ABSTRACT
Techniques for solving typical data processing problems often 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. Real world problems will be illustrated using an online tutorial approach to solve problems such as sorting and grouping data, using summary functions, performing two-, three-, and multi-table joins, and constructing subqueries to avoid having to pass the results of 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 16); 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 will present 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 because a common set of instructions (view) are used by all.

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 are 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 this is the case, users need only execute the desired view using simple calls (or select choices from a menu).

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 can be modified without the slightest hint or impact to users, with the one exception that results and/or output may appear differently. Consequently, change independence can be directly influenced by using views.
TYPES OF VIEWS
Views can be typed or categorized according to their purpose and construction method. Joe Celko has this to say about views, "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
Files 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.COLLGRAD 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
The syntax requirements for left outer joins follows:

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

Figure 1.

The syntax requirements for right outer joins follows:

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

Figure 2.

The syntax requirements for a full outer join follows:

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

Figure 3.
**USING A LEFT OUTER JOIN**

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

**DATASET - AUTOS**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>TYPE</td>
<td>Char</td>
<td>15</td>
</tr>
<tr>
<td>YEAR</td>
<td>Num</td>
<td>4</td>
</tr>
</tbody>
</table>

**DATASET - INVEST**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>GRADE</td>
<td>Char</td>
<td>1</td>
</tr>
<tr>
<td>COLOR</td>
<td>Char</td>
<td>8</td>
</tr>
<tr>
<td>OWNER</td>
<td>Char</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 4.

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 is true and all rows from the left table (AUTOS) that do not match any row in the right (INVEST) table.

**Left Outer Join**

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

**Right Outer Join**

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

**Full Outer Join**

```sql
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 6.

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.

**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**

My sincere thanks are extended to Richard Q. Smith, WUSS Applications Development Section Chair and Faith Renee Sloan, WUSS Program Chair for their support during the preparation of this paper and presentation.
AUTHOR CONTACT

The author may be contacted at the following address with questions:

Kirk Paul Lafler
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
Voice: 619.660.2400
E-mail: KirkLafler@cs.com
Web site: www.software-intelligence.com

SAS, SAS/AF, and SAS/FSP are registered trademarks
of SAS Institute Inc., Cary, NC, USA.