

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 sorting and grouping data, using summary functions, performing two-, three-, multi-table, and outer joins, and constructing subqueries to pass results from one query to another for further processing.

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); constructing views (virtual tables); and creating subqueries. 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.

Although each problem and proposed solution should be examined on their own, an area where the SQL procedure excels is in the joining of two or more tables. In particular, joining two or more tables using the outer join syntax (left, right, and full). One significant difference between outer joins and conventional inner joins is that outer joins can only join two tables at a time, while inner joins can process up to sixteen tables.

The advantages of using an outer join becomes most apparent when trying to determine not only the rows of data that match the WHERE clause, but rows that don't match. Certainly, these techniques can be accomplished using other methods, but the conventions used in the SQL procedure are especially interesting.

This paper presents how views and outer joins can be used to accomplish ordinary data processing tasks.

VIEW ADVANTAGES - WHY USE THEM

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

Minimize, or perhaps eliminate, 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. Consequently, data integrity and control is maintained since a common set of instructions (view) is used.

Reduce 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.

Knowledge of SQL language syntax is non-existent or lacking among users. When users are unfamiliar with the SQL language or the construction techniques of views, 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.

Provide 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 (one should also consider and use security measures at the operating system level).

Change and customization independence. Occasionally, table and/or process changes may be necessary. When this happens, it is advantageous to make it a painless process for users. 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-related 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 a SAS System environment, one looks 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.

```
PROC SQL;
  CREATE VIEW PERM.COLLEGRAD AS
  SELECT LASTNAME, SSN, DOB
  FROM PERM.EMPLOYEE
  WHERE EDUC > 16
  ORDER BY LASTNAME;
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 for all intensive purposes looks and behaves like a real table. Although in this case, something similar to a temporary internal table is created.

OUTER JOIN SYNTAX

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 just require more work. Let's take a look at a few tasks that are possible with outer joins:

- List all customer accounts with purchases during the month, including customer accounts with no purchase activity.
- Compute the number of orders placed by each customer, including customers who have not placed an order.
- Identify automobile owners who had their 35,000-mile checkup at the scheduled interval, 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. The syntax requirements for *left* outer joins follows:

Left Outer Join

```
proc sql;
  title1 'Left Outer Join Syntax';
  select *
    from libref.left
      left join
        libref.right
      on left.key = right.key;
quit;
```

Figure 1.

The syntax requirements for *right* outer joins follows:

Right Outer Join

```
proc sql;
  title1 'Right Outer Join Syntax';
  select *
    from libref.left
      right join
        libref.right
      on left.key = right.key;
quit;
```

Figure 2.

The syntax requirements for a *full* outer join follows:

Full Outer Join

```
proc sql;
  title1 'Full Outer Join Syntax';
  select *
    from libref.left
      full join
        libref.right
      on left.key = right.key;
quit;
```

Figure 3.

USING A LEFT OUTER JOIN

The following data sets are used as input to illustrate left outer join results.

DATASET - AUTOS

Variable	Type	Length
CODE	Char	4
TYPE	Char	15
YEAR	Num	4

Figure 4.

DATASET - INVEST

Variable	Type	Length
CODE	Char	4
GRADE	Char	1
COLOR	Char	8
OWNER	Char	8

Figure 5.

Figure 6 illustrates the result of using a left outer join to identify and match investment autos having an 'A' grade from the AUTOS and INVEST data sets. The resulting output displays all rows for which the SQL expression matched in both tables (is true) and all rows from the left table (AUTOS) that did not match any row in the right (INVEST) table.

Left Outer Join

```
proc sql;
  title1 'Left Outer Join';
  select *
    from work.autos
      left join
        work.invest
      on autos.code = invest.code and
        invest.grade = 'A';
quit;
```

Figure 6.

Figure 7 illustrates the result of using a right outer join to identify and match investment autos having an 'A' grade from the AUTOS and INVEST data sets. The resulting output displays all rows for which the SQL expression is true and all rows from the right table (INVEST) that do not match any row in the left (AUTOS) table.

Right Outer Join

```
proc sql;
  title1 'Right Outer Join';
  select *
    from work.autos
      right join
        work.invest
      on autos.code = invest.code and
        invest.grade = 'A';
quit;
```

Figure 7.

Figure 8 illustrates the result of using a full outer join to identify and match investment autos having an 'A' grade from the AUTOS and INVEST data sets. The resulting output displays all rows for which the SQL expression is true and all rows from both tables (AUTOS and INVEST) that do not match any row in the other table.

Full Outer Join

```

proc sql;
  title1 'Full Outer Join';
  select *
    from work.autos
      full join
      work.invest
    on autos.code = invest.code and
       invest.grade = 'A';
quit;

```

*Figure 8.***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.

Trademark Citations

SAS, SAS Quality Partner, and SAS Certified Professional are registered trademarks of SAS Institute Inc. in the USA and other countries.

® indicates USA registration.

ABOUT THE AUTHOR

Kirk is a SAS Quality Partner® and SAS Certified Professional® with 25 years of experience working with the SAS System. He has authored over one hundred articles on computing and technology that have appeared in professional journals, SUGI, regional SAS User Groups, and local SAS User Groups. His popular SAS Tips column appears regularly in the SANDS and SESUG Newsletters. 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.



Comments and suggestions can be sent to:

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