

When accessed from the SAS Windowing environment, the view produces the results displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW LARGEST_AMOUNT_VIEW AS
    SELECT MAX(INVPRICE*INVQTY) AS Maximum_Price
      FORMAT=DOLLAR12.2
      LABEL="Largest Invoice Amount"
     FROM JOINED_VIEW;
QUIT;
```

Results

	Largest Invoice Amount
1	\$161,700.00

Updatable Views

Once a view has been created from a physical table, it can then be used to modify the view's data of a single underlying table. Essentially, when a view is updated the changes pass through the view to the underlying base table. A view that is designed in this manner is called an *updatable view* and can have INSERT, UPDATE, and DELETE operations performed into the single table from which it's constructed.

Because views are dependent on getting their data from a base table and have no physical existence of their own, you should exercise care when constructing an updatable view. Although useful in modifying the rows in a table, updatable views do have a few limitations that programmers and users should be aware of.

- An updatable view can have only a single base table associated with it. This means that the underlying table cannot be used in a join operation or with any set operators. Because an updatable view has each of its rows associated with only a single row in an underlying table, any operations that involve two or more tables will produce an error and result in update operations not being performed.
- An updatable view cannot contain a subquery. A subquery is a complex query that consists of a SELECT statement that is contained inside another statement. This violates the rules for updatable views and is not allowed.
- An updatable view can update a column using a view's column alias, but cannot contain the DISTINCT keyword, have any aggregate (summary) functions, calculated columns, or derived columns associated with it. Because these columns are produced by an expression, they are not allowed.
- An updatable view can contain a WHERE clause but cannot contain other clauses such as ORDER BY, GROUP BY, or HAVING.

In the remaining sections, three types of updatable views will be examined:

- Views that insert one or more rows of data
- Views that update existing rows of data
- Views that delete one or more rows of data from a single underlying table

Inserting New Rows of Data

You can add or insert new rows of data in a view using the INSERT INTO statement.

Suppose that you have a view that consists of only software products, called SOFTWARE_PRODUCTS_VIEW. The PROC SQL code that is used to create this view consists of a SELECT statement with a WHERE clause. There are four defined columns:

product number, product name, product type, and product cost, in that order. When accessed from the SAS Windowing environment, the view produces the results displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW SOFTWARE_PRODUCTS_VIEW AS
    SELECT prodnum, prodname, prodtype, prodcost
      FORMAT=DOLLAR8.2
    FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) IN ('SOFTWARE');
QUIT;
```

Results

	Product Number	Product Name	Product Type	Product Cost
1	5001	Spreadsheet Software	Software	\$299.00
2	5002	Database Software	Software	\$399.00
3	5003	Wordprocessor Software	Software	\$299.00
4	5004	Graphics Software	Software	\$299.00

Suppose that you want to add a new row of data to this view. This can be accomplished by specifying the corresponding values in a VALUES clause as follows.

SQL Code

```
PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS_VIEW
    VALUES(6002,'Security Software','Software',375.00);
QUIT;
```

As seen from the view results, the INSERT INTO statement added the new row of data corresponding to the WHERE logic in the view. The view contains the new row and consists of the value 6002 in product number, “Security Software” in product name, “Software” in product type, and \$375.00 in product cost.

View Results

	Product Number	Product Name	Product Type	Product Cost
1	5001	Spreadsheet Software	Software	\$299.00
2	5002	Database Software	Software	\$399.00
3	5003	Wordprocessor Software	Software	\$299.00
4	5004	Graphics Software	Software	\$299.00
5	6002	Security Software	Software	\$375.00

As depicted in the table results, the new row of data was added to the PRODUCTS table using the view called SOFTWARE_PRODUCTS_VIEW. The new row in the PRODUCTS table contains the value 6002 in product number, “Security Software” in product name, “Software” in product type, and \$375.00 in product cost. The manufacturer number column is assigned a null value (missing value).

Table Results

	Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1	1110	Dream Machine	111	Workstation	\$3,200.00
2	1200	Business Machine	120	Workstation	\$3,300.00
3	1700	Travel Laptop	170	Laptop	\$3,400.00
4	2101	Analog Cell Phone	210	Phone	\$35.00
5	2102	Digital Cell Phone	210	Phone	\$175.00
6	2200	Office Phone	220	Phone	\$130.00
7	5001	Spreadsheet Software	500	Software	\$299.00
8	5002	Database Software	500	Software	\$399.00
9	5003	Wordprocessor Software	500	Software	\$299.00
10	5004	Graphics Software	500	Software	\$299.00
11	6002	Security Software	600	Software	\$375.00

Now, let's see what happens when a row of data is added through a view that does not meet the condition(s) in the WHERE clause in the view. Suppose that you want to add a row of data that contains the value 1701 for product number, "Travel Laptop SE" in product name, "Laptop" in product type, and \$4200.00 in product cost in the SOFTWARE_PRODUCTS_VIEW view.

SQL Code

```
PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS_VIEW
    VALUES (1701, 'Travel Laptop SE', 'Laptop', 4200.00);
QUIT;
```

Because the new row's value for product type is "Laptop", this value violates the WHERE clause condition when the view SOFTWARE_PRODUCTS_VIEW was created. As a result, the new row of data is rejected and is not added to the table PRODUCTS. The SQL procedure also prevents the new row from appearing in the view because the base table controls what the view contains.

The updatable view does exactly what it is designed to do—that is, it validate each new row of data as each row is added to the base table. Whenever the WHERE clause condition is violated, the view automatically rejects the row as invalid and restores the table to its pre-updated state by rejecting the row in error and deleting all successful inserts before the error occurred. In this example, the following error message was issued to the SAS log to confirm that the view was restored to its original state before the update took place.

SAS Log Results

```

PROC SQL;
  INSERT INTO PRODUCTS_VIEW
    VALUES(1701,'Travel Laptop SE','Laptop',4200.00);
ERROR: The new values do not satisfy the view's where
expression. This
update or add is not allowed.
NOTE: This insert failed while attempting to add data from
VALUES
 clause 1 to the dataset.
NOTE: Deleting the successful inserts before error noted above
to
 restore table to a consistent state.
      QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.04 seconds

```

Views will not accept new rows added to a base table when the number of columns in the VALUES clause does not match the number of columns defined in the view, unless the columns that are being inserted are specified. In the next example, a partial list of columns for a row of data is inserted with a VALUES clause. Because the inserted row of data does not contain a value for product cost, the new row will not be added to the PRODUCTS table. The resulting error message indicates that the VALUES clause has fewer columns specified than exist in the view itself, as shown in the SAS log.

SQL Code

```

PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS_VIEW
    VALUES(6003,'Cleanup Software','Software');
QUIT;

```

SAS Log Results

```

PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS_VIEW
    VALUES(6003,'Cleanup Software','Software');
ERROR: VALUES clause 1 attempts to insert fewer columns than
specified
 after the INSERT table name.
      QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

```

Suppose that a view called SOFTWARE_PRODUCTS_TAX_VIEW was created with the sole purpose of deriving each software product's sales tax amount.

SQL Code

```
PROC SQL;
  CREATE VIEW SOFTWARE_PRODUCTS_TAX_VIEW AS
    SELECT prodnum, prodname, prodtype, prodcost,
           prodcost * .07 AS Tax
           FORMAT=DOLLAR8.2 LABEL='Sales Tax'
      FROM PRODUCTS
     WHERE UPCASE(PRODTYPE) IN ('SOFTWARE');
QUIT;
```

In the next example, an attempt is made to add a new row through the SOFTWARE_PRODUCTS_TAX_VIEW view by inserting a VALUES clause with all columns defined. The row is rejected and an error is produced because an update was attempted against a view that contains a computed (calculated) column. Although the VALUES clause contains values for all columns defined in the view, the reason the row is not inserted into the PRODUCTS table is because of the reference to a computed (or derived) column TAX (Sales Tax) as shown in the SAS log results.

SQL Code

```
PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS_TAX_VIEW
    VALUES(6003,'Cleanup Software','Software',375.00,26.25);
QUIT;
```

SAS Log Results

```
PROC SQL;
  INSERT INTO SOFTWARE_PRODUCTS_TAX_VIEW
    VALUES(6003,'Cleanup Software','Software',375.00,26.25);
WARNING: Cannot provide Tax with a value because it references a
derived column that can't be inserted into.
  QUIT;
NOTE: The SAS System stopped processing this step because of
errors.
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds
```

Updating Existing Rows of Data

The SQL procedure permits rows to be updated through a view. The data manipulation language statement that is specified to modify existing data in PROC SQL is the UPDATE statement. Suppose that you want to create a view to select only laptops from the PRODUCTS table. The SQL procedure code that is used to create the view is called LAPTOP_PRODUCTS_VIEW, and it consists of a SELECT statement with a WHERE clause. There are four defined columns: product number, product name, product type, and product cost, in that specific order. When accessed, the view produces the results displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW LAPTOP_PRODUCTS_VIEW AS
    SELECT PRODNUM, PRODNAME, PRODTYPE, PRODCOST
      FROM PRODUCTS
     WHERE UPCASE(PRODTYPE) = 'LAPTOP';
QUIT;
```

Results

	Product Number	Product Name	Product Type	Product Cost
1	1700	Travel Laptop	Laptop	\$3,400.00

In the next example, all laptops are to be discounted by twenty percent and the new price is to take effect immediately. The changes that are applied through the LAPTOP_PRODUCTS_VIEW view compute the discounted product cost for “Laptop” computers in the PRODUCTS table using an UPDATE statement with corresponding SET clause.

SQL Code

```
PROC SQL;
  UPDATE LAPTOP_PRODUCTS_VIEW
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2);
QUIT;
```

SAS Log Results

```
PROC SQL;
  UPDATE LAPTOP_DISCOUNT_VIEW
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2);
NOTE: 1 row was updated in WORK.LAPTOP_DISCOUNT_VIEW.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.04 seconds
```

Results

	Product Number	Product Name	Product Type	Product Cost
1	1700	Travel Laptop	Laptop	\$2,720.00

Sometimes updates that are applied through a view can change the rows of data in the base table so that once the update is performed the rows in the base table no longer meet the criteria in the view. When this occurs, the changed rows of data cannot be displayed by the view. Essentially, the updated rows that match the conditions in the WHERE clause no longer match the conditions in the view’s WHERE clause after the updates are made. As a result, the view updates the rows with the specified changes, but it is no longer able to display the rows of data that were changed.

Suppose that you want to create a view to select laptops that cost less than \$2,800.00 from the PRODUCTS table. The SQL procedure code that is used to create the view called LAPTOP_DISCOUNT_VIEW consists of a SELECT statement with a WHERE clause. There are four defined columns: product number, product name, product type, and product cost, in that order. When accessed, the view produces the results displayed below.

SQL Code

```
PROC SQL;
  CREATE VIEW LAPTOP_DISCOUNT_VIEW AS
    SELECT PRODNUM, PRODNAME, PRODTYPE, PRODCOST
      FROM PRODUCTS
     WHERE UPCASE(PRODTYPE) = 'LAPTOP' AND
          PRODCOST < 2800.00;
QUIT;
```

Results

	Product Number	Product Name	Product Type	Product Cost
1	1700	Travel Laptop	Laptop	\$2,720.00

The next example illustrates how updates are applied through a view in the Windows environment so that the rows in the table no longer meet the view's criteria. Suppose that a fifteen percent surcharge is applied to all laptops. An UPDATE statement and SET clause are specified to allow the rows in the PRODUCTS table to be updated through the view. Once the update is performed and the view is accessed, a dialog box appears that indicates that no rows are available to display because the data from the PRODUCTS table no longer meets the view's WHERE clause expression.

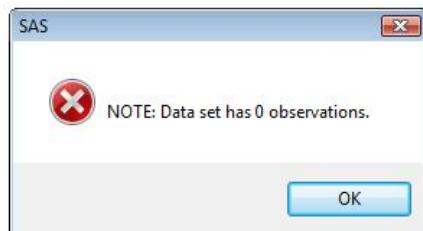
SQL Code

```
PROC SQL;
  UPDATE LAPTOP_DISCOUNT_VIEW
    SET PRODCOST = PRODCOST + (PRODCOST * 0.15);
QUIT;
```

SAS Log Results

```
PROC SQL;
  UPDATE LAPTOP_DISCOUNT_VIEW
    SET PRODCOST = PRODCOST + (PRODCOST * 0.15);
NOTE: 1 row was updated in WORK.LAPTOP_DISCOUNT_VIEW.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.06 seconds
```

Results



Deleting Rows of Data

Now that you have seen how updatable views can add or modify one or more rows of data, you might have a pretty good idea how to create an updatable view that deletes one or more rows of data. Consider the following updatable view that deletes manufacturers whose manufacturer number is 600 from the underlying PRODUCTS table.

SQL Code

```
PROC SQL;
  DELETE FROM SOFTWARE_PRODUCTS_VIEW
  WHERE MANUNUM=600;
QUIT;
```

SAS Log Results

```
PROC SQL;
  DELETE FROM SOFTWARE_PRODUCTS_VIEW
  WHERE MANUNUM=600;
NOTE: 2 rows were deleted from WORK.SOFTWARE_PRODUCTS_VIEW.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.04 seconds
```

Deleting Views

When a view is no longer needed, it's nice to know that there is a way to remove it. Without this ability, program maintenance activities would be more difficult. To remove an unwanted view, specify the DROP VIEW statement and the name of the view. In the next example, the INVENTORY_VIEW view is deleted from the WORK library.

SQL Code

```
PROC SQL;
  DROP VIEW INVENTORY_VIEW;
QUIT;
```

SAS Log

```

PROC SQL;
  DROP VIEW INVENTORY_VIEW;
NOTE: View WORK.INVENTORY_VIEW has been dropped.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.10 seconds

```

When more than a single view needs to be deleted, the DROP VIEW statement works equally as well. Specify a comma between each view name when deleting two or more views.

SQL Code

```

PROC SQL;
  DROP VIEW INVENTORY_VIEW, LAPTOP_PRODUCTS_VIEW;
QUIT;

```

SAS Log

```

PROC SQL;
  DROP VIEW INVENTORY_VIEW, LAPTOP_PRODUCTS_VIEW;
NOTE: View WORK.INVENTORY_VIEW has been dropped.
NOTE: View WORK.LAPTOP_PRODUCTS_VIEW has been dropped.
QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.00 seconds

```

Summary

1. Views are not tables and consequently do not store data (see the “Views—Windows to Your Data” section).
2. Views access one or more underlying tables (base tables) or other views (see the “Views—Windows to Your Data” section)
3. Views improve the change control process when constructed as a “common” set of routines (see the “Views—Windows to Your Data” section).
4. Views eliminate or reduce data redundancy (see the “Eliminating Redundancy” section).
5. Views hide unwanted or sensitive information while displaying specific columns and/or rows (see the “Restricting Data Access—Security” section).
6. Views shield users from making logic and/or data errors (see the “Hiding Logic Complexities” section).
7. Nesting views too deeply can produce unnecessary confusion and maintenance difficulties (see the “Nesting Views” section).
8. Updatable views add, modify, or delete rows of data (see the “Updatable Views” section).
9. Views can be deleted when no longer needed (see the “Deleting Views” section).

Chapter 9: Fuzzy Matching Programming

Introduction	313
Data Sets Used in Examples	314
6-Step Fuzzy Matching Process	316
Determine Matching Variables	317
Understand Data Values Distribution.....	318
Data Cleaning.....	323
Data Transformations.....	326
Exact Matching Process	327
Fuzzy Matching Processing.....	328
Summary	342

Introduction

Data comes in all forms, shapes, sizes, and complexities. Stored in files and data sets, SAS users across industries know all too well that data can be, and often is, problematic and plagued with a variety of issues. When unique and reliable identifiers are available, users routinely are able to match records from two or more data sets using merge, join, and/or hash programming techniques without problem. But, when data originating from multiple sources contain duplicate observations, duplicate and/or unreliable keys, missing values, invalid values, capitalization and punctuation issues, inconsistent matching variables, and imprecise text identifiers, the matching process is often compromised. These types of problems are common and are often found in files and data sets containing a misspelled customer name, mailing address, or email address, where one or more characters are transposed or incorrectly recorded.

When data issues like these exist, SAS users should do everything possible to identify and standardize any and all data irregularities before attempting to search, match, and join data. To assist in this time-consuming and costly process, users often apply special-purpose programming techniques including the application of one or more of the following SAS functions to resolve key identifier issues and to successfully search, merge, and join less than perfect or messy data:

- the family of CAT functions
- various data cleaning techniques
- user-defined validation techniques
- approximate string matching techniques
- an assortment of constructive programming techniques to standardize, combine, and transform data sets together
- the application of the SOUNDEX (for phonetic matching) algorithm

- the SPEDIS, COMPLEV, and COMPGED functions the use of SAS and Perl regular expression functions often offers a more compact solution to complicated string manipulation tasks such as when performing text matching. If you want to learn more, a great resource is Ron Cody's book, *SAS Functions by Example, Second Edition*.

Data Sets Used in Examples

The examples presented in this chapter use three transaction data sets, Customers_with_messy_data, Manufacturers_with_messy_data, and Products_with_messy_data. The internal accounts department is sending us these transaction data sets so that we can add the data and all its content to each of our production data sets: Customers, Manufacturers, and Products. But, before adding the transaction data sets to our production data sets, our organization's protocol requires us to first verify the cleanliness and accuracy of each data set.

After careful inspection, we find that each transaction data set contains data issues, including the existence of spelling errors, punctuation inconsistencies, and invalid values. Our analysis concludes the following information:

- Customers_with_messy_data (Figure 9.1) consists of 3 observations, a data structure of three variables: Custnum, a numeric variable; and Custname and Custcity, character variables. Several data issues are found, including spelling errors, punctuation inconsistencies, and invalid values.
- Manufacturers_with_messy_data (Figure 9.2) contains 2 observations and a data structure consisting of four variables: Manunum, a numeric variable; and Manuname, Manucity, and Manustat, character variables. Several data issues are found including, spelling errors, punctuation inconsistencies, and invalid values.
- Products_with_messy_data (Figure 9.3) contains 4 observations and a data structure consisting of five variables: Prodnum, Manunum, and Prodcost, numeric variables; and Prodname and Prodtype, character variables. Several data issues are found, including spelling errors, punctuation inconsistencies, and invalid values.

Figure 9.1: Customers_with_messy_data data set

Customers_with_messy_data

Customer Number	Customer Name	Customer's Home City
1901	Pacific Beach Metropolis	Pacific Baech
2001	Solana Beach High Tech	Solana Baech
2101	EI Cajon Analytics Center	La Kahone

Figure 9.2: Manufacturers_with_messy_data data set

Manufacturers_with_messy_data			
Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State
800	21st Century Analytics Co	Spring Valley	ca
900	Absolute Best Apps Inc	Spring Valley	XA

Figure 9.3: Products_with_messy_data data set

Products_with_messy_data				
Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
5005	Analytics Software	500	Softwear	\$499.00
5006	Storytelling Software	500	Softwares	\$399.00
5007	Fuzzy Matching Software	500	Softwara	\$399.00
5008	AI Software	500	Softwares	\$399.00

To enable readers to work with these three tables (or data sets), I have included DATA steps so you can copy and paste the code and datalines to recreate the data sets for the purpose of following along with the examples.

Customers Table

```
data Customers;
  input @1 Custnum 4.
    @6 Custname $25.
    @32 Custcity $20.;
  datalines;
1901 Pacific Beach Metropolis  Pacific Baech
2001 Solana Beach High Tech   Solana Baech
2101 El Cajon Analytics Center La Kahone
;
run;
```

Manufacturers Table

```
data Manufacturers;
  input @1 Manunum 4.
    @6 Manuname $25.
    @32 Manucity $20.
    @52 Manustate $2.;

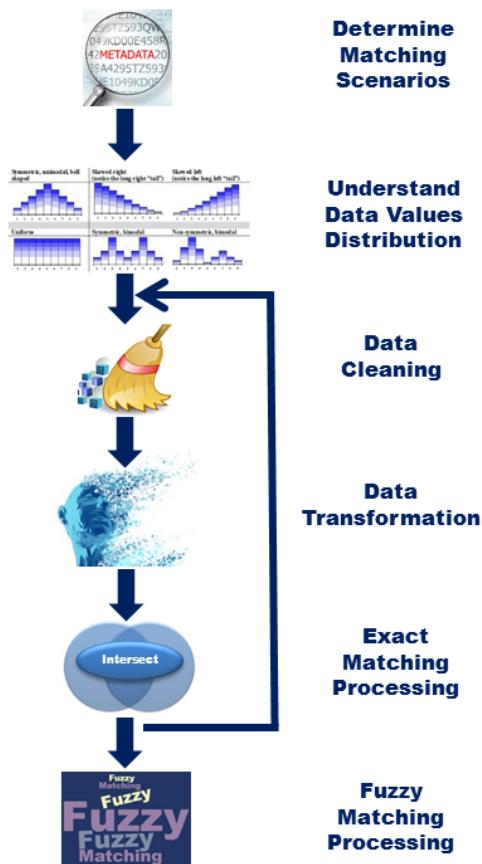
  datalines;
800 21st Century Analytics Co      Spring Valley   ca
900 Absolute Best Apps Inc        Spring Valley   XA
;
run;
proc print data=Manufacturers;
run;
```

Products Table

```
data Products;
  input @1 Prodnum 4.
        @6 Prodname $25.
        @32 Manunum 3.
        @36 Prodtype $15.
        @52 Prodcost 8.2;
  format Prodcost Dollar10.2;
  datalines;
5005 Analytics Software      500 Softwear      49900
5006 Storytelling Software   500 Softwares    39900
5007 Fuzzy Matching Software 500 Softwara     39900
5008 AI Software             500 Softwares    39900
;
run;
proc print data=Products;
run;
```

6-Step Fuzzy Matching Process

If you suspect that your data sets and files contain data issues and/or you want to learn how to prevent data issues such as spelling and punctuation errors, invalid values, and value inconsistencies from creeping into your production data sets, the steps shown in Figure 9.4 should be adhered to.

Figure 9.4: 6-step Fuzzy Matching Process**6-Step Fuzzy Matching Process****Determine Matching Variables**

This first step determines whether any variables exist for matching purposes. Using PROC CONTENTS, PROC DATASETS, or metadata Dictionary tables, the contents of each transaction data set and a sampling of values are examined in greater detail to assess the severity of data issues that exist, as well as the distribution of data values for categorical variables.

PROC CONTENTS Code

```
PROC CONTENTS DATA=Customers_with_Messy_Data;
RUN;
PROC CONTENTS DATA=Manufacturers_with_Messy_Data;
RUN;
PROC CONTENTS DATA=Products_with_Messy_Data;
RUN;
```

Results

The CONTENTS Procedure			
Data Set Name	MYDATA.CUSTOMERS_WITH_MESSY_DATA	Observations	3
Member Type	DATA	Variables	3
Engine	V9	Indexes	0
Created	12/04/2018 01:13:19	Observation Length	48
Last Modified	12/04/2018 01:45:27	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	custcity	Char	20	Customer's Home City
2	custname	Char	25	Customer Name
1	custnum	Num	3	Customer Number

The CONTENTS Procedure			
Data Set Name	MYDATA.MANUFACTURERS_WITH_MESSY_DATA	Observations	2
Member Type	DATA	Variables	4
Engine	V9	Indexes	0
Created	12/04/2018 02:38:48	Observation Length	50
Last Modified	12/04/2018 02:38:48	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Label
3	manucity	Char	20	Manufacturer City
2	manuname	Char	25	Manufacturer Name
1	manunum	Num	3	Manufacturer Number
4	manustat	Char	2	Manufacturer State

The CONTENTS Procedure			
Data Set Name	MYDATA.PRODUCTS_WITH_MESSY_DATA	Observations	4
Member Type	DATA	Variables	5
Engine	V9	Indexes	0
Created	12/04/2018 02:28:21	Observation Length	51
Last Modified	12/04/2018 02:28:21	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
3	manunum	Num	3		Manufacturer Number
5	prodcost	Num	5	DOLLAR9.2	Product Cost
2	prodname	Char	25		Product Name
1	prodnum	Num	3		Product Number
4	prodtype	Char	15		Product Type

Understand Data Values Distribution

To derive a more accurate picture of the data sources, users should conduct some level of data analysis by identifying missing values, outliers, invalid values, minimum and maximum values, averages, value ranges, duplicate observations, distribution of values, and the number of distinct values a categorical variable contains. This important step provides an understanding of the data, while leveraging the data cleaning and standardizing activities that will be performed later. One of the first things data wranglers will want to do is explore the data using the SAS FREQ procedure, or an equivalent approach like Excel Pivot Tables.

PROC FREQ Code

```
PROC FREQ DATA=Customers_with_Messy_Data;
  TABLES _ALL_ / NOCUM NOPERCENT MISSING;
RUN;
```

After reviewing the results, an assortment of data issues is found including data accuracy, inconsistent values, validation issues such as data type and range of values and capitalization versus mixed case, and incomplete (partial) data issues, as shown in the results below.

Results

<code>Customers_with_messy_data</code>	<code>Manufacturers_with_messy_data</code>	<code>Products_with_messy_data</code>																																																																																																												
<p>The FREQ Procedure</p> <table border="1"> <thead> <tr> <th colspan="2">Customer Number</th> </tr> <tr> <th>custnum</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>1901</td> <td>1</td> </tr> <tr> <td>2001</td> <td>1</td> </tr> <tr> <td>2101</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Customer Name</th> </tr> <tr> <th>custname</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>El Cajon Analytics Center</td> <td>1</td> </tr> <tr> <td>Pacific Beach Metropolis</td> <td>1</td> </tr> <tr> <td>Solana Beach High Tech</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Customer's Home City</th> </tr> <tr> <th>custcity</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>La Kahone</td> <td>1</td> </tr> <tr> <td>Pacific Baech</td> <td>1</td> </tr> <tr> <td>Solana Baech</td> <td>1</td> </tr> </tbody> </table>	Customer Number		custnum	Frequency	1901	1	2001	1	2101	1	Customer Name		custname	Frequency	El Cajon Analytics Center	1	Pacific Beach Metropolis	1	Solana Beach High Tech	1	Customer's Home City		custcity	Frequency	La Kahone	1	Pacific Baech	1	Solana Baech	1	<p>The FREQ Procedure</p> <table border="1"> <thead> <tr> <th colspan="2">Manufacturer Number</th> </tr> <tr> <th>manunum</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>800</td> <td>1</td> </tr> <tr> <td>900</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Manufacturer Name</th> </tr> <tr> <th>manuname</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>21st Century Analytics Co</td> <td>1</td> </tr> <tr> <td>Absolute Best Apps Inc</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Manufacturer City</th> </tr> <tr> <th>manucity</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>Spring Valley</td> <td>2</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Manufacturer State</th> </tr> <tr> <th>manustat</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>XA</td> <td>1</td> </tr> <tr> <td>ca</td> <td>1</td> </tr> </tbody> </table>	Manufacturer Number		manunum	Frequency	800	1	900	1	Manufacturer Name		manuname	Frequency	21st Century Analytics Co	1	Absolute Best Apps Inc	1	Manufacturer City		manucity	Frequency	Spring Valley	2	Manufacturer State		manustat	Frequency	XA	1	ca	1	<p>The FREQ Procedure</p> <table border="1"> <thead> <tr> <th colspan="2">Product Number</th> </tr> <tr> <th>prodnum</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>5005</td> <td>1</td> </tr> <tr> <td>5006</td> <td>1</td> </tr> <tr> <td>5007</td> <td>1</td> </tr> <tr> <td>5008</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Product Name</th> </tr> <tr> <th>proddname</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>AI Software</td> <td>1</td> </tr> <tr> <td>Analytics Software</td> <td>1</td> </tr> <tr> <td>Fuzzy Matching Software</td> <td>1</td> </tr> <tr> <td>Storytelling Software</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Manufacturer Number</th> </tr> <tr> <th>manunum</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>500</td> <td>4</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Product Type</th> </tr> <tr> <th>prodtype</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>Software</td> <td>1</td> </tr> <tr> <td>Softwares</td> <td>2</td> </tr> <tr> <td>Softwear</td> <td>1</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">Product Cost</th> </tr> <tr> <th>prodcost</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>\$399.00</td> <td>3</td> </tr> <tr> <td>\$499.00</td> <td>1</td> </tr> </tbody> </table>	Product Number		prodnum	Frequency	5005	1	5006	1	5007	1	5008	1	Product Name		proddname	Frequency	AI Software	1	Analytics Software	1	Fuzzy Matching Software	1	Storytelling Software	1	Manufacturer Number		manunum	Frequency	500	4	Product Type		prodtype	Frequency	Software	1	Softwares	2	Softwear	1	Product Cost		prodcost	Frequency	\$399.00	3	\$499.00	1
Customer Number																																																																																																														
custnum	Frequency																																																																																																													
1901	1																																																																																																													
2001	1																																																																																																													
2101	1																																																																																																													
Customer Name																																																																																																														
custname	Frequency																																																																																																													
El Cajon Analytics Center	1																																																																																																													
Pacific Beach Metropolis	1																																																																																																													
Solana Beach High Tech	1																																																																																																													
Customer's Home City																																																																																																														
custcity	Frequency																																																																																																													
La Kahone	1																																																																																																													
Pacific Baech	1																																																																																																													
Solana Baech	1																																																																																																													
Manufacturer Number																																																																																																														
manunum	Frequency																																																																																																													
800	1																																																																																																													
900	1																																																																																																													
Manufacturer Name																																																																																																														
manuname	Frequency																																																																																																													
21st Century Analytics Co	1																																																																																																													
Absolute Best Apps Inc	1																																																																																																													
Manufacturer City																																																																																																														
manucity	Frequency																																																																																																													
Spring Valley	2																																																																																																													
Manufacturer State																																																																																																														
manustat	Frequency																																																																																																													
XA	1																																																																																																													
ca	1																																																																																																													
Product Number																																																																																																														
prodnum	Frequency																																																																																																													
5005	1																																																																																																													
5006	1																																																																																																													
5007	1																																																																																																													
5008	1																																																																																																													
Product Name																																																																																																														
proddname	Frequency																																																																																																													
AI Software	1																																																																																																													
Analytics Software	1																																																																																																													
Fuzzy Matching Software	1																																																																																																													
Storytelling Software	1																																																																																																													
Manufacturer Number																																																																																																														
manunum	Frequency																																																																																																													
500	4																																																																																																													
Product Type																																																																																																														
prodtype	Frequency																																																																																																													
Software	1																																																																																																													
Softwares	2																																																																																																													
Softwear	1																																																																																																													
Product Cost																																																																																																														
prodcost	Frequency																																																																																																													
\$399.00	3																																																																																																													
\$499.00	1																																																																																																													

Determining the number of distinct values a categorical variable has is critical to the fuzzy matching process. Acquiring this information helps everyone become more involved and have a better understanding of the number of distinct variable levels, the unique values, and the number of occurrences for developing data-driven programming constructs and elements (see Chapter 10, “Tuning for Performance and Efficiency”). The following SAS code provides us with the number of BY-group levels for each variable of interest we see in Figure 9.7.

PROC FREQ Code

```
TITLE "By-group NLevels in Customers_with_Messy_Data";
PROC FREQ DATA=Customers_with_Messy_Data NLEVELS;
RUN;
TITLE "By-group NLevels in Manufacturers_with_Messy_Data";
PROC FREQ DATA=Manufacturers_with_Messy_Data NLEVELS;
RUN;
```

```
TITLE "By-group NLevels in Products_with_Messy_Data";
PROC FREQ DATA=Products_with_Messy_Data NLEVELS;
RUN;
```

Results

By-group NLevels in Customers_with_Messy_Data

The FREQ Procedure

Number of Variable Levels		
Variable	Label	Levels
custnum	Customer Number	3
custname	Customer Name	3
custcity	Customer's Home City	3

Customer Number				
custnum	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1901	1	33.33	1	33.33
2001	1	33.33	2	66.67
2101	1	33.33	3	100.00

Customer Name				
custname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
El Cajon Analytics Center	1	33.33	1	33.33
Pacific Beach Metropolis	1	33.33	2	66.67
Solana Beach High Tech	1	33.33	3	100.00

Customer's Home City				
custcity	Frequency	Percent	Cumulative Frequency	Cumulative Percent
La Kahone	1	33.33	1	33.33
Pacific Baech	1	33.33	2	66.67
Solana Baech	1	33.33	3	100.00

By-group NLevels in Manufacturers_with_Messy_Data

The FREQ Procedure

Number of Variable Levels		
Variable	Label	Levels
manunum	Manufacturer Number	2
manuname	Manufacturer Name	2
manucity	Manufacturer City	1
manustat	Manufacturer State	2

Manufacturer Number				
manunum	Frequency	Percent	Cumulative Frequency	Cumulative Percent
800	1	50.00	1	50.00
900	1	50.00	2	100.00

Manufacturer Name				
manuname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
21st Century Analytics Co	1	50.00	1	50.00
Absolute Best Apps Inc	1	50.00	2	100.00

Manufacturer City				
manucity	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Spring Valley	2	100.00	2	100.00

Manufacturer State				
manustat	Frequency	Percent	Cumulative Frequency	Cumulative Percent
XA	1	50.00	1	50.00
ca	1	50.00	2	100.00

By-group NLevels in Products_with_Messy_Data**The FREQ Procedure**

Number of Variable Levels		
Variable	Label	Levels
prodnum	Product Number	4
prodname	Product Name	4
manunum	Manufacturer Number	1
prodtype	Product Type	3
prodcost	Product Cost	2

Product Number				
prodnum	Frequency	Percent	Cumulative Frequency	Cumulative Percent
5005	1	25.00	1	25.00
5006	1	25.00	2	50.00
5007	1	25.00	3	75.00
5008	1	25.00	4	100.00

Product Name				
prodname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
AI Software	1	25.00	1	25.00
Analytics Software	1	25.00	2	50.00
Fuzzy Matching Software	1	25.00	3	75.00
Storytelling Software	1	25.00	4	100.00

Manufacturer Number				
manunum	Frequency	Percent	Cumulative Frequency	Cumulative Percent
500	4	100.00	4	100.00

Product Type				
prodtype	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Software	1	25.00	1	25.00
Softwares	2	50.00	3	75.00
Softwear	1	25.00	4	100.00

Product Cost				
prodcost	Frequency	Percent	Cumulative Frequency	Cumulative Percent
\$399.00	3	75.00	3	75.00
\$499.00	1	25.00	4	100.00

Data Cleaning

Data cleaning, also called data cleansing or data scrubbing, is the process of identifying and fixing data quality issues including missing values, invalid character and numeric values, outlier values, value ranges, duplicate observations, and other anomalies found in data sets. SAS provides many powerful ways to perform data cleaning tasks. If you want to learn more, a great resource is Ron Cody's book, *Cody's Data Cleaning Techniques Using SAS, Third Edition*.

Use SAS Functions to Modify Data

SAS functions are an essential component of the Base SAS software. Representing a variety of built-in and callable routines, functions serve as the “work horses” in the SAS software providing users with “ready-to-use” tools designed to ease the burden of writing and testing often lengthy and complex code for a variety of programming tasks. The advantage of using SAS functions is evidenced by their relative ease of use, and their ability to provide a more efficient, robust, and scalable approach to simplifying a process or programming task.

It is sometimes necessary to concatenate fields when matching files, because the fields could be concatenated in one file while separate in another. SAS functions span many functional categories, and this paper focuses on those that are integral to the fuzzy matching process. The following is a list of alternative methods of concatenating strings and/or variables together.

- Use the STRIP function to eliminate leading and trailing blanks, and then concatenate the stripped fields using the concatenation operator, and insert blanks between the stripped fields.
- Use one of the following CAT functions to concatenate fields:
 - CAT, the simplest of concatenation functions, joins two or more strings and/or variables together, end-to-end producing the same results as with the concatenation operator.
 - CATQ is similar to the CATX function, but the CATQ function adds quotation marks to any concatenated string or variable.
 - CATS removes leading and trailing blanks and concatenates two or more strings and/or variables together.
 - CATT removes trailing blanks and concatenates two or more strings and/or variables together.
 - CATX, perhaps the most robust CAT function, removes leading and trailing blanks and concatenates two or more strings and/or variables together with a delimiter between each.

Explore Data Issues with PROC FORMAT and PROC SQL

Problems with inaccurately entered data often necessitate time-consuming validation activities. A popular technique used by many to identify data issues is to use the FORMAT procedure. In the next example, a user-defined format called, \$Prodtype_Validation, is created using PROC FORMAT, and a PROC SQL query is used to identify and display data issues associated with the values found in the Prodtype variable.

PROC FORMAT and PROC SQL Query

```

PROC FORMAT LIBRARY=WORK;
  VALUE $Prodtype_Validation
    'Laptop'      = 'Laptop'
    'Phone'       = 'Phone'
    'Software'    = 'Software'
    'Workstation' = 'Workstation'
    Other         = 'ERROR - Invalid Prodtype';
RUN;

PROC SQL;
  TITLE "Validation Report for Products_with_messy_data Prodtype
Variable";
  SELECT Prodnum,
         Prodname,
         Manunum,
         Prodtype,
         Prodtype FORMAT=$Prodtype_Validation.,
         Prodcost
    FROM Products_with_messy_data;
QUIT;

```

Results

Validation Report for Products_with_messy_data Prodtype Variable

Product Number	Product Name	Manufacturer Number	Product Type	Product Type	Product Cost
5005	Analytics Software	500	Softwear	ERROR - Invalid Prodtype	\$499.00
5006	Storytelling Software	500	Softwares	ERROR - Invalid Prodtype	\$399.00
5007	Fuzzy Matching Software	500	Software	ERROR - Invalid Prodtype	\$399.00
5008	AI Software	500	Softwares	ERROR - Invalid Prodtype	\$399.00

Once the invalid Prodtype categories are identified with the validation report, users have the option of using one or more data cleaning techniques to manually correct the data, automate the process, or apply fuzzy matching techniques to correct (or handle) each invalid product type category.

Here are a few additional things to consider using during the data cleaning process.

Add Categories, if Available, to the Start of the Name

Adding categories can help eliminate matches that might occur if two businesses in the same general geographic area have the same name. For example: “Johnson’s Best” could describe a hardware store, a restaurant, or another type of business. By adding a variable that contains the industry type (e.g., Agriculture, Bank, Restaurant, Hospital, Hotel, etc.) could prevent mismatches from occurring.

Remove Special or Extraneous Characters

Punctuation can differ even when names or titles are the same. Therefore, consider removing the following characters: ‘ “ & ? – from product names, company names, addresses, and other character values. For example, “Del Mar’s Tech Center” and “Del Mar Tech Center” most likely refers to the same customer name in the Customers data set even though the former contains an apostrophe and the latter does not.

Put All Characters in Uppercase Notation and Remove Leading Blanks

Different databases could have different standards for capitalization, and some character strings can be copied in with leading blanks. As found in our example data sets the value contained in the Title variable can be stored as all lowercase, uppercase, or in mixed-case which can impact the success of traditional merge and join matching techniques.

Consequently, to remedy the issues associated with case and leading blanks, consider using the STRIP function to remove leading and trailing blanks along with the UPCASE function to convert all character values to uppercase.

Remove Words that Might or Might not Appear in Key Fields

Commonly used words in language, referred to as stop words, are frequently ignored by many search and retrieval processes. Stop words are classified as irrelevant and, as a result, are inserted into stop lists and are ignored. Examples include The, .com, Inc, LTD, LLC, DIVISION, CORP, CORPORATION, CO., and COMPANY. Some database tables might include these, while others might not.

Choose a Standard for Addresses and Phone Numbers

Address values can present a challenge when analyzing and processing data sources. To help alleviate comparison issues, decide whether to use Avenue or Ave, Road or Rd, Street or St, etc, and then convert the address values accordingly or create a user-defined lookup process using PROC FORMAT to match the standard values.

Phone numbers could be standardized to remove “,“ and “–“ to convert them to digits and validated using the ALLDIGITS() function (and similar functions) as well as for the valid number of digits.

Normalize ZIP Codes when Matching Addresses and Use Geocodes when Available

Another useful technique is to remove the last 4 digits of 9-digit ZIP codes, because some files might have only 5-digit ZIP codes. Since some files might have ZIP codes as numeric fields, and other files might have ZIP codes as character fields, be sure to include leading zeros. For example, ZIP codes with a leading zero, as in 04101, would appear in a numeric field as 4101 requiring the leading zero to be inserted along with the specification of a Z5. informat and format being assigned to the ZIP code variable.

If working with US ZIP codes, make sure they are all numeric. This may not apply for other countries. One common mistake to watch for is when state codes or abbreviations, such as Canada, with abbreviation CA, are put in as the state CA (California) instead of the country CA (Canada). Since Canada has an alphanumeric 6-character ZIP code, this, hopefully, will be caught when checking for numeric ZIP codes.

If the user has access to geocodes, or if they are in the input data, geocodes can provide a further level of validation in addition to the ZIP codes.

Specify the DUPOUT=, NODUPRECS, or NODUPKEYS Options

A popular and frequently used procedure, PROC SORT, identifies and removes duplicate observations from a data set. By specifying one or more of the SORT procedure's three

options: **DUPOUT=**, **NODUPRECS**, and **NODUPKEYS**, users are able to control how duplicate observations are identified and removed.

PROC SORT's DUPOUT= option is often used to identify duplicate observations before removing them from a data set. A DUPOUT= option, often specified when a data set is too large for visual inspection, can be used with the NODUPKEYS or NODUPRECS options to review a data set that contains duplicate keys or duplicate observations. In the next example, the DUPOUT=, OUT= and NODUPKEY options are specified to identify duplicate keys. The NODUPKEY option removes observations that have the same key values, so that only one remains in the output data set. The PROC SORT is followed by PROC SQL queries so the results can be displayed and examined.

PROC SORT and PROC SQL Code

```
PROC SORT DATA=Products_with_Messy_Data /* Input data set */
           DUPOUT=Products_Dupout_NoDupkey /* Data set containing dups
*/ 
           OUT=Products_Sorted_Cleaned_NoDupkey /* Data set with dups
removed */
           NODUPKEY;
BY Prodname; /* Key */
RUN;

PROC SQL;
  TITLE "Observations Slated for Removal";
  SELECT *
    FROM Products_Dupout_NoDupkey;
  TITLE "Cleaned Movies Data Set";
  SELECT *
    FROM Movies_Sorted_Cleaned_NoDupkey;
QUIT;
```

The NODUPKEY option retains only one observation from any group of observations with duplicate keys. When observations with identical key values are not adjacent to each other, users may first need to specify the NODUPKEY or NODUPKEYS option and sort the data set by all the variables (BY _ALL_;) to ensure the observations are in the correct order to remove all duplicates (SAS Usage Note 1566, 2000).

Although the removal of duplicates using PROC SORT is a popular technique among many SAS users, an element of care should be given to using this method when processing large data sets. Since sort operations can often be CPU-intensive, it is recommended to compare PROC SORT to procedures like PROC SQL with the SELECT DISTINCT keyword and/or SAS PROC SUMMARY with the CLASS statement to determine the performance impact of one method versus another.

Data Transformations

Data transformations can be necessary to compare files. Data set structures are sometimes not in the desired format and might need to be converted from wide to long or long to wide, and might need to be reconciled by having their variables grouped in different ways. When a data set's structure and data are transformed, it is typically recommended that a new data set be created from the original one.

PROC TRANSPOSE is handy for restructuring data in a data set, and is typically used in preparation for special types of processing like array processing. In its simplest form, data can be transformed with or without grouping. In the next example, the Products data set is first sorted in ascending order by the variable Prodtype then the sorted data set is transposed using the Prodtype variable as the BY-group variable. The results show the product names transformed within each product type.

PROC TRANSPOSE Code

```
PROC SORT DATA=Products_with_Messy_Data
           OUT=work.Products_Sorted;
   BY Prodtype; /* BY-Group to Transpose */
RUN;

PROC TRANSPOSE DATA=work.Products_Sorted
                 OUT=work.Products_Transposed;
   VAR Prodname; /* Variable to Transpose */
   BY Prodtype; /* BY-Group to Transpose */
RUN;

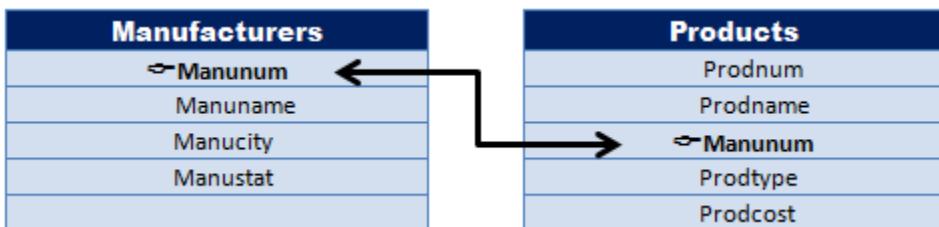
PROC SQL;
   SELECT *
      FROM work.Products_Transposed;
QUIT;
```

Results

Product Type	NAME OF FORMER VARIABLE	LABEL OF FORMER VARIABLE	COL1	COL2
Software	prodname	Product Name	Fuzzy Matching Software	
Softwares	prodname	Product Name	Storytelling Software	AI Software
Softwear	prodname	Product Name	Analytics Software	

Exact Matching Process

In an age of endless spreadsheets, text files, apps, and relational database management systems (RDBMS), it's unusual to find a single spreadsheet, CSV file, table, or data set that contains all the data needed to answer an organization's questions. Today's data exists in many forms and all too often involves matching two or more data sources to create a combined file. The matching process typically involves combining two or more data sets, spreadsheets, and/or files possessing a shared, common and reliable, identifier (or key) to create a single data set, spreadsheet, and/or file. The matching process, illustrated in Figure 9.5, shows two tables, Manufacturers and Products, with a key, Manunum, to combine (or join) the two tables together.

Figure 9.5: Matching Process

Since we are trying to match entries that have an exact match, we can save processing time by immediately eliminating the observations (or rows) with missing key information. This can be accomplished in a number of ways, including constructing PROC SQL join queries to bypass processing observations with missing manufacturer and/or product information. Once missing observations with missing keys are eliminated, the focus can then be turned to processing observations that have exact matches on name, address, company name, and as with our example data sets, the Manunum variable.

PROC SQL Code

```
PROC SQL;
  CREATE TABLE Manufacturers_Products_Matches AS
    SELECT M.Manunum,
           M.Manuname,
           M.Manucity,
           M.Manustat,
           P.Prodnum,
           P.Prodname,
           P.Prodtype,
           P.Prodcost
      FROM Manufacturers M,
           Products P
     WHERE M.Manunum = P.Manunum
       AND P.Prodname NE "";
  TITLE Matched Observations;
  SELECT * FROM Manufacturers_Products_Matches;
QUIT;
```

Results

Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State	Product Number	Product Name	Product Type	Product Cost
111	Cupid Computer	Houston	TX	1110	Dream Machine	Workstation	\$3,200.00
120	Storage Devices Inc	San Mateo	CA	1200	Business Machine	Workstation	\$3,300.00
210	Global Comm Corp	San Diego	CA	2101	Analog Cell Phone	Phone	\$35.00
210	Global Comm Corp	San Diego	CA	2102	Digital Cell Phone	Phone	\$175.00
500	KPL Enterprises	San Diego	CA	5001	Spreadsheet Software	Software	\$299.00
500	KPL Enterprises	San Diego	CA	5002	Database Software	Software	\$399.00
500	KPL Enterprises	San Diego	CA	5003	Wordprocessor Software	Software	\$299.00
500	KPL Enterprises	San Diego	CA	5004	Graphics Software	Software	\$299.00

Fuzzy Matching Processing

As we've seen, when a shared and reliable key exists between input data sources, the matching process is fairly uncomplicated. But when a shared key associated with the various

input data sources is nonexistent, inexact, or unreliable, the matching process often becomes more involved and problematic. Stephen Sloan and Dan Hoicowitz suggest that special processes are needed to successfully match the names and addresses from different files when they are similar, but not exactly the same. In a constructive and systematic way, Lafler and Sloan, describe their six-step approach to cleaning data and performing fuzzy matching processes.

Once the data has been cleaned and transformed, a variety of fuzzy matching techniques are available for use. These techniques are designed to be used in a systematic way when a reliable key between data sources is nonexistent, inexact, or unreliable.

Fuzzy matching techniques are available with most, if not all, the leading software languages including R, Python, Java, and others. SAS offers four techniques to help make fuzzy matching easier and more effective: the SOUNDEX algorithm and the SPEDIS, COMPLEV, and COMPGED functions.

Soundex Algorithm

The Soundex (phonetic matching) algorithm involves matching files on words that sound alike. As one of the earliest fuzzy matching techniques, Soundex was invented and patented by Margaret K. Odell and Robert C. Russell in 1918 and 1922 to help match surnames that sound alike. It is limited to finding phonetic matches and adheres to the following rules when performing a search:

- Ignores case (case insensitive)
- Ignores embedded blanks and punctuation marks
- Is better at finding English-sounding names

Although the Soundex algorithm does a fairly good job with English-sounding names, it frequently falls short when dealing with the multitude of data sources found in today's world economy where English and non-English sounding names are commonplace. It also has been known to miss similar-sounding surnames like Rogers and Rodgers while matching dissimilar surnames such as Smith, Snthe, and Schmitt.

So, how does the Soundex algorithm work? As implemented, SAS determines whether a name (or a variable's contents) sounds like another by converting each word to a code. The value assigned to the code consists of the first letter in the word followed by one or more digits. Vowels, A, E, I, O, and U, along with H, W, Y, and non-alphabetical characters do not receive a coded value and are ignored. Double letters (e.g., 'TT') are assigned a single code value for both letters. The codes derived from each word conform to the letters and values are found in Table 9.1.

Table 9.1: Soundex Algorithm Rules

Letter	Value
B, P, F, V	1
C, S, G, J, K, Q, X, Z	2
D, T	3

Letter	Value
L	4
M, N	5
R	6

The general syntax of the Soundex algorithm takes the form of:

Variable =* “character-string”

The following example illustrates how the customer city, “Pacific Beach”, is assigned a Soundex value. P has a value of 1 but is retained as P, A is ignored, C is assigned a value of 2, I is ignored, F is assigned a value of 1, I is ignored, C is assigned a value of 2, B is assigned a value of 1, E is ignored, A is ignored, C is assigned a value of 2, and H is ignored. The converted code of P21212 for “Pacific Beach” is then matched with any other customer city name that has the same assigned code.

In the next example, the Soundex algorithm is illustrated using the =* operator in a PROC SQL step with a WHERE clause to find a similar sounding customer city of “Solana Beach”, in the Customers_with_messy_data data set.

PROC SQL Code with SOUNDEX Algorithm

```
PROC SQL;
  SELECT *
    FROM Customers_with_messy_data
   WHERE Custcity =* "Solana Beach";
QUIT;
```

The result from the SOUNDEX algorithm selected “Solana Baech” which has the same derived Soundex value as “Solana Beach”.

Results

Customer Number	Customer Name	Customer's Home City
2001	Solana Beach High Tech	Solana Baech

SPEDIS Function

The SPEDIS, or Spelling Distance, function measures how close one word is to another word when it comes to spelling. By translating a keyword into a query, the SPEDIS function returns a value that can be used to determine the spelling distance between two words. Because the SPEDIS function evaluates numerous scenarios, users may experience varying performance issues in comparison to other fuzzy matching techniques such as the COMPLEV and COMPGED functions, particularly when using it with long strings.

The SPEDIS function evaluates query and keyword arguments returning nonnegative spelling distance values. A derived value of zero indicates an exact match. Generally, derived values are less than 100, but never greater than 200. It is recommended that when using the SPEDIS

function for matching tasks to specify spelling distance values greater than zero and in increments of 10 (e.g., 10, 20, etc.).

So, how does the SPEDIS function work? As implemented, the SPEDIS function determines whether two names (or variables' contents) are alike by computing an asymmetric spelling distance between two words. Before the SPEDIS function returns the distance between a query and a keyword, it removes trailing blanks. A derived value of 0 is returned if the keyword exactly matches the query. SPEDIS assigns costs (or penalty points) for each operation that is required to convert the keyword to the query. For example, when the first letter incorrectly matches then more points are assigned than when other letters do not match. Once the total costs have been computed, the resulting value represents a percentage of the length of the first argument. Table 9.2 illustrates the costs corresponding to the operations performed by the SPEDIS function.

Table 9.2: SPEDIS Cost Rules

Operation	Cost	Description
Match	0	No change
Singlet	25	Delete one of the double letters
Doublet	50	Double a letter
Swap	50	Reverse the order of two consecutive letters
Truncate	50	Delete a letter from the end
Append	35	Add a letter to the end
Delete	50	Delete a letter from the middle
Insert	100	Insert a letter in the middle
Replace	100	Replace a letter in the middle
Firstdel	100	Delete the first letter
Firstins	200	Insert a letter at the beginning
Firstrep	200	Replace the first letter

The general syntax of the SPEDIS function takes the form of:

SPEDIS (query, keyword)

In this example, a simple PROC SQL query with a WHERE clause and CALCULATED keyword is specified to capture and show the observations derived by the SPEDIS function for finding exact matches for the customer city, “Solana Beach”.

PROC SQL Code with SPEDIS Function

```
PROC SQL;
  TITLE "SPEDIS Function Matches";
  SELECT *,
```

```

SPEDIS(Custcity,"Solana Beach") AS Spedis_Value
FROM Customers_with_messy_data
WHERE CALCULATED Spedis_Value LE 10;
QUIT;

```

The result from the SOUNDEX algorithm example is displayed below. The observation associated with “Solana Baech” was selected as a match for “Solana Beach” and derived a SPEDIS value of 4. This result is the same as what was selected by the SOUNDEX algorithm.

Results

SPEDIS Function Matches

Customer Number	Customer Name	Customer's Home City	Spedis_Value
2001	Solana Beach High Tech	Solana Baech	4

In the next example, a PROC SQL query with a WHERE clause and CALCULATED keyword is specified to capture and display the observations derived by the SPEDIS function for the manufacturer state, “CA”, in the Manufacturers_with_messy_data data set.

PROC SQL Code with SPEDIS Function

```

PROC SQL;
  TITLE "SPEDIS Function Matches";
  SELECT *,
    SPEDIS(Manustat,"CA") AS Spedis_Value
  FROM Manufacturers_with_messy_data
  WHERE CALCULATED Spedis_Value GE 0;
QUIT;

```

Results

The result from the SOUNDEX algorithm example is displayed below. The observations associated with “ca” and “XA” were selected for “CA” with the derived SPEDIS values of 150 and 100, respectively.

SPEDIS Function Matches

Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State	Spedis_Value
800	21st Century Analytics Co	Spring Valley	ca	150
900	Absolute Best Apps Inc	Spring Valley	XA	100

COMPLEV Function

The COMPLEV, or Levenshtein Edit Distance, function is another fuzzy matching SAS technique. COMPLEV counts the minimum number of single-character insert, delete, or replace operations needed to determine how close two strings are. Unlike the SPEDIS function and COMPGED function (discussed in the next section), the COMPLEV function

assigns a score for each operation and returns a value indicating the number of operations. The general syntax of the COMPLEV function takes the form of:

COMPLEV (string-1, string-2 <, cutoff-value> <, modifier>)

Required Arguments:

string-1 specifies a character variable, constant or expression.

string-2 specifies a character variable, constant or expression.

Optional Arguments:

cutoff-value specifies a numeric variable, constant or expression. If the actual Levenshtein edit distance is greater than the value of **cutoff**, the value that is returned is equal to the value of **cutoff**.

modifier specifies a value that alters the action of the COMPLEV function. Valid modifier values are:

- i or I Ignores the case in string-1 and string-2.
- l or L Removes leading blanks before comparing the values in string-1 or string-2.
- n or N Ignores quotation marks around string-1 or string-2.
- :(colon) Truncates the longer of string-1 or string-2 to the length of the shorter string.

In the example below, the COMPLEV function determines the best possible match for the Custcity value, “Solana Beach”, in the Customers_with_messy_data data set. The COMPLEV_Number column displays the number of operations that have been performed. The lower the value the better the match (e.g., 0 = Best match, 1 = Next Best match, etc.). The search argument of, “Solana Baech”, produces a derived COMPLEV_Number value of 2.

PROC SQL Code with COMPLEV Function

```
PROC SQL;
  SELECT *,
    COMPLEV(Custcity,"Solana Beach") AS COMPLEV_Number
  FROM Customers_with_messy_data
  ORDER BY Custnum;
QUIT;
```

The output below shows the derived values for the Levenshtein Edit Distance for different spelling variations for, “Solana Beach”, in the column, Custcity.

Results

Customer Number	Customer Name	Customer's Home City	COMPLEV_Number
1901	Pacific Beach Metropolis	Pacific Baech	9
2001	Solana Beach High Tech	Solana Baech	2
2101	EI Cajon Analytics Center	La Kahone	11

In the next example, the COMPLEV function's computed value is limited to 1 or less using a WHERE clause. The results show the observation associated with the product type, "Software".

PROC SQL Code with COMPLEV Function

```
PROC SQL;
  SELECT *,
    COMPLEV(Prodtype,"Software") AS COMPLEV_Number
  FROM Products_with_messy_data
  WHERE CALCULATED COMPLEV_Number LE 1
    ORDER BY Prodname;
QUIT;
```

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost	COMPLEV_Number
5008	AI Software	500	Softwares	\$399.00	1
5007	Fuzzy Matching Software	500	Software	\$399.00	1
5006	Storytelling Software	500	Softwares	\$399.00	1

In the next example, the COMPLEV function has a modifier value of "INL" to ignore the case, remove leading blanks, and ignore quotation marks around string-1 and string-2 and a value for the COMPLEV_Score of 0 (perfect match). The results show the observation associated with the Manustat, "CA" in the argument for string-1 matches the value of "ca" in the argument for string-2.

PROC SQL Code with COMPLEV Function and Arguments

```
PROC SQL;
  TITLE "COMPLEV Function Matches";
  SELECT *,
    COMPLEV(Manustat,"CA","INL") AS COMPLEV_Score
  FROM Manufacturers_with_messy_data
  WHERE CALCULATED COMPLEV_Score = 0;
QUIT;
```

Results

COMPLEV Function Matches

Manufacturer Number	Manufacturer Name	Manufacturer City	Manufacturer State	COMPLEV_Score
800	21st Century Analytics Co	Spring Valley	ca	0

COMPGED Function

The COMPGED function is another fuzzy matching technique that is facilitated as a SAS function. It works by computing a Generalized Edit Distance (GED) score when comparing two text strings. The Generalized Edit Distance score is a generalization of the Levenshtein edit distance, which is a measure of dissimilarity between two strings where the edit distance derived as the number of operations (deletions, insertions, or replacements) of a single character to transform string-1 into string-2. In Sloan and Hoicowitz's paper, "*Fuzzy Matching: Where Is It Appropriate and How Is It Done? SAS Can Help,*" they describe using

the COMPGED function to match data sets with unreliable identifiers (or keys), the higher the GED score the less likely the two strings match. Conversely, for the greatest likelihood of a match with the COMPGED function users should seek the lowest derived score from evaluating all the possible ways of comparing the two strings, string-1 with string-2.

The COMPGED function returns values that are multiples of 10, e.g., 20, 100, 200, etc. Based on experience, a COMPGED score of 100 or less often translates to valid matches for the comparison that they are performing. The COMPGED function compares two character strings, along with optional parameters to control case-sensitive scenarios, leading blanks, the use of quotation marks, and the handling of strings with unequal lengths. The general syntax of the COMPGED function takes the form of:

COMPGED (string-1, string-2 <, cutoff-value> <, modifier>)

Required Arguments:

string-1 specifies a character variable, constant or expression.

string-2 specifies a character variable, constant or expression.

Optional Arguments:

cutoff-value specifies a numeric variable, constant or expression. If the actual generalized edit distance is greater than the value of **cutoff**, the value that is returned is equal to the value of **cutoff**.

modifier specifies a value that alters the action of the COMPGED function. Valid modifier values are:

- i or I Ignores the case in string-1 and string-2.
- l or L Removes leading blanks before comparing the values in string-1 or string-2.
- n or N Ignores quotation marks around string-1 or string-2.
- : (colon) Truncates the longer of string-1 or string-2 to the length of the shorter string.

Table 9.3 shows the different point values that COMPGED assigns for changes from one character string to another.

Table 9.3: COMPGED Scoring Algorithm

Operation	Default Cost in Units	Description of Operation
APPEND	50	When the output string is longer than the input string, add any one character to the end of the output string without moving the pointer.
BLANK	10	Do any of the following: Add one space character to the end of the output string without moving the pointer. When the character at the pointer is a space character, advance the pointer by one position without changing the output string.

Operation	Default Cost in Units	Description of Operation
		When the character at the pointer is a space character, add one space character to the end of the output string, and advance the pointer by one position.
		If the cost for BLANK is set to zero by the COMPCOST function, the COMPGED function removes all space characters from both strings before doing the comparison.
DELETE	100	Advance the pointer by one position without changing the output string.
DOUBLE	20	Add the character at the pointer to the end of the output string without moving the pointer.
FDELETE	200	When the output string is empty, advance the pointer by one position without changing the output string.
FINSE	200	When the pointer is in position one, add any one character to the end of the output string without moving the pointer.
FREPLACE	200	When the pointer is in position one and the output string is empty, add any one character to the end of the output string, and advance the pointer by one position.
INSERT	100	Add any one character to the end of the output string without moving the pointer.
MATCH	0	Copy the character at the pointer from the input string to the end of the output string, and advance the pointer by one position.
PUNCTUATION	30	<p>Do any of the following:</p> <p>Add one punctuation character to the end of the output string without moving the pointer.</p> <p>When the character at the pointer is a punctuation character, advance the pointer by one position without changing the output string.</p> <p>When the character at the pointer is a punctuation character, add one punctuation character to the end of the output string, and advance the pointer by one position.</p>
REPLACE	100	Add any one character to the end of the output string, and advance the pointer by one position.
SINGLE	20	When the character at the pointer is the same as the character that follows in the input string, advance the pointer by one position without changing the output string.
SWAP	20	<p>Copy the character that follows the pointer from the input string to the output string. Then copy the character at the pointer from the input string to the output string.</p> <p>Advance the pointer two positions.</p>

Operation	Default Cost in Units	Description of Operation
TRUNCATE	10	When the output string is shorter than the input string, advance the pointer by one position without changing the output string.

Although the COMPGED function provides users with a robust and comprehensive approach to performing searches and/or matches, some users still resort to using traditional data transformation methods. In the example below, traditional WHERE clause logic with the UPCASE function is specified to perform a concatenation of two queries by indicating the matching criteria. As the results show, the WHERE clause permits both data sets to be concatenated, but the results fail to collapse (or consolidate) the values for the categorical variable, Prodtype, to the value, “Software”.

PROC SQL Code with Traditional WHERE clause Logic

```
PROC SQL;
  SELECT *
    FROM Products
   WHERE UPCASE(Prodtype) IN
("SOFTWARE", "SOFTWEAR", "SWOTWARA", "SOFTWARES")
  OUTER UNION CORRESPONDING
  SELECT *
    FROM Products_with_messy_data
   WHERE UPCASE(Prodtype) IN
("SOFTWARE", "SOFTWEAR", "SWOTWARA", "SOFTWARES");
QUIT;
```

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
5001	Spreadsheet Software	500	Software	\$299.00
5002	Database Software	500	Software	\$399.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00
5005	Analytics Software	500	Softwear	\$499.00
5006	Storytelling Software	500	Softwares	\$399.00
5007	Fuzzy Matching Software	500	Softwara	\$399.00
5008	AI Software	500	Softwares	\$399.00

To address the shortcomings of the preceding example, the next example specifies a UNION set operator, a CASE expression, and a WHERE clause with an UPCASE function to concatenate and consolidate “Software” products from the Products and Products_with_messy_data data sets. As the code and results suggest, a UNION set operator and a WHERE clause concatenates and consolidates both data sets on “Software” products.

PROC SQL Code with CASE Expressions and WHERE clause Logic

```

PROC SQL;
  CREATE TABLE Products_Concatenated AS
    SELECT Prodnum, Prodname, Manunum, Prodcost,
      CASE
        WHEN UPCASE(Prodtype) IN
          ("SOFTWARE", "SOFTWEAR", "SW", "SOFTWARES")
          THEN "Software"
        ELSE "ERROR - Prodtype Unknown"
      END AS Prodtype
    FROM Products
    WHERE UPCASE(Prodtype) IN
      ("SOFTWARE", "SOFTWEAR", "SW", "SOFTWARES")
    UNION
    SELECT Prodnum, Prodname, Manunum, Prodcost,
      CASE
        WHEN UPCASE(Prodtype) IN
          ("SOFTWARE", "SOFTWEAR", "SW", "SOFTWARES")
          THEN "Software"
        ELSE "ERROR - Prodtype Unknown"
      END AS Prodtype
    FROM Products_with_messy_data
    WHERE UPCASE(Prodtype) IN
      ("SOFTWARE", "SOFTWEAR", "SW", "SOFTWARES");
    SELECT * FROM Products_Concatenated;
QUIT;

```

Results

Product Number	Product Name	Manufacturer Number	Product Cost	Prodtype
5001	Spreadsheet Software	500	\$299.00	Software
5002	Database Software	500	\$399.00	Software
5003	Wordprocessor Software	500	\$299.00	Software
5004	Graphics Software	500	\$299.00	Software
5005	Analytics Software	500	\$499.00	Software
5006	Storytelling Software	500	\$399.00	Software
5007	Fuzzy Matching Software	500	\$399.00	Software
5008	AI Software	500	\$399.00	Software

In the next example, the COMPGED function is specified to determine the best possible match for the Custcity value, “Solana Beach”, found in the Customers_with_messy_data data set. As the code and results suggest, the lower the value of the COMPGED_Score the better the match (e.g., 0 = Best match, 10 = Next Best match, etc.). The search argument, “Solana Beach”, produces a derived COMPGED_Score of 20.

PROC SQL Code with COMPGED Function

```

PROC SQL;
  SELECT *,
    COMPGED(Custcity, "Solana Beach") AS COMPGED_Score
  FROM Customers_with_messy_data
  ORDER BY Custnum;
QUIT;

```

The results show the derived COMPGED_Score values for the spelling variation of “Solana Baech” in the column Custcity.

Results

Customer Number	Customer Name	Customer's Home City	COMPGED_Score
1901	Pacific Beach Metropolis	Pacific Baech	820
2001	Solana Beach High Tech	Solana Baech	20
2101	EI Cajon Analytics Center	La Kahone	750

In the next example, the COMPGED function is specified to determine the best possible match for the Prodtype value, “Software”, in the Products_with_messy_data data set. As the code and Figure 20 suggests, the lower the value of the COMPGED_Score the better the match (e.g., 0 = Best match, 10 = Next Best match, etc.). The search argument, “Software”, produces derived scores for COMPGED_Score of 50, 100 and 200.

PROC SQL Code with COMPGED Function

```
PROC SQL;
  SELECT *,
    COMPGED(Prodtype,"Software") AS COMPGED_Score
  FROM Products_with_messy_data
    ORDER BY Prodbname;
QUIT;
```

The results show the derived COMPGED_Score values for the different spelling variations associated with “Software” in the column Prodtype.

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost	COMPGED_Score
5008	AI Software	500	Softwares	\$399.00	50
5005	Analytics Software	500	Softwear	\$499.00	200
5007	Fuzzy Matching Software	500	Software	\$399.00	100
5006	Storytelling Software	500	Softwares	\$399.00	50

The next example specifies a CASE expression, a COMPGED function with the “INL” argument, a UNION set operator, and a WHERE clause to restrict the results of the concatenated and consolidated “Software” Products and Products_with_messy_data data sets to a COMPGED score of no more than 200. As the code and results suggest, the specification of the UNION set operator with the WHERE clause concatenates and consolidates “Software” products from both data sets.

PROC SQL Code with CASE Expression, COMPGED Function, and WHERE clause Logic

```
PROC SQL;
  CREATE TABLE Products_Concatenated AS
    SELECT Prodnum, Prodbname, Manunum, Prodcost,
      CASE
        WHEN COMPGED(Prodtype,"Software","INL") LE 200 THEN
          "Software"
        ELSE "ERROR - Prodtype Unknown"
      END AS Prodtype
    FROM Products
    UNION
    SELECT Prodnum, Prodbname, Manunum, Prodcost,
      CASE
        WHEN COMPGED(Prodtype,"Software","INL") LE 200 THEN
          "Software"
        ELSE "ERROR - Prodtype Unknown"
      END AS Prodtype
    FROM Products_with_messy_data
    WHERE COMPGED(Prodtype,"Software","INL") LE 200;
```

```

        END AS Prodtype
        FROM Products
        WHERE COMPGED(Prodtype,"Software","INL") LE 200
UNION
SELECT Prodnum, Prodname, Manunum, Prodcost,
CASE
    WHEN COMPGED(Prodtype,"Software","INL") LE 200 THEN
"Software"
    ELSE "ERROR - Prodtype Unknown"
END AS Prodtype
FROM Products_with_messy_data
WHERE COMPGED(Prodtype,"Software","INL") LE 200;
SELECT * FROM Products_Concatenated;
QUIT;

```

Results

Product Number	Product Name	Manufacturer Number	Product Cost	Prodtype
5001	Spreadsheet Software	500	\$299.00	Software
5002	Database Software	500	\$399.00	Software
5003	Wordprocessor Software	500	\$299.00	Software
5004	Graphics Software	500	\$299.00	Software
5005	Analytics Software	500	\$499.00	Software
5006	Storytelling Software	500	\$399.00	Software
5007	Fuzzy Matching Software	500	\$399.00	Software
5008	AI Software	500	\$399.00	Software

CALL COMPCOST Routine

Readers are also able to set the costs for each operation performed by the COMPGED function using the CALL COMPCOST routine. In the next example, the default costs for each operation performed by the COMPGED function are displayed. But using a DATA step, we can assign our own costs for any operation. In the DATA step, we execute the CALL COMPCOST routine to assign costs to three operations : Insert, Delete, and Replace. Finally, we display the results using PROC SQL.

CALL COMPCOST and PROC SQL Code

```

PROC SQL;
    SELECT *,
        COMPGED(Custcity,"Solana Beach") AS COMPGED_Score
    FROM Customers_with_messy_data
    ORDER BY Custnum;
QUIT;
data compcost;
    set Customers_with_messy_data;
    if _n_=1 then CALL COMPCOST('insert=',10,'delete=',11,'replace=',
12);
    Compged_Score = COMPGED(Custcity,"Solana Beach");
run;
PROC SQL;
    SELECT *

```

```

FROM compcost
  ORDER BY Custnum;
QUIT;

```

Results

As you can see, the derived costs are different in each output result. The first set of results illustrates the default costs derived for operations performed with the COMPGED function. The second set of results illustrates the revised costs defined with the CALL COMPCOST routine and performed with the COMPGED function.

Customer Number	Customer Name	Customer's Home City	COMPGED_Score
1901	Pacific Beach Metropolis	Pacific Baech	820
2001	Solana Beach High Tech	Solana Baech	20
2101	El Cajon Analytics Center	La Kahone	750

Customer Number	Customer Name	Customer's Home City	Compged_Score
1901	Pacific Beach Metropolis	Pacific Baech	103
2001	Solana Beach High Tech	Solana Baech	21
2101	El Cajon Analytics Center	La Kahone	120

Use the Lower Score

For those fuzzy matching techniques that are not commutative (it matters which data set is placed first and which is placed second), use the lower score that results from the different sequences.

Validation

As can be seen when comparing the SOUNDEX and SPEDIS methods, and when looking at the results of COMPLEV and COMPGED, these methods worked well on a test data set that was designed to illustrate the results. It should be noted that the COMPLEV function is best used when comparing simple strings where data sizes and/or speed of comparison is important, such as when working with large data sets. It should also be noted that the COMPGED function generally requires more processing time to complete because of its more exhaustive and thorough capabilities.

Stephen Sloan and Dan Hoicowitz conducted research on 50,000 business names to manually identify fuzzy matches using the SAS COMPGED function. The intent of their study was to identify false negatives by looking at an alphabetic sort of the business names. From the extracted test files the researchers identified false positives. Finally, the conditions that were specified in the COMPGED function were repeated until the false positives and false negatives were significantly reduced. These conditions then became part of the fuzzy matching process and achieved improved results efficiently and on demand.

Summary

When data originating from multiple sources contains duplicate observations, duplicate and/or unreliable keys, missing values, invalid values, capitalization and punctuation issues, inconsistent matching variables, and imprecise text identifiers, the matching process is often compromised by unreliable and/or unpredictable results. This chapter demonstrates a six-step approach including identifying, cleaning and standardizing data irregularities, conducting data transformations, and utilizing special-purpose programming techniques such as the application of SAS functions, the SOUNDEX algorithm, the SPEDIS function, approximate string matching functions including COMPLEV and COMPGED, and an assortment of constructive programming techniques to standardize and combine data sets together when the matching columns are unreliable or less than perfect.

Chapter 10: Data-driven Programming

Introduction	343
Programming Paradigms.....	343
SAS Metadata Sources.....	344
DICTIONARY Tables	350
Accessing and Displaying the Number of Rows in Tables.....	351
Accessing and Displaying a Cross-reference List of a Variable Defined in All Tables	352
Capturing a List of Variables from the COLUMNS Dictionary Table	352
CALL EXECUTE Routine	354
Custom-defined Formats	357
Macro Language	360
Producing Multiple Excel Files	361
Summary	364

Introduction

Data-driven programming, or data-oriented programming (DOP), is a specific programming paradigm where the data or data structures, and not the program logic, control the flow of a program. Often, data-driven programming approaches are applied in organizations with structured and unstructured data for filtering, aggregating, transforming, and calling other programs. This chapter explores several data-driven programming techniques that are available to SAS and PROC SQL users.

The SAS System collects and populates valuable information (“metadata”) about SAS libraries, data sets (tables), catalogs, indexes, macros, system options, titles, views, and other read-only tables called dictionary tables. Dictionary tables serve a special purpose by providing system-related information about the current SAS session’s SAS libraries, databases, and content. When a query is requested against a dictionary table, SAS automatically launches a discovery process at runtime to collect information pertinent to that table. Metadata content can be very useful for developing data-driven techniques. Other data-driven programming techniques include dynamically constructing a user-defined format directly from data, and using the SQL procedure and the macro language to perform an automated iterative (looping) process.

Programming Paradigms

Programming languages are often classified by their basic features into one of many programming paradigms. Three popular programming paradigms in use today by programming professionals are:

1. **Procedural programming** – represented by blocks of code being organized logically by function, such as data input, data processing or manipulation, results, and output;
2. **Object-oriented programming** – represented by a combination of functionality (behaviors) and data (attributes) hidden inside an object which can then be arranged into classes;
3. **Data-driven programming** – represented by data controlling the flow of execution in a program.

In this chapter, we will be focusing on data-driven programming. Data-driven programming involves a program where the decisions and processes (the flow of execution) are controlled (or dictated) by the data (or data structures). The advantages of a data-driven programming environment are many including the ability to make our code a bit shorter and as a result possibly easier to read, enhanced flexibility due to the program's ability to adapt to changing data and parameters, the assignment of default actions to tasks, and a reduction in maintenance and support activities due to the elimination of “hardcoded” logic and values.

SAS Metadata Sources

SAS users have traditionally used the metadata results from PROC CONTENTS and PROC DATASETS to better understand the contents of SAS libraries and its various member types (i.e., DATA, INDEX, and VIEW); to construct user, system, and operations documentation; the creation of KEEP (or SELECT) lists of variables (or columns); automating one or more data sets variable attributes and characteristics; and other processing.

- **PROC CONTENTS** – Produces a directory of the SAS library and the details associated with each member type stored in a SAS library.
- **PROC DATASETS** – In Michael A. Raithel’s (2016) paper, titled “PROC DATASETS; The Swiss Army Knife of SAS® Procedures,” he shows why PROC DATASETS is one of the most versatile data management procedures. Like PROC CONTENTS, the PROC DATASETS CONTENTS statement produces a directory of the SAS library and the details associated with each member type (e.g., DATA, VIEW, INDEX) stored in a SAS library.

To understand the nature of the data structures and the associated metadata content of a SAS library (or database) and its member types, a PROC CONTENTS (or PROC DATASETS) can be specified. PROC CONTENTS displays summary information about the structure and member types of our SAS library (or database) environment. In the next example, a PROC CONTENTS is executed with the **DIRECTORY** option to display a list of the SAS files in the SAS library and the **NODS** option to suppress the contents of individual files when the **_ALL_** option is specified in the **DATA=** option.

PROC CONTENTS Code

```
PROC CONTENTS DATA=mydata._all_ directory nods;
RUN;
```

Results

The CONTENTS Procedure				
Directory				
#	Name	Member Type	File Size	Last Modified
1	CUSTOMERS	DATA	5KB	06/04/2004 00:08:08
2	CUSTOMERS\$2	DATA	5KB	04/18/2002 00:09:00
3	CUSTOMERS_BACKUP	DATA	5KB	08/18/2004 09:32:44
4	CUSTOMERS_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:18
5	INVENTORY	DATA	9KB	03/28/2002 20:38:40
6	INVENTORY_VIEW	VIEW	5KB	08/25/2002 21:14:06
7	INVOICE	DATA	5KB	11/02/1999 05:30:36
8	INVOICE_1K_VIEW	VIEW	5KB	08/21/2002 23:10:10
9	JOINED_VIEW	VIEW	5KB	08/21/2002 23:16:44
10	LAPTOP_DISCOUNT_VIEW	VIEW	5KB	08/23/2002 09:40:22
11	LAPTOP_PRODUCTS_VIEW	VIEW	5KB	08/23/2002 00:48:20
12	LARGEST_AMOUNT_VIEW	VIEW	5KB	08/21/2002 23:24:38
13	MANUFACTURERS	DATA	5KB	07/22/2004 01:19:50
14	MANUFACTURERS_VIEW	VIEW	5KB	08/25/2002 21:14:06
15	MANUFACTURERS_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:20
16	PRODUCTS	DATA	17KB	07/30/2004 07:18:00
	PRODUCTS	INDEX	13KB	07/30/2004 07:18:00
17	PRODUCTS_VIEW	VIEW	5KB	08/25/2002 21:14:06
18	PRODUCTS_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:20
19	PRODUCTS_WITH_NULLS	DATA	9KB	06/24/2013 11:48:06
20	PURCHASES	DATA	5KB	04/18/2011 09:17:24
21	PURCHASES\$2	DATA	5KB	08/13/2013 22:23:00
22	PURCHASES_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:20
23	SOFTWARE_PRODUCTS	DATA	5KB	08/16/2004 21:35:18
24	SOFTWARE_PRODUCTS_TAX_VIEW	VIEW	5KB	08/23/2002 00:30:40
25	SOFTWARE_PRODUCTS_VIEW	VIEW	5KB	08/21/2002 20:44:36
26	VIEW_CUSTOMERS	VIEW	5KB	08/08/2002 22:58:34
27	WORKSTATION_PRODUCTS_VIEW	VIEW	5KB	08/21/2002 23:05:08

In the next example, a PROC CONTENTS is executed with the **MEMTYPE=DATA** option to list the names of the SAS data sets that reside in the user-assigned MYDATA SAS library.

PROC CONTENTS Code

```
proc contents data=mydata._all_
            memtype=data
            directory
            nobs;
run;
```

Results

The CONTENTS Procedure				
Directory				
#	Name	Member Type	File Size	Last Modified
1	CUSTOMERS	DATA	5KB	06/04/2004 00:08:08
2	CUSTOMERS2	DATA	5KB	04/18/2002 00:09:00
3	CUSTOMERS_BACKUP	DATA	5KB	08/18/2004 09:32:44
4	CUSTOMERS_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:18
5	INVENTORY	DATA	9KB	03/28/2002 20:39:40
6	INVOICE	DATA	5KB	11/02/1999 05:30:36
7	MANUFACTURERS	DATA	5KB	07/22/2004 01:19:50
8	MANUFACTURERS_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:20
9	PRODUCTS	DATA	17KB	07/30/2004 07:18:00
	PRODUCTS	INDEX	13KB	07/30/2004 07:18:00
10	PRODUCTS_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:20
11	PRODUCTS_WITH_NULLS	DATA	9KB	06/24/2013 11:48:06
12	PURCHASES	DATA	5KB	04/18/2011 09:17:24
13	PURCHASES2	DATA	5KB	08/13/2013 22:23:00
14	PURCHASES_WITH_MESSY_DATA	DATA	128KB	07/12/2018 12:52:20
15	SOFTWARE_PRODUCTS	DATA	5KB	08/16/2004 21:35:18

In the next PROC CONTENTS example, the VARNUM option is specified to display the specific SAS library metadata content associated with data sets, columns (or variables), and column attributes, to better understand the SAS data environment.

PROC CONTENTS Code

```
proc contents data=mydata._all_
    memtype=data
    varnum;
run;
```

Results

The CONTENTS Procedure

Data Set Name	MYDATA.CUSTOMERS	Observations	18
Member Type	DATA	Variables	3
Engine	V9	Indexes	0
Created	04/17/2002 16:15:20	Observation Length	48
Last Modified	08/03/2004 16:08:08	Deleted Observations	4
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	WINDOWS_32		
Encoding	Default		

Engine/Host Dependent Information

Data Set Page Size	4096
Number of Data Set Pages	1
First Data Page	1
Max Obs per Page	84
Obs in First Data Page	22
Number of Data Set Repairs	0
Filename	/folders/myfolders/SAS Tables - PROC SQL 3rd Edition/customers.sas7bdat
Release Created	8.0202M0
Host Created	WN_PRO
Inode Number	77191
Access Permission	rwxrwx---
Owner Name	root
File Size	5KB
File Size (bytes)	5120

Variables in Creation Order

#	Variable	Type	Len	Label
1	custnum	Num	3	Customer Number
2	custname	Char	25	Customer Name
3	custcity	Char	20	Customer's Home City

The CONTENTS Procedure				
Data Set Name	MYDATA.SOFTWARE_PRODUCTS	Observations	4	
Member Type	DATA	Variables	4	
Engine	V9	Indexes	0	
Created	08/16/2004 13:35:17	Observation Length	48	
Last Modified	08/16/2004 13:35:17	Deleted Observations	0	
Protection		Compressed	NO	
Data Set Type		Sorted	NO	
Label				
Data Representation	WINDOWS_32			
Encoding	wlatin1 Western (Windows)			

Engine/Host Dependent Information				
Data Set Page Size	4096			
Number of Data Set Pages	1			
First Data Page	1			
Max Obs per Page	84			
Obs in First Data Page	4			
Number of Data Set Repairs	0			
Filename	/folders/myfolders/SAS Tables - PROC SQL 3rd Edition/software_products.sas7bdat			
Release Created	9.0101M0			
Host Created	XP_HOME			
Inode Number	77187			
Access Permission	rwxrwx---			
Owner Name	root			
File Size	5KB			
File Size (bytes)	5120			

Variables in Creation Order					
#	Variable	Type	Len	Format	Label
1	prodnum	Num	3		Product Number
2	prodname	Char	25		Product Name
3	prodtype	Char	15		Product Type
4	prodcost	Num	5	DOLLAR9.2	Product Cost

In the next example, a PROC CONTENTS is specified to collect the CUSTOMERS SAS data set's metadata and save the results to a user-assigned SAS data set using the OUT= option. The resulting SAS data set contains one observation for each variable found in the original data set. The SAS data set contains information about the original SAS data set and specific information about each variable. A PROC SQL SELECT query is then specified to view the data set metadata content.

PROC CONTENTS and PROC SQL Code

```

PROC CONTENTS DATA=MYDATA.Customers
              OUT=WORK.Customers_Contents_Structure
              NOPRINT;
RUN;

PROC SQL;
  SELECT *
  FROM WORK.Customers_Contents_Structure;
QUIT;

```

Results

Library Name	Library Member Name	Data Set Label	Special Data Set Type (From TYPE=)	Variable Name	Variable Type	Variable Length	Variable Number	Variable Label	Variable Format	Format Length	Number of Format Decimals	Variable Informat
MYDATA	CUSTOMERS			custcity	2	20	3	Customer's Home City		0	0	
MYDATA	CUSTOMERS			custname	2	25	2	Customer Name		0	0	
MYDATA	CUSTOMERS			custnum	1	3	1	Customer Number		0	0	

Informat Length	Number of Informat Decimals	Justification	Position in Buffer	Observations in Data Set	Engine Name	Create Date	Last Modified Date	Deleted Observations in Data Set	Use of Variable in Indexes	Library Member Type	Number of Indexes for Data Set	Password Protection (Read Write Alter)
0	0	0	25	18	V9	17APR02:16:15:20	03JUN04:16:08:08	4	NONE	DATA	0	---
0	0	0	0	18	V9	17APR02:16:15:20	03JUN04:16:08:08	4	NONE	DATA	0	---
0	0	1	45	18	V9	17APR02:16:15:20	03JUN04:16:08:08	4	NONE	DATA	0	---

Update Flags (Protect Contribute Add)	Compression Routine	Reuse Space	Sorted and/or Validated	Position of Variable in Sortedby Clause	Host Character Set	Collating Sequence	Sort Option: No Duplicate Keys	Sort Option: No Duplicate Records	Encryption Routine	Point to Observations	Maximum Number of Generations	Generation Number	Next Generation Number	Character Variables Transcoded
--	NO	NO	.	.			NO	NO	NO	YES	0	.	.	YES
--	NO	NO	.	.			NO	NO	NO	YES	0	.	.	YES
--	NO	NO	.	.			NO	NO	NO	YES	0	.	.	YES

Finally, the next example specifies the ODS OUTPUT destination to collect and save the results of a SAS data set's metadata with the ATTRIBUTES= parameter. The resulting temporary SAS data set contains the attribute information such as data set name, the date the data set was created, the date the data set was last modified, the number of observations, the number of variables, and much more from each data set contained in the 'MYDATA' SAS library. A simple PROC SQL SELECT query is then specified to view the SAS data set metadata attribute content.

PROC CONTENTS and PROC SQL Code

```

ODS OUTPUT ATTRIBUTES=work.Data_Set_Attributes;
PROC CONTENTS DATA=MYDATA._ALL_
      MEMTYPE=DATA;
RUN;

PROC SQL;
  SELECT *
  FROM WORK.Data_Set_Attributes;
QUIT;

```

Results

Member	Label1	cValue1	nValue1	Label2	cValue2	nValue2
MYDATA.CUSTOMERS	Data Set Name	MYDATA.CUSTOMERS	.	Observations	18	18.000000
MYDATA.CUSTOMERS	Member Type	DATA	.	Variables	3	3.000000
MYDATA.CUSTOMERS	Engine	V9	.	Indexes	0	0
MYDATA.CUSTOMERS	Created	04/17/2002 16:15:20	1334679320	Observation Length	48	48.000000
MYDATA.CUSTOMERS	Last Modified	08/03/2004 16:08:08	1401898088	Deleted Observations	4	4.000000
MYDATA.CUSTOMERS	Protection		.	Compressed	NO	.
MYDATA.CUSTOMERS	Data Set Type		.	Sorted	NO	.
MYDATA.CUSTOMERS	Label		.			0
MYDATA.CUSTOMERS	Data Representation	WINDOWS_32	.			0
MYDATA.CUSTOMERS	Encoding	Default	.			0
MYDATA.CUSTOMERS2	Data Set Name	MYDATA.CUSTOMERS2	.	Observations	8	8.000000
MYDATA.CUSTOMERS2	Member Type	DATA	.	Variables	3	3.000000
MYDATA.CUSTOMERS2	Engine	V9	.	Indexes	0	0
MYDATA.CUSTOMERS2	Created	04/17/2002 16:07:11	1334678831	Observation Length	48	48.000000
MYDATA.CUSTOMERS2	Last Modified	04/17/2002 16:08:59	1334678939	Deleted Observations	19	19.000000
MYDATA.CUSTOMERS2	Protection		.	Compressed	NO	.
MYDATA.CUSTOMERS2	Data Set Type		.	Sorted	NO	.
MYDATA.CUSTOMERS2	Label		.			0
MYDATA.CUSTOMERS2	Data Representation	WINDOWS_32	.			0
MYDATA.CUSTOMERS2	Encoding	Default	.			0

MYDATA.PURCHASES_WITH_MESSY_DATA	Data Set Name	MYDATA.PURCHASES_WITH_MESSY_DATA	.	Observations	57	57.000000
MYDATA.PURCHASES_WITH_MESSY_DATA	Member Type	DATA	.	Variables	4	4.000000
MYDATA.PURCHASES_WITH_MESSY_DATA	Engine	V9	.	Indexes	0	0
MYDATA.PURCHASES_WITH_MESSY_DATA	Created	07/12/2018 03:52:19	1846988739	Observation Length	16	16.000000
MYDATA.PURCHASES_WITH_MESSY_DATA	Last Modified	07/12/2018 03:52:19	1846988739	Deleted Observations	0	0
MYDATA.PURCHASES_WITH_MESSY_DATA	Protection		.	Compressed	NO	.
MYDATA.PURCHASES_WITH_MESSY_DATA	Data Set Type		.	Sorted	NO	.
MYDATA.PURCHASES_WITH_MESSY_DATA	Label		.			0
MYDATA.PURCHASES_WITH_MESSY_DATA	Data Representation	SOLARIS_X86_64_LINUX_X86_64_ALPHATRUE,LINUX_IA64	.			0
MYDATA.PURCHASES_WITH_MESSY_DATA	Encoding	utf-8 Unicode (UTF-8)	.			0
MYDATA.SOFTWARE_PRODUCTS	Data Set Name	MYDATA.SOFTWARE_PRODUCTS	.	Observations	4	4.000000
MYDATA.SOFTWARE_PRODUCTS	Member Type	DATA	.	Variables	4	4.000000
MYDATA.SOFTWARE_PRODUCTS	Engine	V9	.	Indexes	0	0
MYDATA.SOFTWARE_PRODUCTS	Created	08/18/2004 13:35:17	1408282517	Observation Length	48	48.000000
MYDATA.SOFTWARE_PRODUCTS	Last Modified	08/18/2004 13:35:17	1408282517	Deleted Observations	0	0
MYDATA.SOFTWARE_PRODUCTS	Protection		.	Compressed	NO	.
MYDATA.SOFTWARE_PRODUCTS	Data Set Type		.	Sorted	NO	.
MYDATA.SOFTWARE_PRODUCTS	Label		.			0
MYDATA.SOFTWARE_PRODUCTS	Data Representation	WINDOWS_32	.			0
MYDATA.SOFTWARE_PRODUCTS	Encoding	wlatin1 Western (Windows)	.			0

DICTIONARY Tables

PROC SQL users can quickly and conveniently obtain useful information about their SAS session with a number of read-only SAS system tables called DICTIONARY tables. At any time during a SAS session, DICTIONARY tables can be accessed using the libref DICTIONARY in the FROM clause of a PROC SQL SELECT statement to capture information related to currently defined librefs, table names, column names, column attributes, formats, and more. The first notable characteristic about the libref DICTIONARY as it relates to DICTIONARY tables is that it can only be specified in the FROM clause of PROC SQL – and is not allowed as a libref in any other SAS procedure or in the DATA step. The second notable characteristic of the libref DICTIONARY is that it seems to hold special “super powers” where the maximum length of a libref anywhere else in the SAS System is limited (or constrained) to eight characters.

SAS 9.1 software supported 22 DICTIONARY tables and SASHELP views, SAS 9.2 supported 29 DICTIONARY tables and SASHELP views, SAS 9.3 supported 30 DICTIONARY tables and SASHELP views, and SAS 9.4 supports 32 DICTIONARY tables and SASHELP views. It should be noted that the content contained in the various DICTIONARY tables can also be found and accessed from the SASHELP views using any of your favorite procedures or in the DATA step.

Accessing and Displaying the Number of Rows in Tables

The DICTIONARY table, TABLES, can be accessed to capture and display each table name and the number of observations in the user-assigned MYDATA libref. The following PROC SQL code provides a handy way to quickly determine the number of rows in one or all tables in a libref without having to execute multiple PROC CONTENTS statements by using the stored information in the DICTIONARY table, TABLES.

PROC SQL Code

```
PROC SQL;
  SELECT LIBNAME, MEMNAME, NOBS
    FROM DICTIONARY.TABLES
   WHERE LIBNAME      = "MYDATA"
     AND UPCASE(MEMTYPE) = "DATA";
QUIT;
```

Results

Library Name	Member Name	Number of Physical Observations
MYDATA	CUSTOMERS	22
MYDATA	CUSTOMERS2	27
MYDATA	CUSTOMERS_BACKUP	3
MYDATA	CUSTOMERS_WITH_MESSY_DATA	18
MYDATA	INVENTORY	8
MYDATA	INVOICE	7
MYDATA	MANUFACTURERS	6
MYDATA	MANUFACTURERS_WITH_MESSY_DATA	6
MYDATA	PRODUCTS	12
MYDATA	PRODUCTS_WITH_MESSY_DATA	10
MYDATA	PRODUCTS_WITH_NULLS	13
MYDATA	PURCHASES	57
MYDATA	PURCHASES2	7
MYDATA	PURCHASES_WITH_MESSY_DATA	57
MYDATA	SOFTWARE_PRODUCTS	4

Accessing and Displaying a Cross-reference List of a Variable Defined in All Tables

To retrieve and display a list of the table (data set) names containing a selected variable, you could execute one or more PROC CONTENTS statements against the selected tables.

Although this would produce the desired results, it often involves more effort and additional time mulling through piles of results to find the desired information. Or, using a more efficient approach, you could access the DICTIONARY table, COLUMNS, to produce a cross-reference listing of all the table names that contain the variable CUSTNUM in the user-assigned MYDATA libref, as shown below.

PROC SQL Code

```
PROC SQL;
  SELECT LIBNAME, MEMNAME, NAME, TYPE, LENGTH
    FROM DICTIONARY.COLUMNS
   WHERE LIBNAME      = "MYDATA"
     AND UPCASE(NAME) = "CUSTNUM"
     AND UPCASE(MEMTYPE) = "DATA";
QUIT;
```

Results

Library Name	Member Name	Column Name	Column Type	Column Length
MYDATA	CUSTOMERS	custnum	num	3
MYDATA	CUSTOMERS2	custnum	num	3
MYDATA	CUSTOMERS_BACKUP	custnum	num	8
MYDATA	CUSTOMERS_WITH_MESSY_DATA	custnum	num	3
MYDATA	INVOICE	custnum	num	3
MYDATA	PURCHASES	custnum	num	4
MYDATA	PURCHASES2	Custnum	num	8
MYDATA	PURCHASES_WITH_MESSY_DATA	custnum	num	4

Capturing a List of Variables from the COLUMNS Dictionary Table

In lieu of specifying a PROC CONTENTS statement, the DICTIONARY table, COLUMNS, can be accessed to capture and display a table's column names in any user-assigned libref. The following PROC SQL code provides a handy way to quickly capture and store the names of any columns contained in the CUSTOMERS table to a macro variable. To better visualize the type of data that is captured and stored in the user-defined macro variables, a %PUT statement is specified to display the contents of both macro variables on the SAS log.

PROC SQL Code

```
PROC SQL NOPRINT;
  SELECT NAME,
         COUNT(NAME)
    INTO :MVARIALES SEPARATED BY ', '
      ,:MVARIALENUM
   FROM DICTIONARY.COLUMNS
```

```

WHERE LIBNAME      = "MYDATA"
  AND UPCASE(MEMNAME) = "CUSTOMERS";
QUIT;
%PUT &MVARIALES &MVARIALESNUM;

```

SAS Log Results

```

%PUT &MVARIALES &MVARIALESNUM;
Custnum, Custname, Custity      3

```

The previous example can be expanded so that only the character-defined variables are saved in the macro variable. The next example illustrates how PROC SQL code captures the names of the character-defined columns contained in the CUSTOMERS table. The contents of the macro variable are then specified in a SELECT statement to produce a report. As in the previous example, a %PUT statement is specified to display the contents of the user-defined macro variable on the SAS log.

PROC SQL Code

```

PROC SQL NOPRINT;
  SELECT NAME
    INTO :MVARIALES SEPARATED BY ', '
    FROM DICTIONARY.COLUMNS
   WHERE LIBNAME      = "MYDATA"
     AND UPCASE(MEMNAME) = "CUSTOMERS"
     AND UPCASE(TYPE)    = "CHAR";
%PUT &MVARIALES;
RESET PRINT;
SELECT &MVARIALES FROM MYDATA.CUSTOMERS;
QUIT;

```

SAS Log Results

```

%PUT &MVARIALES;
Custname, Custcity

```

Results

Customer Name	Customer's Home City
La Mesa Computer Land	La Mesa
Vista Tech Center	Vista
Coronado Internet Zone	Coronado
La Jolla Computing	La Jolla
Alpine Technical Center	Alpine
Oceanside Computer Land	Oceanside
San Diego Byte Store	San Diego
Jamul Hardware & Software	Jamul
Del Mar Tech Center	Del Mar
Lakeside Software Center	Lakeside
Bonsall Network Store	Bonsall
Rancho Santa Fe Tech	Rancho Santa Fe
Spring Valley Byte Center	Spring Valley
Poway Central	Poway
Valley Center Tech Center	Valley Center
Fairbanks Tech USA	Fairbanks Ranch
Blossom Valley Tech	Blossom Valley
Chula Vista Networks	

CALL EXECUTE Routine

SAS users have a powerful DATA step routine called CALL EXECUTE that can be used for data-driven processing. The CALL EXECUTE routine accepts a single argument where the value can be a character-string or, when needed, a character expression containing SAS code elements to be executed after they are resolved. The CALL EXECUTE routine permits SAS statements and macro code to be stacked together and then executed.

When the CALL EXECUTE routine contains SAS statement code without macro variables or macro references, the code is appended to the input stack for immediate execution after the DATA step ends. The argument can be specified with single or double quotation marks, dynamically generating SAS code for execution. To leverage data-driven processes with CALL EXECUTE, a control data set called, Product_Types, is created using the PROC SQL CREATE TABLE and INSERT INTO statements with four distinct product types (i.e., “Laptop”, “Phone”, “Software”, and “Workstation”) represented as observations. The DATA step then reads the contents of the control data set populating the unique value for the Prodtype variable in the individual CALL EXECUTE statements. Note: The CATS function is used to strip blanks and concatenate multiple strings together. The CATX function is used to insert blanks ‘ ‘ and special characters between parameters.

In the next example, the PROC SQL step creates a control data set that contains the unique values for each product type (i.e., “Laptop”, “Phone”, “Software”, and “Workstation”). Then, a DATA _NULL_ step is specified to read the contents of the control data set followed by a series of CALL EXECUTE statements to generate and execute SAS code. The result is the creation of four Excel spreadsheets – one for each of the product types specified in the control data set.

Code

```
PROC SQL;
  CREATE TABLE work.Product_Types /* Control Data Set */
    ( Prodtype CHAR(15) LABEL='Product Type' );
  INSERT INTO work.Product_Types /* Populate Product Type Rows */
    SET Prodtype='Laptop'
    SET Prodtype='Phone'
    SET Prodtype='Software'
    SET Prodtype='Workstation';
  QUIT;

DATA _NULL_;
  SET work.Product_Types;
  CALL EXECUTE(CATS('ods Excel
    file="/folders/myfolders/', Prodtype, ' - Products_Rpt.xlsx"
      style=styles.barrettsblue
      options(embedded_titles="yes");'));
  CALL EXECUTE(CATX(' ', 'title ', Prodtype, ' Products;'));
  CALL EXECUTE('proc sql;');
  CALL EXECUTE(CATS('select * from mydata.Products(where=
                (Prodtype='',Prodtype,''))';));
  CALL EXECUTE('quit;');
  CALL EXECUTE('ods Excel close;');
RUN;
```

The dynamically generated SAS code produced by the preceding CALL EXECUTE statements is displayed below.

Partial SAS Log Results

```
1 + ods Excel
2 +   file="/folders/myfolders/Laptop-Products_Rpt.xlsx"
3 +   style=styles.barrettsblue
4 +   options(embedded_titles="yes");
5 + title Laptop Products;
6 + proc sql ;
7 + select * from mydata.Products(where=
8 +                               (Prodtype="Laptop"));
9 + quit;
10 + ods Excel close;

11 + ods Excel
12 +   file="/folders/myfolders/Phone-Products_Rpt.xlsx"
13 +   style=styles.barrettsblue
14 +   options(embedded_titles="yes");
15 + title Phone Products;
16 + proc sql;
17 + select * from mydata.Products(where=
18 +                               (Prodtype="Phone"));
```

```

19 + quit;
20 + ods Excel close;

21 + ods Excel
22 +      file="/folders/myfolders/Software-Products_Rpt.xlsx"
23 +      style=styles.barrettsblue
24 +      options(embedded_titles="yes");
25 +      title Software Products;
26 + proc sql;
27 + select * from mydata.Products(where=
28 +                               (Prodtype="Software"));
29 + quit;
30 + ods Excel close;

31 + ods Excel
32 +      file="/folders/myfolders/Workstation-Products_Rpt.xlsx"
33 +      style=styles.barrettsblue
34 +      options(embedded_titles="yes");
35 +      title Workstation Products;
36 + proc sql ;
37 + select * from mydata.Products(where=
38 +                               (Prodtype="Workstation"));
39 + quit;
40 + ods Excel close;

```

Results

	A	B	C	D	E
1	Laptop Products				
2					
3	Product Number	Manufacturer Product Name		Product Number	Product Type
4	1700	Travel Laptop		170	Laptop
					\$3,400.00

	A	B	C	D	E
1	Phone Products				
2					
3	Product Number	Manufacturer Product Name		Product Number	Product Type
4	2101	Analog Cell Phone		210	Phone
5	2102	Digital Cell Phone		210	Phone
6	2200	Office Phone		220	Phone
					\$35.00
					\$175.00
					\$130.00

	A	B	C	D	E
1	Software Products				
2					
3	Product Number Product Name		Manufacturer Number Product Type		Product Cost
4	5001 Spreadsheet Software		500 Software	\$299.00	
5	5002 Database Software		500 Software	\$399.00	
6	5003 Wordprocessor Software		500 Software	\$299.00	
7	5004 Graphics Software		500 Software	\$299.00	
	A	B	C	D	E
1	Workstation Products				
2					
3	Product Number Product Name		Manufacturer Number Product Type		Product Cost
4	1110 Dream Machine		111 Workstation	\$3,200.00	
5	1200 Business Machine		120 Workstation	\$3,300.00	

Custom-defined Formats

The FORMAT procedure is a powerful tool for building user-defined informats and formats. These “custom” user-defined informats and formats provide SAS with instructions on how to read data into SAS variables and write (or display) output. Custom-defined informats and formats are defined as temporary or permanent, and are stored as entries in SAS catalogs. The available operations performed by the FORMAT procedure include the ability to:

- Convert character values to numeric values
- Convert numeric values to character values
- Convert character values to other character values

To prevent hardcoding VALUE clauses, custom-defined formats can be dynamically created from a SAS data set. In a paper, Harry Droogendyk, argues that dynamically created custom-defined formats offer users a more efficient approach than processing sort, merge, and join operations by leveraging data-driven processes. Consequently, the FORMAT procedure is able to create informats and formats without specifying hardcoded INVALUE, PICTURE, or VALUE clauses by using a SAS control data set as input.

User-defined formats can be created from raw data or a SAS data set using PROC FORMAT’s CNTLIN= option. In order to create a user-defined format, we’ll need to construct a control data set with a few required variables. The control data set is specified with the CNTLIN= option of PROC FORMAT. To start the process, the control data set must have the following required variables:

- FMTNAME – specifies the name of a character variable whose value is the format or informat name.
- START – specifies the name of a character variable that contains the value to be converted.

- LABEL – specifies the name of a character variable that contains the converted value.

In the next example, a PROC SQL CREATE TABLE and INSERT INTO statements are specified to produce a control data set with the variables: FMTNAME, Start, and Label. The contents of the control data set are then displayed for verification purposes with a SELECT query. The control data set is then specified in the PROC FORMAT CNTLIN option. Finally, a SELECT query is processed using the FORMAT=\$Prodtype to produce the desired expanded results. Note: Readers can obviously use other approaches such as, a DATA step, to construct the control data set.

Code

```
PROC SQL;
  CREATE TABLE work.Product_Type_Control /* Control Data Set */
  (
    FMTNAME CHAR(9)   LABEL='Format Name',
    Start    CHAR(11)  LABEL='Product Type',
    Label    CHAR(20)  LABEL='Format Label' );
  INSERT INTO work.Product_Type_Control /* Populate Product Type Rows */
  SET FMTNAME = '$Prodtype',
      Start   = 'Laptop',
      Label   = 'Laptop Products';
  SET FMTNAME = '$Prodtype',
      Start   = 'Phone',
      Label   = 'Phone Products';
  SET FMTNAME = '$Prodtype',
      Start   = 'Software',
      Label   = 'Software Products';
  SET FMTNAME = '$Prodtype',
      Start   = 'Workstation',
      Label   = 'Workstation Products';

  /* Display the control data set */
  SELECT *
    FROM work.Product_Type_Control;
  QUIT;

  /* Create the user-defined format */
  PROC FORMAT LIBRARY=WORK CNTLIN=Product_Type_Control;
  RUN;

  /* Use the user-defined format to display the PRODTYPE value */
  PROC SQL;
    SELECT Prodnum,
           Prodname,
           Manunum,
           Prodtype FORMAT=$Prodtype,
           Prodcost
      FROM MYDATA.PRODUCTS;
  QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE TABLE work.Product_Type_Control /* Control Data Set */
  (
    FMTNAME CHAR(9)   LABEL='Format Name',
    Start    CHAR(11)  LABEL='Product Type',
```

```

      Label      CHAR(20)  LABEL='Format Label' );
NOTE: Table WORK.PRODUCT_TYPE_CONTROL created, with 0 rows and 3
columns.
      INSERT INTO work.Product_Type_Control /* Populate Product
Type Rows */
      SET FMTNAME = '$Prodtype',
          Start   = 'Laptop',
          Label    = 'Laptop Products'
      SET FMTNAME = '$Prodtype',
          Start   = 'Phone',
          Label    = 'Phone Products'
      SET FMTNAME = '$Prodtype',
          Start   = 'Software',
          Label    = 'Software Products'
      SET FMTNAME = '$Prodtype',
          Start   = 'Workstation',
          Label    = 'Workstation Products';
NOTE: 4 rows were inserted into WORK.PRODUCT_TYPE_CONTROL.

      SELECT *
         FROM work.Product_Type_Control;
      QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time            0.03 seconds
      cpu time             0.03 seconds

PROC FORMAT LIBRARY=WORK CNTLIN=Product_Type_Control;
NOTE: Format $PRODTYPE has been output.
      RUN;

NOTE: PROCEDURE FORMAT used (Total process time):
      real time            0.00 seconds
      cpu time             0.00 seconds

NOTE: There were 4 observations read from the data set
WORK.PRODUCT_TYPE_CONTROL.

PROC SQL;
      SELECT Prodnum,
             Prodname,
             Manunum,
             Prodtype FORMAT=$Prodtype.,
             Prodcost
        FROM MYDATA.PRODUCTS;
      QUIT;

```

Results

The results from processing the dynamically generated control data set against the PRODUCTS table are displayed below.

Format Name	Product Type	Format Label
\$Prodtype	Laptop	Laptop Products
\$Prodtype	Phone	Phone Products
\$Prodtype	Software	Software Products
\$Prodtype	Workstation	Workstation Products

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation Products	\$3,200.00
1200	Business Machine	120	Workstation Products	\$3,300.00
1700	Travel Laptop	170	Laptop Products	\$3,400.00
2101	Analog Cell Phone	210	Phone Products	\$35.00
2102	Digital Cell Phone	210	Phone Products	\$175.00
2200	Office Phone	220	Phone Products	\$130.00
5001	Spreadsheet Software	500	Software Products	\$299.00
5002	Database Software	500	Software Products	\$399.00
5003	Wordprocessor Software	500	Software Products	\$299.00
5004	Graphics Software	500	Software Products	\$299.00

Macro Language

The SQL procedure and the macro language are two versatile tools found in the Base SAS software. Combining the two together provides users with all the tools necessary to construct highly useful and effective data-driven programs. In this section, I will explain a data-driven approach to create multiple Excel files. Triggered by calling a macro to reduce coding requirements, the process uses the Macro language, PROC SQL, the ODS Excel destination, and PROC FREQ to send output (results) to Excel. The **ODS Excel Destination** became production in SAS 9.4 (M4). It serves as an interface between SAS and Excel. The ODS Excel features include:

- SAS Results and Output can be sent directly to Excel
- Offers a Flexible way to create Excel files
- Supports Reports, Tables, Statistics, and Graphs
- Formats Data into Excel Worksheet cells
- Permits Automation of Production-level Workbooks

The ODS Excel destination easily sends output and results to Excel. The ODS Excel syntax simplifies the process of sending output, reports, tables, statistics, and graphs to Excel files. The ODS Excel options are able to:

- Programmatically generate output and results
- Control font used and font sizes
- Add special features to row and column headers
- Adjust row and column sizes
- Format data values
- Align data to the left, center, or right
- Add hyperlinks for drill-down capability

Producing Multiple Excel Files

In the next example, a data-driven approach using PROC SQL SELECT code embedded inside a user-defined macro routine is constructed to dynamically produce separate Excel spreadsheets containing the frequency results for each unique BY-group (e.g., Product Type). The SELECT query processes the Products table, creates a single-value macro variable with the number of unique (distinct) product types, and a value-list macro variable with a list of the unique product types separated with a tilde “~”. The user-defined macro uses the FREQ procedure, both macro variables along with their respective values, an iterative macro %DO statement, a %SCAN function, and WHERE= data set option to dynamically send the results to an Excel spreadsheet for each BY-group.

Macro and PROC SQL Code

```
%macro multExcelfiles;
  proc sql noprint;
    select count(distinct prodtype)
      into :mprodtype_cnt /* number of unique product types */
      from MYDATA.PRODUCTS
        order by prodtype;
    select distinct prodtype
      into :mprodtype_lst separated by "~" /* list of product types */
      from MYDATA.PRODUCTS
        order by prodtype;
  quit;
  %do i=1 %to &mprodtype_cnt;
    ods Excel file="/folders/myfolders/%SCAN(&mprodtype_lst,&i,~)-Rpt.xlsx"
      style=styles.barrettsblue
      options(embedded_titles="yes");
    title "%SCAN(&mprodtype_lst,&i,~) Products";
    proc freq data=MYDATA.PRODUCTS(where=
      (prodtype="%SCAN(&mprodtype_lst,&i,~)"));
      tables Prodname;
    run;
    ods Excel close;
  %end;
  %put &mprodtype_lst;
%mend multExcelfiles;

%multExcelfiles;
```

The dynamically generated SAS code produced by the iterative %DO statement is displayed below.

SAS Log Results

```
%macro multExcelfiles;
  proc sql noprint;
    select count(distinct prodtype)
      into :mproptype_cnt /* number of unique product types */
      from MYDATA.PRODUCTS
      order by prodtype;
    select distinct prodtype
      into :mproptype_lst separated by "~" /* list of product
types */
      from MYDATA.PRODUCTS
      order by prodtype;
  quit;
%do i=1 %to &mproptype_cnt;
  ods Excel
  file="/folders/myfolders/%SCAN(&mproptype_lst,&i,~)-Rpt.xlsx"
    style=styles.barrettsblue
    options(embedded_titles="yes");
  title "%SCAN(&mproptype_lst,&i,~) Products";
  proc freq data=MYDATA.PRODUCTS(where=
    (prodtype="%SCAN(&mproptype_lst,&i,~)"));
    tables Prodname;
  run;
  ods Excel close;
%end;
%put &mproptype_lst;
%mend multExcelfiles;
%multExcelfiles;

NOTE: PROCEDURE SQL used (Total process time):
      real time            0.02 seconds
      cpu time             0.01 seconds

NOTE: There were 1 observations read from the data set
MYDATA.PRODUCTS.
      WHERE prodtype='Laptop';
NOTE: PROCEDURE FREQ used (Total process time):
      real time            0.08 seconds
      cpu time             0.08 seconds

NOTE: Writing EXCEL file: /folders/myfolders/Laptop-Rpt.xlsx

NOTE: There were 3 observations read from the data set
MYDATA.PRODUCTS.
      WHERE prodtype='Phone';
NOTE: PROCEDURE FREQ used (Total process time):
      real time            0.04 seconds
      cpu time             0.04 seconds

NOTE: Writing EXCEL file: /folders/myfolders/Phone-Rpt.xlsx

NOTE: There were 4 observations read from the data set
MYDATA.PRODUCTS.
      WHERE prodtype='Software';
```

```

NOTE: PROCEDURE FREQ used (Total process time):
      real time            0.04 seconds
      cpu time             0.03 seconds

NOTE: Writing EXCEL file: /folders/myfolders/Software-Rpt.xlsx

NOTE: There were 2 observations read from the data set
MYDATA.PRODUCTS.
      WHERE prodtype='Workstation';

NOTE: PROCEDURE FREQ used (Total process time):
      real time            0.04 seconds
      cpu time             0.03 seconds

NOTE: Writing EXCEL file: /folders/myfolders/Workstation-Rpt.xlsx

Laptop~Phone~Software~Workstation

```

Results

Four Excel spreadsheets are produced, as shown below.

	A	B	C	D	E
1	Laptop Products				
2					
3	<i>The FREQ Procedure</i>				
4					
5	Product Name				
6	prodname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
7	Travel Laptop	1	100.00	1	100.00

	A	B	C	D	E
1	Phone Products				
2					
3	<i>The FREQ Procedure</i>				
4					
5	Product Name				
6	prodname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
7	Analog Cell Phone	1	33.33	1	33.33
8	Digital Cell Phone	1	33.33	2	66.67
9	Office Phone	1	33.33	3	100.00

	A	B	C	D	E
1	Software Products				
2					
3	<i>The FREQ Procedure</i>				
4					
5	Product Name				
6	prodname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
7	Database Software	1	25.00	1	25.00
8	Graphics Software	1	25.00	2	50.00
9	Spreadsheet Software	1	25.00	3	75.00
10	Wordprocessor Software	1	25.00	4	100.00

	A	B	C	D	E
1	Workstation Products				
2	<i>The FREQ Procedure</i>				
3					
4					
5	Product Name				
6	prodname	Frequency	Percent	Cumulative Frequency	Cumulative Percent
7	Business Machine	1	50.00	1	50.00
8	Dream Machine	1	50.00	2	100.00

Summary

Unlike procedural programming languages where a program's flow of execution is described using a detailed step-by-step logical approach to solving a problem or like object-oriented programming where an object is told how to behave, data-driven programming involves writing code that has its decisions and processes (the flow of execution) controlled (or dictated) by the data (or data structures). Data-driven programming offers SAS users many advantages over rival programming paradigms including enhanced flexibility and easier maintenance due to a reduction, or elimination, of "hardcoded" values.

The first data-driven programming technique uses metadata DICTIONARY tables to provide valuable information about SAS libraries, data sets, columns and attributes, catalogs, indexes, macros, system options, and views. The second data-driven programming technique uses the CALL EXECUTE routine to process (or execute) code generated by a DATA step. The third data-driven programming technique creates a user-defined format from a control data set. The fourth data-driven programming technique uses the SQL procedure and the macro language to construct an automated iterative data-driven process.

Chapter 11: Troubleshooting and Debugging

Introduction	365
The World of Bugs	365
The Debugging Process	366
Types of Problems	367
Troubleshooting and Debugging Techniques	368
Validating Queries with the VALIDATE Statement.....	368
Documented PROC SQL Options and Statement.....	369
Undocumented PROC SQL Options	382
Macro Variables	383
Troubleshooting and Debugging Examples	385
Summary	389

Introduction

When it comes to tracking down the source of SQL coding problems, users can be very resourceful. To find all those pesky errors, warnings, notes, and unexpected results, you need to carefully inspect the SAS log and telltale clues from code reviews and sample runs. Using reliable troubleshooting and debugging techniques will help get to the root of any syntax, data, system-related, and logic errors that are or could be problematic to the successful execution of a PROC SQL program.

This chapter introduces a number of strategies and techniques for troubleshooting and debugging SQL procedure coding problems. The guidelines presented in this chapter will help you learn about the different types of bugs, the steps involved in troubleshooting problems, the types of errors that can occur, and the options and statements that are available in the SQL procedure to troubleshoot and debug problems.

The World of Bugs

Since the birth of the software industry, software problems (often referred to as *bugs*) have been created by programmers (and users) of all skill levels in virtually every conceivable form. A bug is something the software or a program does that it is not supposed to do. When the software encounters a bug, it can cause the software to cease to operate or to misbehave.

Bugs come in all forms, shapes, and sizes. A problem that occurs while you're using the SQL procedure incorrectly and directly violating the rules of the language is referred to as a *usage error*. The severity of these types of errors depends on the nature of the violation as well as the effect on the user. Generally speaking though, a usage error results in a syntax error and the stoppage of the program or step.

Other types of bugs can be problematic, too. Bugs such as resource problems, data dependencies, and implementation errors can cause a program step to stop or produce unreliable results, which will then display errors or warnings in the SAS log.

The Debugging Process

The debugging process consists of a number of recommended steps to correct an identified problem and verify that it does not reappear. The objective is to identify, classify, fix, and verify a problem in a PROC SQL step, program, or application as quickly and easily as possible. Not every problem requires the application of each step recommended in this chapter. Rather, you can pick and choose from the suggestions to effectively correct problems and make sure they do not reappear. Adhering to a methodical and effective debugging process increases the likelihood that problems are identified and fixed correctly, thereby expediting and improving the way you handle problems.

To further explore this topic, see *Effective Methods for Software Testing* by William E. Perry (Wiley Publishing, Inc., 2006) and *The Science of Debugging* by Matthew A. Telles and Yuan Hsieh (The Coriolis Group, 2001).

The debugging process consists of five steps. See Table 11.1.

Table 11.1: Debugging Process Steps and Tasks

Debugging Step	Task Description
Problem Identification	<ol style="list-style-type: none"> Determine if a bug exists in the code. Determine what the problem is by playing detective. Describe why the problem is a bug. Determine what the code should do. Determine what the code is doing.
Information Collection	<ol style="list-style-type: none"> Collect user comments and feedback. Collect personal observations and symptoms. Review SAS log information. Collect test case(s) that highlight the problem. Capture environmental information (for example, system settings, operating environment, external elements, etc.).
Problem Assessment & Classification	<ol style="list-style-type: none"> Develop a theory about what caused the problem. Review the code. Classify the problem into one of the following categories: <ul style="list-style-type: none"> ◦ Requirements problem ◦ Syntax error ◦ CPU problem

Debugging Step	Task Description
Problem Resolution	<ul style="list-style-type: none"> ◦ Memory problem ◦ Storage problem ◦ I/O problem ◦ Logic problem <ol style="list-style-type: none"> 1. Propose a solution. 2. Describe why the solution will fix the problem. 3. Verify that the solution will not cause additional problems. 4. Fix the problem by implementing the solution.
Validate Solution	<ol style="list-style-type: none"> 1. Verify whether the problem still exists. 2. Verify whether the problem can be recreated or reproduced. 3. Determine whether other methods can cause the same problem to occur. 4. Verify that the solution does not cause other problems.

Types of Problems

The leading causes of SQL programming problems include misusing the language syntax; referencing data, especially column data, incorrectly; ignoring or incorrectly specifying system parameters; and constructing syntactically correct but illogical code that does not produce results as expected.

Usage errors can cause SAS to stop processing, produce warnings, or produce unexpected results. Table 11.2 illustrates the four types of usage errors and briefly describes each.

Table 11.2: Types of Usage Errors

Problem	Description
Syntax	Syntax problems are a result of one or more violations of the SQL procedure language constructs. These problems can prevent a program from processing until you make the required changes. For example, a variable that is referenced in a SELECT statement but is not found in the referenced table causes the program to stop.
Data	Data problems are a result of an inconsistency between the data and the program specification. These problems can prevent a program from processing, but might allow processing to continue and result in the assignment of missing, incomplete, or unreliable data. For example, missing values might be generated in a column when character data is incorrectly referenced as numeric data.

Problem	Description
System-related	System-related problems frequently result from specifying incompatible system options or choosing the wrong system option values. These problems can prevent a program from processing, but most frequently permit processing to occur with unsatisfactory results. For example, forgetting to specify a title statement has little effect on the production of output, but it might force the program to be rerun after the addition of one or more titles.
Logic	Logic problems are frequently the result of not specifying a coding condition correctly. For example, specifying an OR condition when an AND condition is needed might produce wrong results without any warnings, errors, or notes.

Other sources of problems can occur in a program for a variety of reasons. Table 11.3 illustrates these other problem sources and briefly describes each.

Table 11.3: Other Types of Errors

Problem	Description
Feature Creep	Additional features can creep into a program or application during the design, implementation, or testing phases. This can create a greater likelihood of problems.
Solution Complexity	More complex or esoteric solutions can also translate into problems. Difficult-to-maintain programs can result.
Requirements	Requirements might be inadequately stated, misunderstood, or omitted. In these situations, programs might not meet the needs of the user community and be classified as bugs.
Testing Environment	Inadequate testing environments, poor test plans, or insufficient test time often lead to bug-filled programs. In these situations, bugs can slip through the testing phase and into the production environment.

Troubleshooting and Debugging Techniques

PROC SQL provides numerous troubleshooting and debugging capabilities for the practitioner to choose from. From statements to options to macro variables, you can use the various techniques to control the process of finding and gathering information about SQL procedure coding problems quickly and easily.

Validating Queries with the VALIDATE Statement

The SQL procedure syntax checker identifies syntax errors before any data is processed. Syntax checking is automatically turned on in the SQL procedure without any statements or options being specified. But to enable syntax checking without automatically executing a

step, you can specify a VALIDATE statement at the beginning of a SELECT statement or, as will be presented later, specify the NOEXEC option.

The VALIDATE statement is available for SAS users to control what SELECT statement is checked. Because you specify a VALIDATE statement before a SELECT statement, you are better able to control the process of debugging code. A message that indicates syntax correctness is automatically displayed in the SAS log when code syntax is valid. Otherwise, an error message is displayed that identifies the coding violation. In the next example, a VALIDATE statement is specified at the beginning of a SELECT statement to enable syntax checking without code execution. The SAS log shows that the code contains valid syntax and automatically displays a message to that effect.

SQL Code

```
PROC SQL;
  VALIDATE
    SELECT *
      FROM PRODUCTS
        WHERE PRODTYPE = 'Software';
  QUIT;
```

SAS Log Results

```
PROC SQL;
  VALIDATE
    SELECT *
      FROM PRODUCTS
        WHERE PRODTYPE = 'Software';
  NOTE: PROC SQL statement has valid syntax.
  QUIT;
```

Documented PROC SQL Options and Statement

This section illustrates examples of widely used and documented troubleshooting and debugging SQL options along with the RESET statement. A description and a working example of each option and statement are presented below.

FEEDBACK/NOFEEDBACK Option

The FEEDBACK option displays additional documentation with SELECT queries. When specified, this option expands a SELECT * (wildcard) statement into a list of columns that it represents by displaying the names of each column in the underlying table(s) as well as any resolved macro values and macro variables. The column display order of a SELECT * statement is determined by the order defined in the table's record descriptor.

The following example illustrates the FEEDBACK option. Because a SELECT * statement does not automatically display the columns it represents, it might be important to expand the individual column names by specifying the FEEDBACK option. This becomes particularly useful for determining whether a desired column is present in the output and available for documentation purposes. The results of the expanded list of columns are displayed in the SAS log.

SQL Code

```
PROC SQL FEEDBACK;
  SELECT *
    FROM PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL FEEDBACK;
  SELECT *
    FROM PRODUCTS;
NOTE: Statement transforms to:
  select PRODUCTS.prodnum, PRODUCTS.prodname, PRODUCTS.manunum,
        PRODUCTS.prodtype, PRODUCTS.prodcost
      from WORK.PRODUCTS;
QUIT;
```

The FEEDBACK option can be particularly helpful in determining the column order when joining two or more tables. The next example illustrates the expansion of the columns in the SELECT * statement in a two-way equijoin. The FEEDBACK option displays all the columns in both tables.

SQL Code

```
PROC SQL FEEDBACK;
  SELECT *
    FROM PRODUCTS, MANUFACTURERS
   WHERE PRODUCTS.MANUNUM = MANUFACTURERS.MANUNUM AND
         MANUFACTURERS.MANUNAME = 'KPL Enterprises';
QUIT;
```

SAS Log Results

```
PROC SQL FEEDBACK;
  SELECT *
    FROM PRODUCTS, MANUFACTURERS
   WHERE PRODUCTS.MANUNUM = MANUFACTURERS.MANUNUM AND
         MANUFACTURERS.MANUNAME = 'KPL Enterprises';
NOTE: Statement transforms to:
  select PRODUCTS.prodnum, PRODUCTS.prodname, PRODUCTS.manunum,
        PRODUCTS.prodtype, PRODUCTS.prodcost,
        MANUFACTURERS.manunum,
        MANUFACTURERS.manuname, MANUFACTURERS.manucity,
        MANUFACTURERS.manustat
      from PRODUCTS, MANUFACTURERS
     where (PRODUCTS.manunum=MANUFACTURERS.manunum) and
           (MANUFACTURERS.manuname='KPL Enterprises');
QUIT;
```

The FEEDBACK option can also be used to display macro value and macro variable resolution. The next example shows the macro resolution of the macro variables &LIB, &TABLE, and &GROUPBY for debugging purposes.

SQL Code

```
%MACRO DUPS(LIB, TABLE, GROUPBY);
  PROC SQL FEEDBACK;
    SELECT &GROUPBY, COUNT(*) AS Duplicate_Rows
    FROM &LIB..&TABLE
    GROUP BY &GROUPBY
    HAVING COUNT(*) > 1;
  QUIT;
%MEND DUPS;
%DUPS (WORK, PRODUCTS, PRODTYPE);
```

SAS Log Results

```
%MACRO DUPS(LIB, TABLE, GROUPBY);
  PROC SQL FEEDBACK;
    SELECT &GROUPBY, COUNT(*) AS Duplicate_Rows
    FROM &LIB..&TABLE
    GROUP BY &GROUPBY
    HAVING COUNT(*) > 1;
  QUIT;
%MEND DUPS;
%DUPS (WORK, PRODUCTS, PRODTYPE);
NOTE: Statement transforms to:
  select PRODUCTS.prodtype, COUNT(*) as Duplicate_Rows
  from WORK.PRODUCTS
  group by PRODUCTS.prodtype
  having COUNT(*)>1;
```

You can also specify a %PUT statement instead of the FEEDBACK option to display the values of macro variables after macro resolution. The next example illustrates inserting the macro statement %PUT LIB = &LIB TABLE = &TABLE GROUPBY = &GROUPBY between the QUIT and %MEND statements to produce the results illustrated below.

SQL Code

```
%MACRO DUPS(LIB, TABLE, GROUPBY);
  PROC SQL;
    SELECT &GROUPBY, COUNT(*) AS Duplicate_Rows
    FROM &LIB..&TABLE
    GROUP BY &GROUPBY
    HAVING COUNT(*) > 1;
  QUIT;
  %PUT LIB = &LIB TABLE = &TABLE GROUPBY = &GROUPBY;
%MEND DUPS;
%DUPS (WORK, PRODUCTS, PRODTYPE);
```

SAS Log Results

```
%MACRO DUPS(LIB, TABLE, GROUPBY);
  . . . code not shown . . .
  LIB = WORK TABLE = PRODUCTS GROUPBY = PRODTYPE
```

_METHOD and _TREE Options with MSGLEVEL=I

As a troubleshooting and debugging tools, PROC SQL supports the _METHOD and _TREE options. These options are extremely useful for understanding the hierarchy of processing methods selected by the SQL optimizer and the internal form of the query plan, respectively. To help users understand the specific codes that are generated and displayed in the SAS log from the _METHOD option, a table of codes along with a brief description appears in Table 11.4.

When combined with the _METHOD and _TREE options, the MSGLEVEL=I System option displays additional messages in the SAS log that pertain to index usage, merge processing, and sort utility used.

Table 11.4: Codes That Are Generated and Displayed in the SAS Log from the _METHOD Option

Code	Description
SQXCRTA	Create table as Select.
SQXSLCT	Select statement or clause.
SQXJSL	Step loop join (Cartesian).
SQXJM	Merge join operation.
SQXJNDX	Index join operation.
SQXJHSH	Hash join operation.
SQXSORT	Sort operation.
SQXSRC	Source rows from table.
SQXFIL	Rows filtration.
SQXSUMG	Summary stats (aggregates) with GROUP BY clause.
SQXSUMN	Summary stats with no GROUP BY clause.

SQL Code

```
OPTIONS MSGLEVEL=I;
PROC SQL _METHOD _TREE;
  SELECT sum(inventory.invenqty)
    AS Products_Ordered_Before_09012000
   FROM PRODUCTS P,
        INVOICE I,
        CUSTOMERS C,
        INVENTORY I2
  WHERE I2.orddate < mdy(09,01,00) AND
        P.prodnum = I.prodnum AND
        I.custnum = C.custnum AND
        I.prodnum = I2.prodnum;
QUIT;
```

SAS Log Results

```

OPTIONS MSGLEVEL=I;
PROC SQL _METHOD_ TREE;
  SELECT sum(inventory.invenqty)
        AS Products_Ordered_Before_09012000
    FROM PRODUCTS      P,
         INVOICE        I,
         CUSTOMERS     C,
         INVENTORY     I2
   WHERE I2.orddate < mdy(09,01,00) AND
         P.prodnum   = I.prodnum AND
         I.custnum   = C.custnum AND
         I.prodnum   = I2.prodnum;

NOTE: SQL execution methods chosen are:
      sqxslct
      sqxsumm
      sqxjhsh
      sqxjhsh
      sqxjhsh
      sqxsric( WORK.CUSTOMERS(alias = C) )
      sqxsric( WORK.INVOICE(alias = I) )
      sqxsric( WORK.PRODUCTS(alias = P) )
      sqxsric( WORK.INVENTORY(alias = I2) )

```

The `_TREE` option, when specified, provides a graphical representation of the way the query optimizer handled the execution of a query. Essentially, the query execution plan shows how a query was interpreted and executed by the optimizer based on the statements, clauses, options, keywords, and other coding constructs used.

When a query is submitted for execution, the optimizer determines the most optimal way to execute the query, and consequently constructs an execution plan. Typically, the decision about what execution plan a cost-based optimizer takes is influenced by the estimated CPU and IO processing costs, and the speed in which the query is expected to execute. The result is a query execution plan depicting a tree structure of labels, nodes, input sources, join parameters, where-clause conditions, and other descriptive information, as shown below.

```

Tree as planned.

      /-OBJ---| /-SYM-A- (Products_Ordered_Before_09012000:1 flag=0039)
      /-AGGR--|   /-OBJ---| /-SYM-V- (I2.invenqty:2 flag=0001)
      --JOIN---|     /-OBJ---| /-SYM-V- (P.prodnum:1 flag=0001)
                  |--SYM-V- (I.prodnum:6 flag=0001)
                  |--SYM-V- (I.custnum:3 flag=0001)
                  \-SYM-V- (C.custnum:1 flag=0001)
      /-JOIN---|     /-OBJ---| /-SYM-V- (I.prodnum:6 flag=0001)
                  |--SYM-V- (I.custnum:3 flag=0001)
                  \-SYM-V- (C.custnum:1 flag=0001)
      /-JOIN---|     /-OBJ---| /-SYM-V- (C.custnum:1 flag=0001)
                  |--SRC---| \-TABL[WORK].CUSTOMERS opt=' '
      --FROM---|           /-OBJ---| /-SYM-V- (I.prodnum:6 flag=0001)
                  |--SRC---| \-SYM-V- (I.custnum:3 flag=0001)
                  \-TABL[WORK].INVOICE opt=' '
      --empty-|           /-OBJ---| /-SYM-V- (C.custnum:1)
                  \-CEO---| \-SYM-V- (I.custnum:3)
      --FROM---|           /-SYM-V- (P.prodnum:1 flag=0001)
                  \-SRC---| \-TABL[WORK].PRODUCTS opt=' '
      --empty-|           /-SYM-V- (I.prodnum:6)
                  \-CEQ---| \-SYM-V- (P.prodnum:1)
      --FROM---|           /-SYM-V- (I2.invenqty:2 flag=0001)
                  /-OBJ---| \-SYM-V- (I2.prodnum:1 flag=0001)
      \-SRC---|           /-TABL[WORK].INVENTORY opt=' '
                  \-NAME--(orddate:3)
                  \-CLT---| \-LITN(14854) DATE.
      --empty-|           /-SYM-V- (I.prodnum:6)
                  \-CEQ---| \-SYM-V- (I2.prodnum:1)
      --empty-
      --empty-
      --empty-
      --empty-          /-SYM-A- (Products_Ordered_Before_09012000:1 flag=0039)
      /-ASGN--|           \-SYM-G- (#TEMG001:2 stat=8,0 from invenqty(0,0))
      --OBJE--|
      --empty-          /-SYM-G- (#TEMG001:2 stat=8,0 from invenqty(0,0))
      --TLST--|           /-SYM-S- (invenqty:3 ss=0220x)
      \-SLST---|
      --SSEL--|

```

INOBS= Option

The INOBS= option reduces the amount of query execution time by restricting the number of rows that PROC SQL processes. This option is most often used for troubleshooting or debugging purposes where a small number of rows are needed as opposed to all the rows in the table source. Controlling the number of rows processed on input with the INOBS= option is similar to specifying the SAS System option to OBS=. The following example illustrates the INOBS= option being limited to the first 10 rows in the PRODUCTS table. A warning message is also displayed in the SAS log that indicates that the number of records read was restricted to 10.

SQL Code

```
PROC SQL INOBS=10;
  SELECT *
    FROM PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL INOBS=10;
  SELECT *
    FROM PRODUCTS;
WARNING: Only 10 records were read from WORK.PRODUCTS due to
INOBS= option.
QUIT;
```

In the next example, the INOBS= option is set to 5 in a Cartesian product join (the absence of a WHERE clause). A two-way join with the INOBS=5 specified without a WHERE clause limits the number of rows from each table to 5, which produces a maximum of 25 rows.

SQL Code

```
PROC SQL INOBS=5;
  SELECT prodname, prodcost, manufacturers.manunum, manuname
    FROM PRODUCTS, MANUFACTURERS;
QUIT;
```

SAS Log Results

```
PROC SQL INOBS=5;
  SELECT prodname, prodcost, manufacturers.manunum, manuname
    FROM PRODUCTS, MANUFACTURERS;
NOTE: The execution of this query involves performing one or more
Cartesian product joins that cannot be optimized.
WARNING: Only 5 records were read from WORK.MANUFACTURERS due to
INOBS= option.
WARNING: Only 5 records were read from WORK.PRODUCTS due to
INOBS= option.
QUIT;
```

LOOPS= Option

The LOOPS= option reduces the amount of query execution time by restricting how many times processing occurs through a query's inner loop. As with the INOBS= and OUTOBS= options, the LOOPS= option is used for troubleshooting or debugging to prevent the consumption of excess computer resources or the creation of large internal tables as with the processing of multi-table joins. The following example shows the LOOPS= option being restricted to eight inner loops through the rows in the PRODUCTS table.

SQL Code

```
PROC SQL LOOPS=8;
  SELECT *
    FROM PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL LOOPS=8;
  SELECT *
    FROM PRODUCTS;
WARNING: PROC SQL statement interrupted by LOOPS=8 option.
QUIT;
```

Results

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1110	Dream Machine	111	Workstation	\$3,200.00
1200	Business Machine	120	Workstation	\$3,300.00
1700	Travel Laptop	170	Laptop	\$3,400.00

The next example shows what happens when the LOOPS= option is applied in a three-way join by restricting the number of processed inner loops to 50 to prevent the creation of a large and inefficient internal table. To determine an adequate value to assign to the LOOPS= option, you can specify an &SQLLOOPS macro variable in a %PUT statement. To learn more about this macro variable, see the “Macro Variables” section later in this chapter.

SQL Code

```
PROC SQL LOOPS=50;
  SELECT P.prodname, P.prodcost,
         M.manuname,
         I.invqty
    FROM PRODUCTS  P,
         MANUFACTURERS  M,
         INVOICE  I
   WHERE P.manunum = M.manunum AND
         P.prodnum = I.prodnum AND
         M.manunum = 500;
QUIT;
```

Results

Product Name	Product Cost	Manufacturer Name	Invoice Quantity - Units Sold
Spreadsheet Software	\$299.00	KPL Enterprises	5
Database Software	\$399.00	KPL Enterprises	2

RESET Statement

The RESET statement is used to add, drop, or change one or more PROC SQL options without the need to restart the procedure. Once an option is specified, it stays in effect until it is changed or reset. Being able to change options with the RESET statement is a handy

debugging technique. The following example illustrates turning off the FEEDBACK option by resetting it to NOFEEDBACK. By turning off this option, you prevent the expansion of a SELECT * (wildcard) statement into a list of columns that it represents.

SQL Code

```
PROC SQL FEEDBACK;
  SELECT *
    FROM PRODUCTS;
RESET NOFEEDBACK;
  SELECT *
    FROM PRODUCTS
      WHERE PRODTYPE='Software';
QUIT;
```

SAS Log Results

```
SELECT *
  FROM PRODUCTS;
NOTE: Statement transforms to:
select PRODUCTS.prodnum, PRODUCTS.prodname, PRODUCTS.manunum,
      PRODUCTS.prodtype, PRODUCTS.prodcost
  from PRODUCTS;
RESET NOFEEDBACK;
  SELECT *
    FROM PRODUCTS
      WHERE PRODTYPE='Software';
```

Multiple options can be reset in a single RESET statement. Options in the PROC SQL and RESET statements can be specified in any order. The next example shows how, in a single RESET statement, double-spaced output is changed to single-spaced output with the NODOUBLE option, row numbers are suppressed with the NONNUMBER option, and output rows are changed to the maximum number of rows with the OUTOBS= option.

SQL Code

```
PROC SQL DOUBLE NUMBER OUTOBS=1;
  SELECT *
    FROM PRODUCTS
      WHERE PRODTYPE='Software';
RESET NODOUBLE NONNUMBER OUTOBS=MAX;
  SELECT *
    FROM PRODUCTS
      WHERE PRODTYPE='Software';
QUIT;
```

SAS Log Results

```
PROC SQL DOUBLE NUMBER OUTOBS=1;
  SELECT *
    FROM PRODUCTS
      WHERE PRODTYPE='Software';
WARNING: Statement terminated early due to OUTOBS=1 option.
RESET NODOUBLE NONNUMBER OUTOBS=MAX;
  SELECT *
```

```

    FROM PRODUCTS
      WHERE PRODTYPE='Software';
    QUIT;
  
```

Output Results

Row	Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
1	5001	Spreadsheet Software	500	Software	\$299.00

Product Number	Product Name	Manufacturer Number	Product Type	Product Cost
5001	Spreadsheet Software	500	Software	\$299.00
5002	Database Software	500	Software	\$399.00
5003	Wordprocessor Software	500	Software	\$299.00
5004	Graphics Software	500	Software	\$299.00

The next example shows an UPDATE query that reverses any updates that have been performed up to the point of an error using the UNDO_POLICY=REQUIRED (default value) option. (Note: Because this is the default value for this option, it could have been omitted.) A RESET statement of UNDO_POLICY=NONE is issued before the second update query to change the way updates are handled. The NONE option keeps any updates that have been made regardless of whether an error is detected. A warning message is displayed alerting you to the change in the way updates are handled.

SQL Code

```

PROC SQL UNDO_POLICY=REQUIRED;
  UPDATE PRODUCTS
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2)
      WHERE UPCASE(PRODTYPE) = 'LAPTOP';
  RESET UNDO_POLICY=NONE;
  UPDATE PRODUCTS
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2)
      WHERE UPCASE(PRODTYPE) = 'LAPTOP';
QUIT;
  
```

SAS Log Results

```

PROC SQL UNDO_POLICY=REQUIRED;
  UPDATE PRODUCTS
    SET PRODCOST = PRODCOST - (PRODCOST * 0.2)
      WHERE UPCASE(PRODTYPE) = 'LAPTOP';
  NOTE: 1 row was updated in WORK.PRODUCTS.
  RESET UNDO_POLICY=NONE;
  UPDATE PRODUCTS
  
```

```

SET PRODCOST = PRODCOST - (PRODCOST * 0.2)
      WHERE UPCASE(PRODTYPE) = 'LAPTOP';
WARNING: The SQL option UNDO_POLICY=REQUIRED is not in effect.
If an
error is detected when processing this UPDATE statement, that
error
will not cause the entire statement to fail.
NOTE: 1 row was updated in WORK.PRODUCTS.
QUIT;

```

EXEC/NOEXEC Option

The NOEXEC option checks all non-query statements such as CREATE TABLE or ALTER TABLE within the SQL procedure for syntax-related errors and displays any identified errors in the SAS log. The NOEXEC option is similar to the VALIDATE statement in that it checks for syntax correctness without the execution of any input data. The only difference between the NOEXEC option and the VALIDATE statement is in the way they are specified. As was presented earlier, the VALIDATE statement is specified before each SELECT statement; the NOEXEC / EXEC option can be specified as a PROC SQL statement option or in a RESET statement.

To illustrate the troubleshooting and debugging process, the following non-query SQL code will be shown using the NOEXEC and EXEC options. The next example illustrates that the non-query CREATE TABLE statement is not to be executed after its syntax is checked for correctness. As a result of the NOEXEC option being specified, any syntax-related errors are automatically displayed in the SAS log.

SQL Code

```

PROC SQL NOEXEC;
CREATE TABLE SOFTWARE_PRODUCTS
SELECT *
      FROM PRODUCTS
      WHERE PRODTYPE = 'Software';

```

As illustrated on the SAS log, a syntax error is found and the resulting error is displayed.

SAS Log Results

```

PROC SQL NOEXEC;
CREATE TABLE SOFTWARE_PRODUCTS
SELECT *
-----
73
ERROR 73-322: Expecting an AS.
      FROM PRODUCTS
      WHERE PRODTYPE = 'Software';

```

In the next example, with the NOEXEC option still in effect, the AS keyword is specified with the CREATE TABLE statement, which corrects the syntax error that is displayed in the previous example.

SQL Code

```
CREATE TABLE SOFTWARE_PRODUCTS AS
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE = 'Software';
```

As illustrated in the SAS log, no syntax errors were found, but because the NOEXEC option is still active the CREATE TABLE statement was not executed.

SAS Log Results

```
PROC SQL NOEXEC;
  CREATE TABLE SOFTWARE_PRODUCTS AS
    SELECT *
      FROM PRODUCTS
     WHERE PRODTYPE = 'Software';
NOTE: Statement not executed due to NOEXEC option.
```

The next example illustrates changing the NOEXEC option to EXEC with a RESET statement, which results in error-free non-query code to execute.

SQL Code

```
RESET EXEC;
CREATE TABLE SOFTWARE_PRODUCTS AS
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE = 'Software';
QUIT;
```

As illustrated in the SAS log, no syntax errors were found. And because the RESET statement was specified to change the NOEXEC option to EXEC, the CREATE TABLE statement executed.

SAS Log Results

```
RESET EXEC;
CREATE TABLE SOFTWARE_PRODUCTS AS
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE = 'Software';
NOTE: Table WORK.SOFTWARE_PRODUCTS created, with 4 rows and 5
columns.
```

ERRORSTOP/NOERRORSTOP Option

The ERRORSTOP option instructs SQL to stop executing statements and to continue to check syntax in noninteractive or batch sessions when an error is encountered. Conversely, the NOERRORSTOP option tells SQL to execute statements when an error is encountered and to continue checking syntax. The ERRORSTOP / NOERRORSTOP option can be specified as a PROC SQL statement option or in a RESET statement. The next example illustrates a CREATE TABLE statement being used incorrectly, and shows the SAS log results that are generated when the ERRORSTOP option is specified.

SQL Code

```
PROC SQL ERRORSTOP;
  CREATE TABLE SOFTWARE_PRODUCTS
    SELECT *
      FROM PRODUCTS
        WHERE PRODTYPE = 'Software';
QUIT;
```

As illustrated in the SAS log, the ERRORSTOP option automatically stops executing the SQL statements and displays a SAS log error message when a syntax error is encountered.

SAS Log Results

```
PROC SQL ERRORSTOP;
  CREATE TABLE SOFTWARE_PRODUCTS
    SELECT *
    -----
    73
ERROR 73-322: Expecting an AS.
  FROM PRODUCTS
    WHERE PRODTYPE = 'Software';
```

OUTOBS= Option

The OUTOBS= option reduces the amount of query execution time by restricting the number of rows that PROC SQL sends as output to a designated output source. As with the INOBS= option, the OUTOBS= option is most often used for troubleshooting or debugging purposes where a small number of rows are needed in an output table. Controlling the number of rows that are sent as output with the OUTOBS= option is similar to setting the SAS System option OBS= (or the data set option OBS=). The following example creates an output table with five rows as designated by the OUTOBS= option. A SAS log warning message shows that the new table contains five rows.

SQL Code

```
PROC SQL OUTOBS=5;
  CREATE TABLE PRODUCTS_SAMPLE AS
    SELECT *
      FROM PRODUCTS;
QUIT;
```

SAS Log Results

```
PROC SQL OUTOBS=5;
  CREATE TABLE PRODUCTS_SAMPLE AS
    SELECT *
      FROM PRODUCTS;
WARNING: Statement terminated early due to OUTOBS=5 option.
NOTE: Table WORK.PRODUCTS_SAMPLE created, with 5 rows and 5
columns.
QUIT;
```

PROMPT Option

The PROMPT option is issued during interactive sessions to prompt users to continue or stop processing when the limits of an INOBS=, LOOPS=, and/or OUTOBS= option are reached. If the PROMPT option is specified along with one or more of these options, a dialog box appears that indicates that the limits of the specified option have been reached, and asks whether to stop or continue processing. This prompting feature is a useful process for stepping through a running application.

The following example shows the PROMPT option being issued to initiate a dialogue between the SQL procedure session and the user. The PROMPT option specifies that the INOBS= option limit input processing to the first five rows in the PRODUCTS table. A dialog box automatically appears after five rows are read asking whether processing is to stop or continue. If processing is continued, another five rows are processed and, if additional rows are available for processing, another prompt dialog box appears. This dialogue process continues until all rows are processed or until the user halts processing.

SQL Code

```
PROC SQL PROMPT INOBS=5;
  SELECT *
    FROM PRODUCTS;
  QUIT;
```

SAS Log Results

```
PROC SQL PROMPT INOBS=5;
  SELECT *
    FROM PRODUCTS;
WARNING: Only 5 records were read from WORK.PRODUCTS due to
INOBS= option.
  QUIT;
```

Undocumented PROC SQL Options

This section lists several undocumented PROC SQL options. Although undocumented options can be freely explored and used, you should carefully consider the ramifications before using them. Because of unannounced changes, possible removal, or nonsupport in future releases, you should exercise care when using them throughout SQL procedure applications. However, undocumented options provide a wealth of opportunities for identifying and resolving coding problems. Table 11.5 presents several of these undocumented SQL procedure options for troubleshooting and debugging purposes.

Table 11.5: Undocumented PROC SQL Options

Option	Description
_AGGR	Displays a tree structure in the SAS log with a before-and-after summary.
_ASGN	Displays a tree structure in the SAS log that consists of resolved before-and-after names.
_DFR	Displays a before-and-after dataflow and subcall resolution in a tagged tree structure in the SAS log.

Option	Description
_PJD	Displays various table attributes including the number of observations (rows), the logical record length (lrecl), the number of restricted rows, and the size of the table in bytes.
_RSLV	Displays a tree structure in the SAS log that consists of before-and-after early semantic checks.
_SUBQ	Displays subquery transformations as a tree structure in the SAS log.
_UTIL	Displays a breakdown in the SAS log of each step defined in the procedure including each row's buffer length, each column by name, and the column position with each row (or offset).

Macro Variables

To assist with the process of troubleshooting and debugging problematic coding constructs, PROC SQL assigns values to three automatic macro variables after the execution of each statement. The contents of these three macro variables can be used to test the validity of SQL procedure code as well as to evaluate whether processing should continue.

SQLOBS Macro Variable

The SQLOBS macro variable displays the number of rows that are processed by an SQL procedure statement. To display the contents of the SQLOBS macro variable in the SAS log, specify a %PUT macro statement. The following example retrieves the software products from the PRODUCTS table and displays SAS output with a SELECT statement. The %PUT statement displays the number of rows processed and sent to SAS output as four rows.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE='Software';
%PUT SQLOBS = &SQLOBS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE='Software';
%PUT SQLOBS = &SQLOBS;
SQLOBS = 4
QUIT;
```

The next example shows two new products inserted in the PRODUCTS table with an INSERT INTO statement. The %PUT statement displays the number of rows added to the PRODUCTS table as two rows.

SQL Code

```
PROC SQL;
  INSERT INTO PRODUCTS
    (PRODNUM, PRODNAME, PRODTYPE, PRODCOST)
    VALUES(6002,'Security Software','Software',375.00)
    VALUES(1701,'Travel Laptop SE', 'Laptop', 4200.00);
  %PUT SQLOBS = &SQLOBS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  INSERT INTO PRODUCTS
    (PRODNUM, PRODNAME, PRODTYPE, PRODCOST)
    VALUES(6002,'Security Software','Software',375.00)
    VALUES(1701,'Travel Laptop SE', 'Laptop', 4200.00);
NOTE: 2 rows were inserted into WORK.PRODUCTS.
  %PUT SQLOBS = &SQLOBS;
  SQLOBS = 2
QUIT;
```

SQLOOPS Macro Variable

The SQLOOPS macro variable displays the number of times that the inner loop is processed by the SQL procedure. To display the contents of the SQLOOPS macro variable in the SAS log, specify a %PUT macro statement. The following example retrieves the software products from the PRODUCTS table and displays SAS output with a SELECT statement. The %PUT statement displays the number of times the inner loop is processed as 15 times even though there are only 10 product rows. As a query becomes more complex, the number of times the inner loop of the SQL procedure processes also increases proportionally.

SQL Code

```
PROC SQL;
  SELECT *
  FROM PRODUCTS
  WHERE PRODTYPE='Software';
  %PUT SQLOOPS = &SQLOOPS;
QUIT;
```

SAS Log Results

```
PROC SQL;
  SELECT *
  FROM PRODUCTS
  WHERE PRODTYPE='Software';
  %PUT SQLOOPS = &SQLOOPS;
  SQLOOPS = 15
QUIT;
```

SQLRC Macro Variable

The SQLRC macro variable displays a status value that indicates whether the PROC SQL statement was successful or not. A %PUT macro statement is specified to display the contents of the SQLRC macro variable. The following example retrieves the software products from the PRODUCTS table and displays SAS output with a SELECT statement. The %PUT statement displays a return code of zero, which indicates that the SELECT statement was successful.

SQL Code

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE='Software';
%PUT SQLRC = &SQLRC;
QUIT;
```

SAS Log Results

```
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE PRODTYPE='Software';
%PUT SQLRC = &SQLRC;
SQLRC = 0
QUIT;
```

Troubleshooting and Debugging Examples

This section shows a number of errors that I have personally experienced while working on SQL procedure problems. Although not representative of all of the possible errors that might occur, it does illustrate a set of common problems along with a technical approach for correcting each problem.

ERROR 78-322: Expecting a ','

Problem Description

Syntax errors messages can, at times, provide confusing information about the specific problem at hand. A case in point is the error, **78-322: Expecting a ','**. In the example below, it initially appears that a comma is missing between two column names in the SELECT statement. On closer review, the actual problem points to a violation of the column's naming conventions that was caused by specifying an invalid character in the assigned column alias in the AS keyword.

Code and Error

```
PROC SQL;
  SELECT CUSTNUM, ITEM, UNITS * UNITCOST AS Total-Cost
```

```

ERROR 78-322: Expecting a ',',.
  FROM PURCHASES
    ORDER BY TOTAL;
QUIT;

```

Corrective Action

Correct the problem that is associated with the assigned column-alias name by adhering to valid SAS naming conventions. For example, replace the hyphen “-” in Total-Cost with an underscore, as in Total_Cost.

ERROR 202-322: The option or parameter is not recognized and will be ignored

Problem Description

Sometimes problems occur because of unfamiliarity with the SQL procedure language syntax. In the syntax error below, an unrecognized option or parameter is encountered, which results in the procedure stopping before any processing occurs.

Code and Error

```

PROC SQL;
  SELECT prodtype,
         MIN(prodcost) AS Cheapest
           Format=dollar9.2 Label='Least Expensive'
    FROM PRODUCTS
   ORDER BY cheapest
      GROUP BY prodtype;
  -----
  22      202

ERROR 22-322: Syntax error, expecting one of the following: ;, !, !!,
&, (, *, **, +, ',', -, '.', /, <, <=, >, =, >=, ?, AND, ASC,
ASCENDING, BETWEEN, CONTAINS, DESC, DESCENDING, EQ, EQT, GE, GET, GT,
GTT, IN, IS, LE, LET, LIKE, LT, LTT, NE, NET, NOT, NOTIN, OR, ^, ^=,
!, ||, ~, =~.

ERROR 202-322: The option or parameter is not recognized and will be ignored.

QUIT;

```

Corrective Action

This problem is the result of the SELECT statement clauses not being specified in the correct order. It can be corrected by specifying the SELECT statement's GROUP BY clause before the ORDER BY clause.

ERROR Ambiguous reference, column

Problem Description

In the next example, the syntax error points to a problem where a column name that is specified in a SELECT statement appears in more than one table resulting in a column ambiguity. This problem not only creates confusion for the SQL processor, but also prevents the query from executing.

Code and Error

```
PROC SQL;
  SELECT prodname, prodcost,
         manunum, manuname
    FROM PRODUCTS AS P, MANUFACTURERS AS M
   WHERE P.manunum = M.manunum;
ERROR: Ambiguous reference, column manunum is in more than one table.
QUIT;
```

Corrective Action

To remove any and all column ambiguities, you should reference each column that appears in two or more tables with its respective table name in a SELECT statement and its clauses. For example, to reference the MANUNUM column in the MANUFACTURERS table and remove all ambiguities, you would specify the column in the SELECT statement as MANUFACTURERS.MANUNUM.

ERROR 200-322: The symbol is not recognized and will be ignored

Problem Description

In the next example, the syntax error points to the left parenthesis at the end of the second SELECT statement's WHERE clause as being invalid. The key to finding the actual problem is to work backward, line-by-line, from the point where the error is marked. Using this approach, you will notice that the second SELECT statement is a subquery (or inner query) and does not conform to valid syntax rules. As with other syntax errors, the query as well as the subquery does not execute.

Code and Error

```
PROC SQL;
  SELECT *
    FROM INVOICE
   WHERE manunum IN
        (SELECT manunum
          FROM MANUFACTURERS
         WHERE UPCASE(manucity) LIKE 'SAN DIEGO%');

22
-
200
ERROR 22-322: Syntax error, expecting one of the following: ;, !, !!,
&, *, **, +, -, /, AND, ESCAPE, EXCEPT, GROUP, HAVING, INTERSECT, OR,
ORDER, OUTER, UNION, |, ||.
```

```
ERROR 200-322: The symbol is not recognized and will be ignored.
QUIT;
```

Corrective Action

A subquery must conform to valid syntax rules. To correct this problem, a right parenthesis must be added at the beginning of the second SELECT statement immediately after the IN clause.

ERROR 22-322: expecting one of the following: a name, (, '.', AS, ON

Problem Description

In the next example, the syntax error identifies a WHERE clause that is being used in a left outer join. Although the SQL procedure permits an optional WHERE clause to be specified in the outer join syntax, an ON clause must be specified as well.

Code and Error

```
PROC SQL;
  SELECT prodname, prodtype,
         products.manunum, invenqty
    FROM PRODUCTS LEFT JOIN INVENTORY
      WHERE products.manunum =
      -----
      22
      76
ERROR 22-322: Syntax error, expecting one of the following: a name, (,
'.', AS, ON.
ERROR 76-322: Syntax error, statement will be ignored.
               inventory.manunum;
QUIT;
```

Corrective Action

To correct this problem and to conform to valid outer join syntax requirements, specify an ON clause before an optional WHERE clause that is used to subset joined results.

ERROR 180-322: Statement is not valid or it is used out of proper order

Problem Description

In the next example, the syntax error identifies the UNION statement as being invalid or used out of proper order. On further inspection, it is clear that one of two problems exists. The SQL procedure code consists of two separate queries with an invalid UNION operator specified, or a misplaced semicolon appears at the end of the first SELECT query.

Code and Error

```
PROC SQL;
  SELECT *
    FROM products
      WHERE prodcost < 300.00;
UNION
-----
180
ERROR 180-322: Statement is not valid or it is used out of proper
order.
  SELECT *
    FROM products
      WHERE prodtype = 'Workstation';
QUIT;
```

Corrective Action

To correct this problem and conform to valid rules of syntax for a UNION operation, remove the semicolon after the first SELECT query.

ERROR 73-322: Expecting an AS

Problem Description

In the next example, the syntax error identifies a missing AS keyword in the CREATE VIEW statement and highlights the view's SELECT statement. If the AS keyword is not specified, the CREATE VIEW step is not executed and the view is not created.

Code and Error

```
PROC SQL;
  CREATE VIEW WORKSTATION_PRODUCTS_VIEW
    SELECT PRODNUM, PRODNAME, PRODTYPE, PRODCOST
  -----
73
ERROR 73-322: Expecting an AS.
  FROM PRODUCTS
    WHERE UPCASE (PRODTYPE) ="WORKSTATION";
QUIT;
```

Corrective Action

To correct the problem, add the AS keyword in the CREATE VIEW statement to follow valid syntax rules and define the view's query.

Summary

1. The objective of the debugging process is to identify, classify, fix, and verify a problem in a PROC SQL step, program, or application as quickly and easily as possible (see “The Debugging Process” section).
2. Usage errors can cause the SAS System to stop processing, produce warnings, or produce unexpected results (see the “Types of Problems” section).

3. PROC SQL provides numerous troubleshooting and debugging capabilities for the practitioner (see the “Troubleshooting and Debugging Techniques” section).
4. PROC SQL options provide effective troubleshooting and debugging techniques for resolving coding problems (see the “Documented PROC SQL Options and Statements” section).
5. Several useful, but nonproduction, PROC SQL options are available for troubleshooting and debugging (see the “Undocumented PROC SQL Options” section).

Chapter 12: Tuning for Performance and Efficiency

Introduction	391
Understanding Performance Tuning	391
Sorting and Performance	392
User-Specified Sorting (SORTPGM= System Options).....	392
Automatic Sorting	393
Grouping and Performance	393
Splitting Tables.....	393
Indexes and Performance	394
Reviewing CONTENTS Output and System Messages	395
Optimizing WHERE Clause Processing with Indexes	398
Constructing Efficient Logic Conditions.....	398
Avoiding UNIONs	401
Summary	404

Introduction

A book on PROC SQL would not be complete without some discussion of query optimization and performance. Enabling a query to run efficiently involves writing code that can take advantage of the PROC SQL query optimizer. Because PROC SQL is designed to handle a variety of processes while accommodating small to large database environments, this chapter presents a number of query tuning strategies, techniques, and options to help you to write more efficient PROC SQL code. In this chapter, you will find tips and suggestions to help identify areas where a query's inefficiencies might exist and to conduct the tuning process to achieve the best performance possible.

Understanding Performance Tuning

Performance tuning is the process of improving the way a program operates. It involves taking a program and seeing what can be done to improve performance in an intelligent, controlled manner. As you might imagine, a tuned program is one that gets the most from the existing hardware and software environment.

Performance tuning involves measuring, evaluating, and modifying a program until it uses the minimum amount of computer resources to complete its execution. The biggest problem with the tuning process is that it is sometimes difficult to determine the amount of computer resources that a program uses. Complicating matters further, adequate and complete information about resource utilization is often unavailable. In fact, no simple formula exists to determine how efficiently a program runs. Often the only way to assess whether a program

is running efficiently is to evaluate its performance under varying conditions, such as during interactive use or during shortages of specific resources including memory and storage.

Performance issues might be difficult to identify. It is possible to have a program that operates without any apparent problem, but does not perform as efficiently as it could. In fact, a program might perform well in one environment and poorly in another. Take, for example, an organization that has a shortage of Direct Access Storage Device (DASD). A program that uses excessive amounts of this resource might be deemed a poor performer under these circumstances. But if the same program were run in an environment that had adequate levels of DASD, it might not be suspected or tagged as a poor performer. This distinction demonstrates the subjectivity that is frequently used to determine how a program performs and how it is linked to the specific needs (related to resource issues) that an organization has at any point in time.

Sorting and Performance

Sorting data in the SQL procedure, as in other parts of SAS, involves CPU and memory-intensive operations. When the table size is large, and CPU and/or memory resources are in short supply or simply not available, the number of sorts in programs must be minimized. You can minimize problems by understanding your query's requirements and remembering a few simple guidelines.

Performance bottlenecks can occur if sorts are performed on disk as opposed to in memory because processing on disk is typically slower than processing in memory. The most logical and efficient place to perform a sort is in memory. If the sort requires more space than can fit in available memory, the sort must be performed on disk. The objective is to determine how much space a sort will require as well as where the sort will be performed before the sort is executed.

The performance of a sort is most influenced by the number of rows selected in a query. To reduce the row count, select specific parts of a table for processing rather than selecting the entire table. Another performance consideration is to reduce the number of columns that are specified in an ORDER BY clause. The technique of concatenating two or more columns as opposed to specifying each column individually can be used. Finally, sort performance can be influenced by the size of the columns that are specified in an ORDER BY clause. To reduce a column's length, built-in functions such as SUBSTR and TRIM can be used.

User-Specified Sorting (SORTPGM= System Options)

You can control what sort utility SAS uses when performing sorts. By specifying the SORTPGM= system option, you can direct SAS to use the best possible sort utility for the environment in question. The SORTPGM= system options are displayed in Table 12.1.

Table 12.1: SORTPGM= System Options

Sort Option	Purpose
BEST	The BEST option uses the sort utility that is best suited to the data.
HOST	The HOST option tells SAS to use the host sort utility that is available on your host computer. This option might be the most efficient for large tables that contain many rows of data.
SAS	The SAS option tells SAS to use the sort utility that is supplied with SAS.

The next example illustrates how to use the SORTPGM= option to select the sort utility that is most suited to the data. Both options use the name that is specified in the SORTNAME= option.

```
OPTIONS SORTPGM=BEST;
<or>
OPTIONS SORTPGM=HOST;
```

Automatic Sorting

Using the SELECT DISTINCT clause invokes an internal sort to remove duplicate rows. The single exception is when an index exists. If an index exists, then the index is used to eliminate the duplicate rows.

The results of a grouped query are automatically sorted using the grouping columns. When the SELECT clause contains only the columns that are listed in the GROUP BY clause along with any summary functions, then the duplicates in each group based on the grouping columns are removed as soon as any defined summary functions are performed. If additional columns then appear in the SELECT clause, the rows are not collapsed and therefore duplicates are not removed.

Grouping and Performance

As with the ORDER BY clause, processes such as the GROUP BY clause can also impact the speed of a query. Because the GROUP BY clause can trigger sort processing, much of what was discussed with the ORDER BY clause also applies here. In addition, it is recommended that you avoid grouping redundant columns by keeping the number of grouping columns to a minimum.

Splitting Tables

Splitting tables involves moving some of the rows from one table to another table. Data is split for the purpose of separating some predetermined range of data, such as historical data

from current data, so that query performance is improved. This reduces the burden imposed on queries that only access current data. The next example shows the current year's data being copied and then removed from a table that contains five years of data.

SQL Code

```
PROC SQL;
  CREATE TABLE INVENTORY_CURRENT AS
    SELECT *
      FROM INVENTORY
     WHERE YEAR(ORDDATE) = YEAR(TODAY());
  DELETE FROM INVENTORY
  WHERE YEAR(ORDDATE) = YEAR(TODAY());
QUIT;
```

Indexes and Performance

Indexes can be used to allow rapid access to table rows. Rather than physically sorting a table (as performed by the ORDER BY clause or PROC SORT), an index is designed to set up a logical arrangement for the data without the need to physically sort it. This has the advantage of reducing CPU, I/O, and memory requirements. It also reduces data access time when using WHERE clause processing (which is discussed in the “Optimizing WHERE Clause Processing with Indexes” section).

Indexes are useful, but they do have drawbacks. As data in a table is inserted, modified, or deleted, an index must be updated to address the changes. This automatic feature requires additional CPU resources to process any changes to a table. Also, as a separate structure in its own right, an index can consume considerable storage space. As a consequence, care should be exercised not to create too many indexes but to assign indexes to the most discriminating variables in a table. Here are a few suggestions for creating indexes:

- Sort data in ascending order on the key column prior to creating the index.
- Sort data by the key variable first to achieve the greatest performance improvement.
- Sort data in ascending order by the key variable before it is appended to the table.
- Create simple indexes, when possible, to be used by most queries.
- Avoid creating one single index for all queries.
- Assign indexes to the most discriminating of variables (see Table 6.1, Rules of Cardinality, in Chapter 6).
- Select columns that are frequently the subject of summary functions (COUNT, SUM, AVG, MIN, MAX, etc.).
- Only create indexes that are actually needed.
- Avoid taxing CPU resources that are associated with index maintenance (maintaining an index during inserts, modifications, and deletions) by selecting columns that do not change frequently.
- On some operating systems, indexes are stored as a separate file on disk, which uses additional memory and disk space to store the structure.
- To avoid excessive and unnecessary I/O operations, prior to creating an index, sort data in ascending order by the most discriminating key column.

- Attempt to define composite indexes by using the most discriminating of the variables as your first variable in the index.
- Select columns that do not have numerous null values because this results in a large percentage of rows with the same value.

Note: Indexes should only be created on tables where query search time needs to be optimized. Any unnecessary indexes might force SAS to expend resources needlessly—updating and reorganizing after insert, update, and delete operations are performed. And even worse, the SQL optimizer might accidentally use an index when it should not.

Note: When creating an index is impractical or not feasible, consider storing the data in ascending (or descending) order depending on whether the query needs to access data from the first or second half of the table. This often results in a more efficient query because once a WHERE-clause expression is satisfied, the query stops processing.

Reviewing CONTENTS Output and System Messages

While no two organizations are alike, it is not surprising to find numerous causes for a program to run at less than peak efficiency. Performance is frequently affected by the specific needs of an organization or its lack of resources. SAS users need to learn as many techniques as possible to correct problems that are associated with poorly performing queries. Attention should be given to individual program functions, because poor query performance often points to one or more inefficient techniques being used.

Two methods can be used to better understand potential performance issues. The first approach uses PROC CONTENTS output to examine engine/host information and library data sets (tables). The output provides information to determine whether a table is large enough. (The page count in the following output shows the performance improvements offered by an index). The general rule that the SQL processor adheres to is that when a table is relatively small (usually fewer than three pages), there is no real advantage to using an index. In fact, using an index with a small table can actually degrade performance levels because in these situations sequential processing would be just as fast as using an index.

Results

The CONTENTS Procedure

Data Set Name	WORK.INVENTORY	Observations	7
Member Type	DATA	Variables	5
Engine	V9	Indexes	0
Created	Wed, Sep 04, 2013 08:43:16 AM	Observation Length	20
Last Modified	Wed, Sep 04, 2013 08:43:16 AM	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		

Engine/Host Dependent Information

Data Set Page Size	4096
Number of Data Set Pages	1
First Data Page	1
Max Obs per Page	202
Obs in First Data Page	7
Number of Data Set Repairs	0
Filename	C:\Users\KPL\AppData\Local\Temp\SAS Temporary Files\TD5444_KPL-PC_\inventory.sas7bdat
Release Created	9.0301M0
Host Created	W32_VSPRO

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format	Informat	Label
4	invencst	Num	6	DOLLAR10.2		Inventory Cost
2	invenqty	Num	3			Inventory Quantity
5	manunum	Num	3			Manufacturer Number
3	orddate	Num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
1	prodnum	Num	3			Product Number

The second approach uses PROC SQL to access the dictionary tables, TABLES and COLUMNS, to determine whether a table is large enough to take advantage of the performance improvements that are offered by an index. See the output below.

SQL Code

```
PROC SQL;
  SELECT MEMNAME, NPAGE
  FROM DICTIONARY.TABLES
  WHERE LIBNAME='WORK' AND
  MEMNAME='INVENTORY';
```

```

SELECT VARNUM, NAME, TYPE, LENGTH, FORMAT,
       INFORMAT, LABEL
  FROM DICTIONARY.COLUMNS
 WHERE LIBNAME='WORK' AND
       MEMNAME='INVENTORY';
QUIT;

```

Results

Member Name	Number of Pages
INVENTORY	1

Column Number in Table	Column Name	Column Type	Column Length	Column Format	Column Informat	Column Label
1	prodnum	num	3			Product Number
2	invenqty	num	3			Inventory Quantity
3	orddate	num	4	MMDDYY10.	MMDDYY10.	Date Inventory Last Ordered
4	invencst	num	6	DOLLAR10.2		Inventory Cost
5	manunum	num	3			Manufacturer Number

Table 12.2 compares sequential table access with indexed table access. Although performance gains are data dependent, the greatest gains are realized when an index is applied to a small subset of data in a WHERE clause.

Table 12.2: Sequential versus Indexed Table Access

Condition	Sequential	Index
Page count < 3 pages (from CONTENTS output)	Yes	No
Small table	Yes	No
Frequent updates to table	Yes	No
Large subset of data based on WHERE processing	Yes	No
Infrequent access of table	Yes	No
Limited memory and disk space	Yes	No
Small subset of data (1%–25% of population)	No	Yes

System messages are displayed to provide information that can help tune the indexes that are associated with any data sets. Setting the MSGLEVEL= system option to “I” allows SAS to display vital information (if available) that is related to the presence of one or more indexes for optimization of WHERE clause processing. With the MSGLEVEL= option turned on, the SAS log shows that the simple index INVENQTY was selected in the optimization of WHERE clause processing.

SAS Log

```

PROC SQL;
  CREATE INDEX INVENQTY ON INVENTORY;
NOTE: Simple index invenqty has been defined.
NOTE: PROCEDURE SQL used:
      real time          0.04 seconds

      SELECT *
        FROM INVENTORY
          WHERE invenqty < 3;
INFO: Index invenqty selected for WHERE clause
optimization.
      QUIT;
NOTE: PROCEDURE SQL used:
      real time          0.65 seconds

```

Optimizing WHERE Clause Processing with Indexes

To get the best possible performance from programs that contain SQL procedure code, an index and WHERE clause can be used together (see the list below). Using a WHERE clause restricts processing in a table to a subset of selected rows (for more information, see Chapter 2, “Working with Data in PROC SQL”). When an index exists, the SQL processor determines whether to take advantage of it during WHERE clause processing. Although the SQL optimizer determines whether using an index will ultimately benefit performance, when it does use an index the result can be an improvement in processing speeds.

- Comparison operators such as EQ (=), LT (<), GT (>), LE (<=), GE (>=), and NOT
- Comparison operators with the IN operator
- Comparison operators with the colon modifier (for example, NOT =：“Ab”)
- CONTAINS operator
- IS NULL or IS MISSING operator
- Pattern-matching operators such as LIKE and NOT LIKE

Constructing Efficient Logic Conditions

Constructing efficient logic conditions has a direct effect on processing costs. Because the SQL optimizer evaluates a series of AND expressions from left to right, a chain of AND conditions in a WHERE clause should be specified with the most restrictive expression first. Specified this way, fewer resources are expended by bypassing rows that do not satisfy the first conditional value in the WHERE clause. For example, the SQL query below might expend more resources and run slower because the first condition, “SOFTWARE”, occurs even when products cost more than \$99.00.

SQL Code (Less Efficient)

```

PROC SQL;
  SELECT *
    FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) = 'SOFTWARE' AND

```

```

PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE PRODCOST < 100.00 AND
        UPCASE(PRODTYPE) = 'SOFTWARE';
QUIT;

```

For this data, a more efficient way to produce the same results as the previous example, while reducing CPU resources, is to code the least likely condition first so that it appears as shown here.

SQL Code (More Efficient)

```

PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'SOFTWARE' AND
        PRODCOST < 100.00;
QUIT;

```

Another popular construct uses a series of OR condition equality tests or the IN predicate to select rows that match the multiple conditions. Programmers often specify these kinds of lists in order of magnitude or alphabetically to make the lists easier to read and/or maintain. A better and more efficient way is to construct a list of constants in the order of the most frequently occurring value to the least frequently occurring value.

Note: One way to determine the frequency of product type (PRODTYPE) values is to submit the following code:

```

PROC SQL;
  SELECT PRODTYPE, COUNT(PRODTYPE) AS Product_Frequency
    FROM PRODUCTS
   GROUP BY PRODTYPE;
QUIT;

```

Results

Product Type	Product_Frequency
Laptop	1
Phone	3
Software	4
Workstation	2

With the frequencies determined, the list of conditions can be specified in that order. In this way, fewer resources are spent locating frequently occurring values because it has to perform fewer steps to return a value of TRUE. It is also important to remove duplicate values in the constant list because the first identified duplicate value will automatically return a value of TRUE, which results in the second occurrence of the duplicate value being ignored.

The next query illustrates logic conditions that could require more processing resources because the first condition “LAPTOP” occurs less frequently than the value “SOFTWARE”. Consequently, the SQL processor needs to expend more resources to process the second condition in order to find a match of TRUE being returned.

SQL Code (Less Efficient)

```
PROC SQL STIMER;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) IN ('LAPTOP', 'SOFTWARE');
QUIT;
```

SAS Log Results

```
PROC SQL STIMER;
NOTE: SQL Statement used (Total process time):
      real time          0.00 seconds
      cpu time          0.01 seconds

SELECT *
  FROM PRODUCTS
 WHERE UPCASE(PRODTYPE) IN ('LAPTOP', 'SOFTWARE');

NOTE: SQL Statement used (Total process time):
      real time          0.08 seconds
      cpu time          0.04 seconds
```

A more efficient way to process the same data while generating the same results would be to specify the condition, “SOFTWARE”, first as follows:

SQL Code (More Efficient)

```
PROC SQL STIMER;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) IN ('SOFTWARE', 'LAPTOP');
QUIT;
```

SAS Log Results

```
PROC SQL STIMER;
SELECT *
  FROM PRODUCTS
 WHERE UPCASE(PRODTYPE) IN ('SOFTWARE', 'LAPTOP');

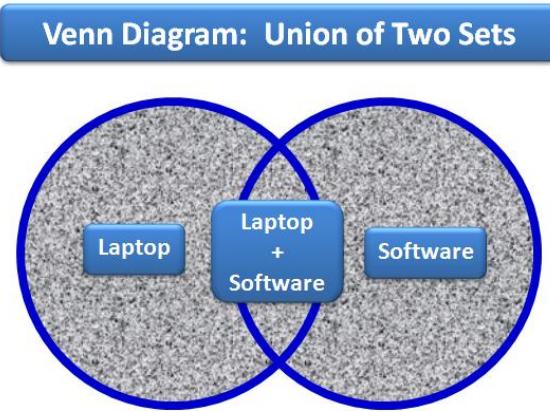
NOTE: SQL Statement used (Total process time):
      real time          0.02 seconds
      cpu time          0.03 seconds

QUIT;
```

Avoiding UNIONS

UNIONS are executed by creating two internal sets, then merge-sorting the results together. Duplicate rows are automatically eliminated from the final results. The Venn diagram for a UNION of two sets represents all distinct elements in the collection, as illustrated in Figure 12.1.

Figure 12.1: Venn Diagram



For example, the SQL procedure code that represents a UNION set first constructs the two result sets from each query, merges and sorts the two sets together, and then eliminates duplicate rows from the final results.

SQL Code (Less Efficient)

```
OPTIONS FULLSTIMER;
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'LAPTOP'

  UNION

  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'SOFTWARE';
QUIT;
```

SAS Log Results

```
OPTIONS FULLSTIMER;
PROC SQL;
  SELECT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'LAPTOP'

  UNION
```

```

SELECT *
  FROM PRODUCTS
 WHERE UPCASE(PRODTYPE) = 'SOFTWARE';
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.16 seconds
      user cpu time      0.00 seconds
      system cpu time   0.04 seconds
      memory            634.09k
      OS Memory         6768.00k
      Timestamp         06/03/2012 04:59:57 PM

```

To improve UNION performance, SQL procedure code can be converted to a single query using OR conditions in a WHERE clause. The next example illustrates the previous SQL procedure code being made more efficient by converting the UNION to a single query using an OR operator in a WHERE clause.

SQL Code (Efficient)

```

OPTIONS FULLSTIMER;
PROC SQL;
  SELECT DISTINCT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'SOFTWARE' OR
        UPCASE(PRODTYPE) = 'LAPTOP';
QUIT;

```

SAS Log Results

```

OPTIONS FULLSTIMER;
PROC SQL;
  SELECT DISTINCT *
    FROM PRODUCTS
   WHERE UPCASE(PRODTYPE) = 'SOFTWARE' OR
        UPCASE(PRODTYPE) = 'LAPTOP';
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.10 seconds
      user cpu time      0.04 seconds
      system cpu time   0.01 seconds
      memory            565.60k
      OS Memory         6768.00k
      Timestamp         06/03/2012 05:06:55 PM

```

The next example illustrates the previous SQL procedure code being made more efficient by converting the UNION to a single query using an IN predicate in a WHERE clause.

SQL Code (Efficient)

```

OPTIONS FULLSTIMER;
PROC SQL;
  SELECT DISTINCT *
    FROM PRODUCTS

```

```

    WHERE UPCASE(PRODTYPE) IN ('SOFTWARE', 'LAPTOP');
QUIT;

```

SAS Log Results

```

OPTIONS FULLSTIMER;
PROC SQL;
  SELECT DISTINCT *
    FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) IN ('SOFTWARE', 'LAPTOP');
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.07 seconds
      user cpu time     0.01 seconds
      system cpu time   0.04 seconds
      memory            561.41k
      OS Memory         6768.00k
      Timestamp         06/03/2012 05:12:56 PM

```

Another approach that can improve the way a query with a SET operator performs is to specify the ALL keyword, as long as duplicates are not an issue or the rows in the table are all unique. Because the ALL keyword prevents SQL from processing the data twice and does not remove duplicate rows, CPU resources might be improved. The next example shows the UNION ALL coding construct being used to perform what amounts to an append operation, thereby bypassing the sort altogether because the duplicate rows are not removed.

SQL Code (Less Efficient)

```

OPTIONS FULLSTIMER;
PROC SQL;
  SELECT *
    FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) = 'LAPTOP'

UNION ALL

SELECT *
  FROM PRODUCTS
    WHERE UPCASE(PRODTYPE) = 'SOFTWARE';
QUIT;

```

SAS Log Results

```

OPTIONS FULLSTIMER;
PROC SQL;
  SELECT *
    FROM PRODUCTS
      WHERE UPCASE(PRODTYPE) = 'LAPTOP'
UNION ALL
SELECT *
  FROM PRODUCTS
    WHERE UPCASE(PRODTYPE) = 'SOFTWARE';
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.06 seconds
      user cpu time      0.00 seconds
      system cpu time   0.04 seconds
      memory            226.42k
      OS Memory         6768.00k
      Timestamp         06/03/2012 05:22:14 PM

```

Summary

1. Performance tuning involves measuring, evaluating, and modifying a query's execution to achieve an optimal balance between competing computer resources (see the "Understanding Performance Tuning" section).
2. Avoid specifying an ORDER BY clause when creating a table or view (see the "Sorting and Performance" section).
3. When sorting is necessary, specify the SORTPGM= system option to instruct the SAS System to use the best possible sort utility relative to the size of the database environment (see the "User-Specified Sorting" section).
4. It is recommended to keep the number of grouping columns with the GROUP BY clause as small as possible (see the "Splitting Tables" section).
5. Care should be exercised to assign indexes to only those discriminating variables in a table and to avoid creating too many indexes (see the "Indexes and Performance" section).
6. There is no advantage in creating or using an index when a table is relatively small (usually fewer than three pages) (see the "Reviewing CONTENTS Output and System Messages" section).
7. Setting the MSGLEVEL= system option to "I" allows SAS to display vital information (if available) relative to the presence of one or more indexes for optimization of WHERE clause processing (see the "Reviewing CONTENTS Output and System Messages" section).
8. Apply WHERE clause processing to restrict the number of rows of the result table (see the "Optimizing WHERE Clause Processing with Indexes" section).
9. When constructing a chain of AND conditions in a WHERE clause, specify the most restrictive conditional values first (see the "Constructing Efficient Logic Conditions" section).

References

- Abolafia, Jeff, and Frank Dilorio. 2008. "Managing The Change And Growth Of A Metadata-Based System." In *Proceedings of the SAS® Global Forum 2008 Conference*. Cary, NC: SAS Institute Inc. <https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/128-2008.pdf>.
- Batkhan, Leonid. 2016. "Modifying variable attributes in all datasets of a SAS library." SAS Users Blog, <http://blogs.sas.com/content/sgf/2016/11/25/modifying-variable-attributes-in-all-datasets-of-a-sas-library/>.
- Batkhan, Leonid. 2017. "CALL EXECUTE made easy for SAS data-driven programming." SAS Users Blog, <https://blogs.sas.com/content/sgf/2017/08/02/call-execute-for-sas-data-driven-programming/>.
- Burlew, Michele M. 2014. SAS® Macro Programming Made Easy, Third Edition. Cary, NC: SAS Institute Inc.
- Cadieux, Richard, and Daniel R. Brethiem. 2014. "Matching Rules: Too Loose, Too Tight, or Just Right?" In *Proceedings of the SAS® Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. <http://support.sas.com/resources/papers/proceedings14/1674-2014.pdf>.
- Carpenter, Art. 2016. *Carpenter's Complete Guide to the SAS® Macro Language, Third Edition*. Cary, NC: SAS Institute Inc.
- Carpenter, Arthur L. 2017. "Building Intelligent Macros: Using Metadata Functions with the SAS® Macro Language." In *Proceedings of the SAS® Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. <http://support.sas.com/resources/papers/proceedings17/0835-2017.pdf>.
- Celko, Joe. 2014. *Joe Celko's SQL for Smarties: Advanced SQL Programming, Fifth Edition*. San Francisco, CA: Morgan Kaufman.
- Cody, Ron. 2010. *SAS® Functions by Example, Second Edition*. Cary, NC: SAS Institute Inc.
- Cody, Ron. 2017. *Cody's Data Cleaning Techniques Using SAS®, Third Edition*. Cary, NC: SAS Institute Inc.
- Davis, Michael. 2001. "You Could Look It Up: An Introduction to SASHELP Dictionary Views." In *Proceedings of the Twenty-Sixth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc. <https://support.sas.com/resources/papers/proceedings/sugi26/p017-26.pdf>.
- Droogendyk, Harry. 2010. "SAS® Formats: Effective and Efficient." In *Proceedings of the 2010 South East SAS Users Group (SESUG) Conference*. https://analytics.ncsu.edu/sesug/2010/FF04_Droogendyk.pdf.
- Dunn, Toby. 2014. "Getting the Warm and Fuzzy Feeling with Inexact Matching." In *Proceedings of the SAS® Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. <http://support.sas.com/resources/papers/proceedings14/1316-2014.pdf>.
- Erinjeri, Jinson J., and Saritha Bathi. 2017. "Applying IFN and IFC Functions." In *Proceedings of the 2017 South East SAS Users Group (SESUG) Conference*. https://analytics.ncsu.edu/sesug/2017/SESUG2017_Paper_153_Final_PDF.pdf.
- Foley, Malachy J. 1999. "FUZZY MERGES: Examples and Techniques." In *Proceedings of the Twenty-Fourth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc. <https://support.sas.com/resources/papers/proceedings/sugi24/AdvTutor/p46-24.pdf>.
- Graebner, Robert W. 2001. "Developing Data-Driven SAS® Programs Using Proc Contents." In *Proceedings of the 2001 MidWest SAS Users Group (MWSUG) Conference*. <https://www.lexjansen.com/mwsug/2001/ApplicationDevelopment/APP-009-developing.pdf>.
- Gupta, Sunil. 2003. *Quick Results with the Output Delivery System*. Cary, NC: SAS Institute Inc.
- Hamilton, Jack. 1998. "Some Utility Applications Using the Dictionary Tables in PROC SQL." In *Proceedings of the 1998 Western Users of SAS Software (WUSS) Conference*, 85-90. <https://www.lexjansen.com/wuss/1998/WUSS98017.pdf>.
- Haworth, Lauren E., Cynthia L. Zender, and Michele M. Burlew. 2009. *Output Delivery System: The Basics and Beyond*. Cary, NC: SAS Institute Inc.

- Hermansen, Sigurd W., and Stanley E. Legum. 2008. "Existential Moments in Database Programming: SAS® PROC SQL EXISTS and NOT EXISTS Quantifiers, and More." In *Proceedings of the SAS® Global Forum 2008 Conference*. Cary, NC: SAS Institute Inc.
<https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/084-2008.pdf>
- Lafler, Kirk Paul. 2005. "Exploring DICTIONARY Tables and SASHELP Views." In *Proceedings of the Thirteenth Annual Western Users of SAS Software Conference*.
https://www.lexjansen.com/wuss/2005/data_warehousing_and_database_management/dwdb_exploring_dictionary.pdf
- Lafler, Kirk Paul. 2006. "Exploring Dictionary Tables with PROC SQL." SAS Press Webinar Series,
<http://support.sas.com/publishing/bbu/webinar.html#lafler2>.
- Lafler, Kirk Paul. 2008. "Undocumented and Hard-to-find PROC SQL® Features." In *Proceedings of the Greater Atlanta SAS Users Group (GASUG) Meeting (June 11th, 2008); Pharmaceutical SAS Users Group (PharmaSUG) Conference (June 1st – 4th, 2008); 2008 Michigan SAS Users Group (MSUG) Meeting (May 29th, 2008); 2008 Vancouver SAS Users Group Meeting (April 23rd, 2008); and 2008 PhilaSUG User Group Meeting (March 13th, 2008)*. <https://www.lexjansen.com/pharmasug/2008/tt/TT02.pdf>
- Lafler, Kirk Paul. 2009. "DATA Step versus PROC SQL Programming Techniques." In *Proceedings of the 2009 South East SAS Users Group (SESUG) Conference*.
<https://analytics.ncsu.edu/sesug/2009/FF003.Lafler.pdf>
- Lafler, Kirk Paul. 2009. "Exploring DICTIONARY Tables and SASHELP Views." In *Proceedings of the 2009 Western Users of SAS Software (WUSS) Conference and 2009 Pharmaceutical SAS Users Group (PharmaSUG) Conference*. <https://www.lexjansen.com/wuss/2009/dmw/DMW-Lafler.pdf>
- Lafler, Kirk Paul. 2012. "Exploring DICTIONARY Tables and SASHELP Views." In *Proceedings of the 2012 South Central SAS Users Group (SCSUG) Conference and Kansas City SAS Users Group (KCSUG) Meeting*.
<https://www.lexjansen.com/scsug/2012/Exploring-DICTIONARY-Tables-and-SASHELP-Views-SCSUG-2012.pdf>.
- Lafler, Kirk Paul. 2013. *PROC SQL: Beyond the Basics Using SAS®, Second Edition*. Cary, NC: SAS Institute, Inc.
- Lafler, Kirk Paul. 2016. "Removing Duplicates Using SAS®." In *Proceedings of the 2016 MidWest SAS Users Group (MWSUG) Conference*. <https://www.lexjansen.com/mwsug/2016/TT/MWSUG-2016-TT02.pdf>.
- Lafler, Kirk Paul. 2016. "Important & Valuable Things You Can Do with SAS® Metadata DICTIONARY Tables and SASHELP Views." In *Proceedings of the 2016 Wisconsin Illinois SAS Users (WIILSU) Conference*.
<http://www.wiilsu.org/sdajgfuirHUTlsdfnloa312/SUSJun2016/Proceedings/Papers/Lafler%20-%20Valuable%20Things%20You%20Can%20Do%20with%20SAS%20DICTIONARY%20Tables%20and%20SASHelp%20Views.pdf>
- Lafler, Kirk Paul. 2017. "Removing Duplicates Using SAS®." In *Proceedings of the 2017 South Central SAS Users Group (SCSUG) Conference*. <https://www.lexjansen.com/scsug/2017/Removing-Duplicates-Using-SAS-SCSUG-2017.pdf>.
- Lafler, Kirk Paul. 2018. "Introduction to Data-driven Programming Using SAS®." In *Proceedings of the 2018 South Central SAS Users Group (SCSUG) Conference*.
[https://www.lexjansen.com/scsug/2018/Introduction%20to%20Data-driven%20Programming%20Using%20SAS%20\(SCSUG%202018\).pdf](https://www.lexjansen.com/scsug/2018/Introduction%20to%20Data-driven%20Programming%20Using%20SAS%20(SCSUG%202018).pdf).
- Lafler, Kirk Paul, and Stephen Sloan. 2017. "A Quick Look at Fuzzy Matching Programming Techniques Using SAS® Software." In *Proceedings of the 2017 Western Users of SAS Software (WUSS) Conference*.
https://www.lexjansen.com/wuss/2017/129_Final_Paper_PDF.pdf.
- Lafler, Kirk Paul, and Stephen Sloan. 2017. "Fuzzy Matching Programming Techniques Using SAS® Software." In *Proceedings of the 2017 South Central SAS Users Group (SCSUG) Conference*.
<https://www.lexjansen.com/scsug/2017/Fuzzy-Matching-Programming-Techniques-Using-SAS-Software-SCSUG-2017.pdf>.
- Lorie, Raymond A., and Jean-Jacques Daudenarde. 1991. *SQL & Its Applications*. Upper Saddle River, NJ: Prentice Hall.
- Matise, Joe. 2016. "Writing Code With Your Data: Basics of Data-Driven Programming Techniques." In *Proceedings of the 2016 South East SAS Users Group (SESUG) Conference*.
https://analytics.ncsu.edu/sesug/2016/BB-229_Final_PDF.pdf.
- McConnell, Steve. 2004. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Redmond, WA: Microsoft Press.

- Olson, Diane. 2000. "Power Indexing: A Guide to Using Indexes Effectively in Nashville Releases." In *Proceedings of the 2000 SAS Users Group International (SUGI) Conference*.
<https://support.sas.com/resources/papers/proceedings/proceedings/sugi25/25/dw/25p124.pdf>.
- Patridge, Charles. 1997. "The Fuzzy Feeling SAS Provides: Electronic Matching of Records without Common Keys." In *Proceedings of the 1997 SAS Users Group International (SUGI) Conference*.
<https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/APPDEVEL/PAPER28.PDF>.
- Perry, William E. 2006. *Effective Methods for Software Testing, Third Edition*. Indianapolis, IN: Wiley Publishing, Inc.
- Raithel, Michael A. 2006. *The Complete Guide to SAS® Indexes*. Cary, NC: SAS Institute Inc.
- Raithel, Michael A. 2016. "Hack 3.21 Saving the Output of PROC CONTENTS to a SAS Data Set." Michael A. Raithel's Blog, <http://michaelraithel.blogspot.com/2016/10/hack-321-saving-output-of-proc-contents.html>.
- Raithel, Michael A. 2016. "PROC DATASETS; The Swiss Army Knife of SAS® Procedures." In *Proceedings of the SAS® Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc.
<http://support.sas.com/resources/papers/proceedings16/3440-2016.pdf>.
- RHO. 2000. "TRANSFORMING SAS DATA SETS." <http://www.rhoworld.com/pdf/ch599.pdf>.
- Roesch, Amanda. 2012. "Matching Data Using Sounds-Like Operators and SAS® Compare Functions." In *Proceedings of the SAS® Global Forum 2012 Conference*. Cary, NC: SAS Institute Inc.
<http://support.sas.com/resources/papers/proceedings12/122-2012.pdf>.
- Russell, Kevin. 2015. "How to perform a fuzzy match using SAS functions." SAS Users blog,
<https://blogs.sas.com/content/sgf/2015/01/27/how-to-perform-a-fuzzy-match-using-sas-functions/>.
- SAS Institute Inc. 2000. "SAS Usage Note 1566: Why duplicate observations occur when using PROC SORT with the NODUPRECS option." <http://support.sas.com/kb/1/566.html>.
- SAS Institute Inc. 2011. *SAS® 9.3 Macro Language: Reference*. Cary, NC: SAS Institute Inc.
- SAS Institute Inc. 2018. *SAS® 9.4 Language Reference: Concepts, Sixth Edition*. Cary, NC: SAS Institute Inc.
- Schreier, Howard. 2006. "SQL Set Operators: So Handy Venn You Need Them." In *Proceedings of the Thirty-first Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc.
<https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/242-31.pdf>.
- Schreier, Howard. 2008. *PROC SQL by Example: Using SQL within SAS®*. Cary, NC: SAS Institute Inc.
- Sloan, Stephen, and Dan Hoicowitz. 2016. "Fuzzy Matching: Where Is It Appropriate and How Is It Done? SAS® Can Help." In *Proceedings of the SAS® Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc.
<http://support.sas.com/resources/papers/proceedings16/7760-2016.pdf>.
- Sloan, Stephen, and Kirk Paul Lafler. 2018. "Fuzzy Matching Programming Techniques Using SAS® Software." In *Proceedings of the 2018 SouthEast SAS Users Group (SESUG) Conference*.
https://www.lexjansen.com/sesug/2018/SESUG2018_Paper-143_Final_PDF.pdf.
- Sloan, Stephen, and Kirk Paul Lafler. 2018. "Fuzzy Matching Programming Techniques Using SAS® Software." In *Proceedings of the SAS® Global Forum 2018 Conference*. Cary, NC: SAS Institute Inc.
<https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2886-2018.pdf>.
- Staum, Paulette. 2007. "Fuzzy Matching using the COMPGED Function." In *Proceedings of the 2007 NorthEast SAS Users Group (NESUG) Conference*. <https://www.lexjansen.com/nesug/nesug07/ap/ap23.pdf>.
- Telles, Matthew A., and Yuan Hsieh. 2001. *The Science of Debugging*. Scottsdale, AZ: The Coriolis Group.
- Teres, Jedediah J. 2011. "Using SQL Joins to Perform Fuzzy Matches on Multiple Identifiers." In *Proceedings of the 2011 NorthEast SAS Users Group (NESUG) Conference*.
<https://www.lexjansen.com/nesug/nesug11/ps/ps07.pdf>.
- Varney, Brian. 2000. "Using Metadata for Data Driven Programming." In *Proceedings of the Twenty-Fifth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc.
<https://support.sas.com/resources/papers/proceedings/proceedings/sugi25/25/cc/25p077.pdf>.
- Villacorte, Renato G. 2012. "Go Beyond The Wizard With Data-Driven Programming." In *Proceedings of the SAS® Global Forum 2012 Conference*. Cary, NC: SAS Institute Inc.
<http://support.sas.com/resources/papers/proceedings12/148-2012.pdf>.

- Wang, Hui. 2015. "Creating Data-Driven SAS® Code with CALL EXECUTE." In *Proceedings of the 2015 PharmaSUG Conference*. <https://www.pharmasug.org/proceedings/2015/BB/PharmaSUG-2015-BB15.pdf>.
- Whitcher, Mike. 2008. "New SAS® Performance Optimizations to Enhance Your SAS® Client and Solution Access to the Database." In *Proceedings of the SAS® Global Forum 2008 Conference*. Cary, NC: SAS Institute Inc. <https://support.sas.com/resources/papers/sgf2008/optimization.pdf>.
- Whitlock, Ian. 1998. "CALL EXECUTE: How and Why." In *Proceedings of the Twenty-Third Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc. <https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/CODERS/PAPER70.PDF>.
- Whitlock, Ian. 2006. "How to Think Through the SAS® DATA Step." In *Proceedings of the Thirty-first Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc. <https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/246-31.pdf>.
- Williams, Christianna S. 2012. "Queries, Joins, and WHERE Clauses, Oh My!! Demystifying PROC SQL." In *Proceedings of the SAS® Global Forum 2012 Conference*. Cary, NC: SAS Institute Inc. <http://support.sas.com/resources/papers/proceedings12/149-2012.pdf>.
- Zirbel, Douglas. 2009. "Learn the Basics of Proc Transpose." In *Proceedings of the SAS® Global Forum 2009 Conference*. Cary, NC: SAS Institute Inc. <http://support.sas.com/resources/papers/proceedings09/060-2009.pdf>.

Ready to take your SAS® and JMP® skills up a notch?



Be among the first to know about new books,
special events, and exclusive discounts.
support.sas.com/newbooks

Share your expertise. Write a book with SAS.
support.sas.com/publish

 sas.com/books
for additional books and resources.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2017 SAS Institute Inc. All rights reserved. M1588358 US.0217


THE POWER TO KNOW®

