Problem Solving Techniques with SQL

Kirk Paul Lafler, Software Intelligence Corporation

Abstract

Solving technical problems involves the use of good techniques. Data processing problems frequently involve a great deal of data manipulation and computing resources. When confronted with problems of this nature, you could approach them using conventional DATA, and/or PROC step methods, or you could use the strengths of the SQL procedure to manipulate and process this data. Attendees will learn how SQL can be used to solve traditional, as well as not so traditional, problems including retrieving and subsetting data, using summary functions to compute statistical information, accessing the read-only dictionary tables to monitor a SAS session, constructing and using views to control what and how data is accessed, performing inner and outer joins for data discovery, and constructing efficient queries that will not only process quickly, but provide ease of maintenance.

Introduction

The SQL procedure is a wonderful tool for querying and subsetting data; creating tables; ordering, grouping, and regrouping data; joining two or more tables (up to 32); and accessing data with views (virtual tables). Occasionally, a problem comes along where the SQL procedure is either better suited or easier to code than conventional DATA and/or PROC step methods. As each situation presents itself, SQL should be examined to see if its use is warranted for the task at hand.

SQL proponents frequently use the power and coding simplicity of SQL to perform an assortment of common everyday tasks to those requiring a highly complex and analytical approach. Whatever the nature of SQL usage, this paper is intended to present a variety of tried and proven techniques that can be used to accomplish a number of ordinary, and not so ordinary, data processing tasks.

Retrieving and Subsetting Data

SQL provides simple, but powerful, retrieval and subsetting capabilities. From inserting a blank row between each row of output, removing rows with duplicate values, using wildcard characters to search for partially known information, and integrating ODS with SQL to create nicer-looking output.

Inserting a Blank Row into Output

SQL can be made to automatically insert a blank row between each row of output. This is generally a handy feature when a single logical row of data spans two or more lines of output. By having SQL insert a blank row between each logical record (observation), you create a visual separation between the rows – making it easier for the person reading the output to read and comprehend the output. The DOUBLE option is specified as part of the SQL procedure statement to insert a blank row and is illustrated in the following SQL code.

SQL Code

```
PROC SQL DOUBLE;
 SELECT *
  FROM MOVIES
   ORDER BY title;
QUIT;
```

Removing Rows with Duplicate Values

When the same value is contained in several rows in a table, SQL can remove the rows with duplicate values. By specifying the DISTINCT keyword prior to the column that is being selected in the SELECT statement automatically removes duplicate rows as illustrated in the following SQL code.

SQL Code

```
PROC SQL;
 SELECT DISTINCT rating
  FROM MOVIES;
QUIT;
```

The output shows that not only are duplicate rows (and values for RATING) removed, but also the unique values are automatically displayed in sorted order.

Output

```
Rating
G
PG
PG-13
R
```

Using Wildcard Characters for Searching

When searching for specific rows of data is necessary, but only part of the data you are searching for is known, then SQL provides the ability to use wildcard characters as part of the search argument. Say you wanted to search for all movies that were classified as an “Action” type of movie. By specifying a query using the wildcard character percent sign (%) in a WHERE clause with the LIKE operator, the query results will consist of all rows containing the word “ACTION” as follows.

SQL Code

```
PROC SQL;
 SELECT title, category
  FROM MOVIES
   WHERE UPCASE(category) LIKE '%ACTION%';
QUIT;
```

Output

Title	Category
Brave Heart	Action Adventure
Jaws	Action Adventure
Jurassic Park	Action
Lethal Weapon	Action Cops & Robber
Rocky	Action Adventure
Scarface	Action Cops & Robber
Star Wars	Action Sci-Fi
The Hunt for Red October	Action Adventure
The Terminator	Action Sci-Fi

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 book, SQL for Smarties: Advanced SQL Programming, he traced the origins of the Soundex algorithm to the developers Margaret O'Dell and Robert C. Russell in 1918.

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, let's search each movie title for the phonetic variation of "Gost" which, by the way, is spelled incorrectly. To help find as many (and hopefully all) possible spelling variations, the sounds-like operator is used to identify select similar sounding names including spelling variations. To find all movies where the movie title sounds like "Gost", the following code is used:

SQL Code

```
PROC SQL;
 SELECT title, category, rating
  FROM MOVIES
   WHERE title =* 'Gost';
QUIT;
```

Output

Title	Category	Rating
Ghost	Drama Romance	PG-13

Integrating ODS with SQL

Little is documented about the output handling capabilities available in SQL. And there's good reason for the lack of information. SQL and the American National Standards Institute (ANSI) guidelines have concentrated almost entirely on capabilities outside the area of reporting. Instead, robust capabilities related to data access, analysis, manipulation, discovery, as well as issues associated with portability have been designed into the language. These features, although well and good, were implemented without any specific mention made to reporting requirements. What this means is that the SQL practitioner has had to find and use alternate tools such as custom report writers offered within the SAS System or by third party vendors to produce quality output. But this is no longer a problem for SQL users in the SAS System.

By integrating Output Delivery System (ODS) statements with SQL code, any output generated by SQL can be made to look nicer without much effort. Although the integration of ODS in SQL code is not part of the ANSI guidelines, ODS does provide SQL users with several output choices. Output can be automatically formatted as traditional monospace output (Listing), RTF, Postscript, HTML, and PDF by using simple ODS statements with the desired output destination (format engine).

Output and Rich Text

To create editable tables in Microsoft Word documents, ODS provides a Rich Text Format (RTF) formatting engine (destination). The RTF destination creates quality-looking output by encapsulating output consisting of specialized fonts, colors, table borders, and other attributes in a packaged, and scalable, RTF file. In the example that follows, ODS is used to create a RTF file that is left justified.

SQL Code

```
OPTIONS nocenter;
ODS LISTING close;
ODS RTF file='c:\output\movies.rtf';
PROC SQL;
 SELECT title, category, rating
  FROM MOVIES
   WHERE rating in ('G','PG')
    ORDER BY title;
QUIT;
ODS RTF close;
ODS LISTING;
```

Output

Title	Category	Rating
Casablanca	Drama	PG
Jaws	Action Adventure	PG
Poltergeist	Horror	PG
Rocky	Action Adventure	PG
Star Wars	Action Sci-Fi	PG
The Hunt for Red October	Action Adventure	PG
The Wizard of Oz	Adventure	G

Output Delivery Goes Web

As the Web continues to take center stage in the world of IT and end-user computing, ODS also allows output to be shared by anyone using a common Web browser. SQL output can be viewed with a Web browser (e.g., MS-IE or Netscape Navigator). By specifying the HTML destination, ODS creates syntactically correct HTML code that is ready for viewing.

ODS allows the SQL practitioner to create exciting new ways of looking at and packaging output. Regardless of whether you plan to deploy the output to the Web, Intranet, Extranet, a server, or simply view it on a single desktop, HTML output offers an exciting alternative to viewing output. The following code creates the necessary HTML files for viewing output using a common Web browser.

SQL Code

```
ODS LISTING close;
ODS HTML path='c:\output'
         body='body.html';
PROC SQL;
 SELECT title, category, rating
  FROM MOVIES
   WHERE rating in ('G','PG')
    ORDER BY title;
QUIT;
ODS HTML close;
ODS LISTING;
```

Output

Title	Category	Rating
Casablanca	Drama	PG
Jaws	Action Adventure	PG
Poltergeist	Horror	PG
Rocky	Action Adventure	PG
Star Wars	Action Sci-Fi	PG
The Hunt for Red October	Action Adventure	PG
The Wizard of Oz	Adventure	G

Using Summary Functions to Group Data

Occasionally it may be important to display data in designated groups. To accomplish this, a GROUP BY clause is used to order groups of data having a column value in common. By aggregating common column values, the output is grouped with data containing the same value. When a GROUP BY clause is used without a summary function, the GROUP BY is transformed into an ORDER BY clause and processed.

When a GROUP BY clause is used with a summary function, the rows are aggregated in a series of groups. What this means is that an aggregate function is evaluated on a group of rows and not on a single row at a time. For example, suppose you wanted to find the shortest movie by rating in the MOVIES table. You could use the GROUP BY clause in a single statement as follows.

SQL Code

```
PROC SQL;
 SELECT rating,
        MIN(length)
        LABEL="Shortest Length"
  FROM MOVIES
   GROUP BY rating;
QUIT;
```

Output

```
         Shortest
Rating     Length
G             101
PG            103
PG-13          97
R             105
```

Subsetting Groups of Data

When processing groups of data, it is frequently useful to be able to subset aggregated rows (or groups) of data. This way, aggregated data can be filtered one group at a time in contrast to the WHERE clause where individual rows of data are filtered one row at a time. SQL provides a convenient way to subset (or filter) groups of data by using the GROUP BY and HAVING clauses. The HAVING clause is applied after the summary of all observations.

Suppose you wanted to identify only those movie-rating groupings that have an average length of less than 90 minutes. Your first inclination might be to use a summary function in a WHERE clause. But, this would not be valid because a WHERE clause is designed specifically to evaluate a single row at a time.

This is in direct contrast with the way a summary function processes data since summary functions evaluate groups of rows at a time, not a single row of data at a time as with a WHERE clause. The easiest and best way of identifying and selecting the movie rating groups is by using the GROUP BY and HAVING clauses together, as follows.

SQL Code

```
PROC SQL;
 SELECT rating,
        AVG(length) FORMAT=3.0
        LABEL='Average Movie Length'
  FROM MOVIES
   GROUP BY rating
    HAVING AVG(length) < 120;
QUIT;
```

Output

Category	Average Movie Length
Action Sci-Fi	116
Adventure	101
Comedy	104
Drama	117
Drama Mysteries	105
Drama Suspense	118

Creating and Using Views

There are many reasons for constructing and using views. A few of the more common reasons are presented below.

Minimizing, or perhaps eliminating, the need to know the table or tables underlying structure

Often a great degree of knowledge is required to correctly identify and construct the particular table interactions that are necessary to satisfy a requirement. When this prerequisite knowledge is not present, a view becomes a very attractive alternative. Once a view is constructed, users can simply execute the view. This results in the baseline or underlying tables being processed. As a result, data integrity and control is maintained since a common set of instructions is used.

Reducing the amount of typing for longer requests

Often, a query will involve many lines of instruction combined with logical and comparison operators. When this occurs, there is any number of places where a typographical error or inadvertent use of a comparison operator may present an incorrect picture of your data. The construction of a view is advantageous in these circumstances, since a simple call to a view virtually eliminates the problems resulting from a lot of typing.

Hiding SQL language syntax and processing complexities from users

When users are unfamiliar with the SQL language, the construction techniques of views, or processing complexities related to table operations, they only need to execute the desired view (by specifying its name) using simple calls (or select choices from a menu). This simplifies the process and enables users to perform simple to complex operations with custom-built views.

Providing security to sensitive parts of a table

Security measures can be realized by designing and constructing views designating what pieces of a table's information is available for viewing. Since data should always be protected from unauthorized use, views can provide some level of protection (also consider and use security measures at the operating system level).

Controlling change / customization independence

Occasionally, table and/or process changes may be necessary. When this happens, it is advantageous to make it as painless for users as possible. When properly designed and constructed, a view modifies the underlying data without the slightest hint or impact to users, with the one exception that results and/or output may appear differently. Consequently, views can be made to maintain a greater level of change independence.

Types of Views

Views can be typed or categorized according to their purpose and construction method. Joe Celko, author of SQL for Smarties and a number of other SQL books, describes views this way, *"Views can be classified by the type of SELECT statement they use and the purpose they are meant to serve."* To classify views in the SAS System environment, one must also look at how the SELECT statement is constructed. The following classifications are useful when describing a view's capabilities.

Single-Table Views

Views constructed from a single table are often used to control or limit what is accessible from that table. These views generally limit what columns, rows, and/ or both are viewed.

Ordered Views

Views constructed with an ORDER BY clause arrange one or more rows of data in some desired way.

Grouped Views

Views constructed with a GROUP BY clause divide a table into sets for conducting data analysis. Grouped views are more often than not used in conjunction with aggregate functions (see aggregate views below).

Distinct Views

Views constructed with the DISTINCT keyword tell the SAS System how to handle duplicate rows in a table.

Aggregate Views

Views constructed using aggregate and statistical functions tell the SAS System what rows in a table you want summary values for.

Joined-Table Views

Views constructed from a join on two or more tables use a connecting column to match or compare values. Consequently, data can be retrieved and manipulated to assist in data analysis.

Nested Views

Views can be constructed from other views, although extreme care should be taken to build views from base tables.

Creating Views

When creating a view, its name must be unique and follow SAS naming conventions. Also, a view cannot reference itself since it does not already exist. The following example illustrates the process of creating an SQL view. In this example, no output is produced since the view must first be created.

SQL Code

```
PROC SQL;
 CREATE VIEW G_MOVIES AS
  SELECT title, category, rating
   FROM MOVIES
    WHERE rating = 'G'
     ORDER BY title;
 SELECT *
  FROM G_MOVIES;
QUIT;
```

In this example the CREATE VIEW statement tells the SAS System that a view is to be created using the instructions and conditions specified in the SELECT statement. The resulting view, G_MOVIES, looks and behaves like a real table, although a virtual table is created.

Outer Joins

Most often, we think of joins as being able to relate rows in one table with rows in another. But occasionally, you may want to include rows from one or both tables that have no related rows. This concept is referred to as row preservation and is a significant feature offered by the outer join construct.

There are operational and syntax differences between inner (natural) and outer joins. First, the maximum number of tables that can be specified in an outer join is two (the maximum number of tables that can be specified in an inner join is 32). Like an inner join, an outer join relates rows in both tables. But this is where the similarities end because the result table also includes rows with no related rows from one or both of the tables. This special handling of "matched" and "unmatched" rows of data is what differentiates an outer join from an inner join.

An outer join can accomplish a variety of tasks that would require a great deal of effort using other methods. This is not to say that a process similar to an outer join could not be programmed – it would probably just require more work. Let's take a look at a few tasks that are possible with outer joins:

- List all customer accounts with rentals during the month, including customer accounts with no purchase activity.
- Compute the number of rentals placed by each customer, including customers who have not rented.
- Identify movie renters who rented a movie last month, and those who did not.

Another obvious difference between an outer and inner join is the way the syntax is constructed. Outer joins use keywords such as LEFT JOIN, RIGHT JOIN, and FULL JOIN, and has the WHERE clause replaced with an ON clause. These distinctions help identify outer joins from inner joins.

Finally, specifying a left or right outer join is a matter of choice. Simply put, the only difference between a left and right join is the order of the tables they use to relate rows of data. As such, you can use the two types of outer joins interchangeably and is one based on convenience.

Using Outer Joins

To illustrate a left outer join, the following code shows the result of using a left outer join to identify and match movie titles from the MOVIES and ACTORS tables. The resulting output displays all rows for which the SQL expression, referenced in the ON clause, matches in both tables and all rows from the left table (MOVIES) that did not match any row in the right (ACTORS) table.

SQL Code

```
PROC SQL;
 SELECT movies.title, actor_leading, rating
  FROM MOVIES
   LEFT JOIN
       ACTORS
    ON movies.title = actors.title;
QUIT;
```

Output

Title	Actor_Leading	Rating
Brave Heart	Mel Gibson	R
Casablanca		PG
Christmas Vacation	Chevy Chase	PG-13
Coming to America	Eddie Murphy	R
Dracula		R
Dressed to Kill		R
Forrest Gump	Tom Hanks	PG-13
Ghost	Patrick Swayze	PG-13
Jaws		PG
Jurassic Park		PG-13
Lethal Weapon	Mel Gibson	R
Michael	John Travolta	PG-13
Vacation	Chevy Chase	PG-13
Poltergeist		PG
Rocky	Sly Stallone	PG
Scarface		R
Silence of the Lambs	Anthony Hopkins	R
Star Wars		PG
Hunt for Red October	Sean Connery	PG
The Terminator	Arnold Schwarzene	R
The Wizard of Oz		G
Titanic	Leonardo DiCaprio	PG-13

The following code illustrates the result of using a right outer join to identify and match movie titles from the MOVIES and ACTORS tables. The resulting output displays all rows for which the SQL expression, referenced in the ON clause, matches in both tables (is true) and all rows from the right table (ACTORS) that did not match any row in the left (MOVIES) table.

```
PROC SQL;
 SELECT movies.title, actor_leading, rating
  FROM MOVIES
   RIGHT JOIN
       ACTORS
    ON movies.title = actors.title;
QUIT;
```

Output

Title	Actor_Leading	Rating
Brave Heart	Mel Gibson	R
Christmas Vacation	Chevy Chase	PG-13
Coming to America	Eddie Murphy	R
Forrest Gump	Tom Hanks	PG-13
Ghost	Patrick Swayze	PG-13
Lethal Weapon	Mel Gibson	R
Michael	John Travolta	PG-13
Vacation	Chevy Chase	PG-13
Rocky	Sly Stallone	PG
Silence of the Lambs	Anthony Hopkins	R
Hunt for Red October	Sean Connery	PG
The Terminator	Arnold Schwarzene	R
Titanic	Leonardo DiCaprio	PG-13

Conclusion

The SQL procedure has an assortment of tools and techniques for solving common data processing problems. Although other methods may exist for resolving certain tasks, at times one method stands out as either being the best or possibly the easiest to code. Performing outer joins in the SQL procedure can certainly be classified as a technique worth further research.

Acknowledgments

The author would like to thank the SESUG Leadership for asking me to write and present this paper at SESUG 2002.

References

Celko, Joe, SQL For Smarties: Advanced SQL Programming, Morgan Kaufmann Publishers, Inc., 1995.

Kent, Paul (2000), "SQL Joins – The Long and The Short of It," Technical Specs TS-553, SAS Institute Inc., Cary, NC, USA.

Lafler, Kirk Paul (2002), "A Visual Introduction to SQL Joins," Proceedings of the 27th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1996), "Solving Business Problems with the SQL Procedure," Proceedings of the 21st Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1996), "Frame Your View of Data with the SQL Procedure," Proceedings of the 21st Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1995), "Diving into the SAS System with the SQL Procedure," Proceedings of the 20th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1993), "Advanced SQL Procedure Features," Proceedings of the 18th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1992) and (1991), "Using the SQL Procedure," Proceedings of the 16th and 17th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul and Fran Larsen (1992), "A Comparison of the SQL Procedure with Conventional DATA and Procedure Step Methods," Proceedings of the 17th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Trademark Citations

About the Author

Kirk is a SAS Alliance Partner™ and SAS Certified Professional® with 25 years of experience working with the SAS System. He has written three books and over one hundred articles on computing and technology – all appearing in professional journals, SAS User Groups proceedings. Kirk's popular SAS Tips column appears regularly in the SANDS and SESUG Newsletters, and selected tips have appeared on the SAS Institute website. His expertise includes application design and development, training, and programming using base-SAS, SQL, ODS, SAS/FSP, SAS/AF, SCL, FRAME, and SAS/EIS software.

The author can be reached at:

Kirk Paul Lafler
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
E-mail: KirkLafler@cs.com
Website: http://www.software-intelligence.com