Paper 145-2009

Simple Applications of SQL

Howard Schreier, Howles Informatics, Arlington VA

ABSTRACT

PROC SQL introduces structured query language to the SAS® System. It's a language very different from the rest of SAS. That may present a learning challenge to the SAS user. Fortunately, SQL lends itself to learning in small pieces. This paper will illustrate and explain several examples of SAS usage which have immediate practical value.

SORTING BY MAGNITUDE

Suppose there is a table which we want to sort according to the values in a particular column. However, we want the sorting process to pay attention only to the magnitudes of those values. In other words, the signs (positive or negative) are to be ignored. Here is the given table, which we'll call ***NotOrdered*:**

Obs	ID	Amount
1	101	0
2	102	4
3	103	0
4	104	-10
5	105	-8
6	106	-6
7	107	-2
8	108	4
9	109	1
10	110	0

Notice that the values of AMOUNT vary both in magnitude and sign. If we run this PROC SQL step:

```
PROC SQL;
CREATE TABLE Ordered AS
 SELECT *
  FROM NotOrdered
 ORDER BY ABS(Amount) , ID
;
QUIT;
```

it produces the table ***Ordered***:

Obs	ID	Amount
1	101	0
2	103	0
3	110	0
4	109	1
5	107	-2
6	102	4
7	108	4
8	106	-6
9	105	-8
10	104	-10

The rows having AMOUNT equal to 0 (zero) appear first, with others following according to magnitude but with no regard to sign.

Some other things to notice:

- The PROC SQL statement launches the procedure and the QUIT statement (not a RUN statement) terminates it.
- Within the step, there is just one rather long statement (beginning with "CREATE" and concluding with the semicolon just above "QUIT".
- In the ORDER BY clause, the first key is an expression rather than just a column name. SQL permits the use of expressions in place of simple names just about anywhere that it makes sense.
- There are two keys in the ORDER by clause, separated by a comma. The comma is the standard separator for SQL lists.
- The SELECT clause and its supporting FROM clause constitute the mandatory nucleus of this and any other query.
- The asterisk in the SELECT clause is a shorthand device calling for the inclusion of all columns found in the data source designated in the FROM clause. Later we will see explicit lists used in this context.
- The line beginning with "CREATE" is sort of a preamble, telling PROC SQL to store the results in a table. Were this line omitted, and the statement begun with "SELECT", the results would be handled by the SAS Output Delivery System (ODS) and, by default, would be displayed with other procedure output.
- Because SQL was not invented by SAS, it has its own terminology. We use the terms "table", "row", and "column" rather than the analogous SAS terms "data set", "observation", and "variable".

TABLE LOOKUP

Suppose you have a table along the lines of this one (which we'll call ***NotVerified***):

Obs	Name	Favorite_Proc
1	Larry	PRINT
2	Moe	SORT
3	Einstein	TABULATE

and you want to validate values in the NAME column. The first step is to acquire, from a reliable source, a complete list of legitimate names, and to load them into a table:

```
DATA AllStooges;
* per http://www.threestooges.com ;
INPUT StoogeName $ 1-9;
CARDS;
Larry
Moe
Curly
Shemp
Joe
Curly Joe
;
```

Notice that this is done with a DATA step. Because PROC SQL is intended to complement the rest of SAS and not replace it, it has no capabilities of its own for reading external files or in-stream data. However, PROC SQL can exchange tables with other procedures and with the DATA step, in both directions.

Now the task is to consider in turn each NAME value in ***NotVerified*** and see if it appears in ***AllStooges***:

```
PROC SQL;
CREATE TABLE Verified AS
 SELECT *
  FROM NotVerified
  WHERE Name IN ( SELECT StoogeName
                  FROM AllStooges   )
;
QUIT;
```

Values which cannot be found are excluded from the output (called ***Verified***), which thus looks like this:

Obs	Name	Favorite_Proc
1	Larry	PRINT
2	Moe	SORT

Look again at the code and notice that the second (right) operand of the IN operator, which would typically be a list of literals, is instead a complete query contained within a pair of parentheses. Because of its subordinate position (within another query), it is termed a "subquery". Subqueries are permitted with the IN operator and in a handful of similar contexts where they are more or less natural. Subqueries which yield scalars (as distinguished from lists or vectors) are also permitted just about anywhere they make sense.

COMBINING AND COMPARING LISTS

PROC SQL provides a number of "set" operators that can consolidate or match the content of two lists. To illustrate, consider these two tables, named ***One***:

Obs	City
1	Boston
2	Chicago
3	Cleveland
4	Detroit
5	New York
6	Philadelphia
7	St. Louis
8	Washington

and ***Two***:

Obs	City
1	Boston
2	Brooklyn
3	Chicago
4	Cincinnati
5	New York
6	Philadelphia
7	Pittsburgh
8	St. Louis

Using the UNION operator, we can consolidate the two:

```
PROC SQL;
SELECT City FROM One
UNION
SELECT City FROM Two
;
QUIT;
```

Notice that each operand is a complete query (a SELECT clause followed by its FROM clause). UNION and other set operators do not work within queries, but rather upon the results of two queries.

In this simple example, each table has just a single column (CITY) and each SELECT clause designates just that column. However, the language permits multiple elements to be processed. In such situations, the rows are compared as entities, and there are rules and options to determine the correspondence between columns in the two operands.

Now let's see the result produced by this code:

```
City
------------
Boston
Brooklyn
Chicago
Cincinnati
Cleveland
Detroit
New York
Philadelphia
Pittsburgh
St. Louis
Washington
```

Notice that the duplicative CITY values (Boston, etc.) are eliminated, so that each city's name appears just once. That's the default behavior, but an option is available to carry repetitions into the results.

This output **looks** different than that of the previous examples. That's because the query is not enveloped in a CREATE TABLE statement, and instead displays its output directly. In contrast, earlier results, which **were** stored in tables, were then rendered using PROC PRINT and an Output Delivery System (ODS) style for presentation in this paper.

There are other set operators. INTERSECT delivers rows found in **both** operands. So, if we run:

```
PROC SQL;
SELECT CITY FROM One
INTERSECT
SELECT City FROM Two
;
QUIT;
```

we get:

```
City
------------
Boston
Chicago
New York
Philadelphia
St. Louis
```

The EXCEPT operator reports rows which are found in the first operand but not in the second. Thus, this code:

```
PROC SQL;

SELECT City FROM Two
EXCEPT
SELECT City FROM One
;
```

yields:

```
City
------------
Brooklyn
Cincinnati
Pittsburgh
```

The nature of the EXCEPT operator makes it non-commutative. So, if we reverse the order of the operands, as in:

```
SELECT CITY FROM One
EXCEPT
SELECT City FROM Two
;

QUIT;
```

The result becomes:

```
City
------------
Cleveland
Detroit
Washington
```

Note that no QUIT statement was coded after the first statement using the EXCEPT operator. Instead, the SQL session was kept open for a second statement, then closed. That demonstrates that a PROC SQL session can accommodate multiple statements.

A Data Cleanup Exercise

Here is the scenario for this example: You have a table (named ***Messy***) and want to make sure that any categorical (grouping) columns contain appropriate values.

Dictionary Tables

The first need is to learn the names and characteristics of the columns. SAS automatically makes available to PROC SQL a special series of tables, called Dictionary Tables, which provide metadata (information about data and system entities). One of these, DICTIONARY.COLUMNS, can provide the information we need:

```
PROC SQL;
SELECT name, type, length
 FROM dictionary.columns
 WHERE libname="WORK" and memname="MESSY";
QUIT;
```

The result is:

```
                    Column    Column
Column Name         Type      Length
------------------------------------
Sex                 char           4
ID                  num            8
```

We see that there is just one categorical column (SEX), and that it is of character type with values having a maximum length of 4.

DISPLAYING DISTINCT VALUES

We are only concerned with knowing the various values which appear in the column SEX. We don't care how many times each value appears. If the table has thousands of rows, we certainly don't want to see them reflected individually in the output. Once a particular value is detected, we want to ignore subsequent occurrences of that value. The keyword “DISTINCT” can be coded immediately after “SELECT” to suppress such repetitions. So, to start, all we need is:

```
PROC SQL;
SELECT DISTINCT sex
 FROM Messy;
```

The result we **expect** is something like:

```
Sex
----
F
M
```

However, suppose instead we see this:

```
Sex
----
  F
  M
  f
 F
 M
 f
 m
F
F
M
M
f
f
m
```

DETECTIVE WORK

It seems that the table is named ***Messy*** for a reason! Glancing at the vector, perhaps the most obvious problem is the mixture of upper case and lower case letters. We can distill that out by exercising the UPCASE function, making the query:

```
SELECT DISTINCT UPCASE( sex )
 FROM Messy;
```

This removes the lower-case variations, giving us:

```
----
 F
  M
 F
 M
F
F
M
M
```

Notice the absence of a column heading. In the first iteration (where UPCASE was not coded), the SELECT statement simply designated a column (SEX). That column has a name, so the name was displayed. Now (in the second iteration) the SELECT clause specifies an expression (formula), creating a **new** column, which has no name or label; hence the lack of a header. The language provides ways to declare names and labels for new columns, but that really doesn't seem necessary here.

Continuing with the problem, we now turn to the visibly irregular indentation, presumably caused by leading blanks in some of the values. The LEFT function can get rid of those, so we wrap it around the formula:

```
  SELECT DISTINCT LEFT( UPCASE( sex ) )
   FROM Messy;
```

It seems that should reduce the vector to just two elements, “F” and “M”. However, surprisingly, the result is:

```
  ----
  F
  F
  M
  M
```

There are no **visible** clues to explain the apparent repetitions here. In this situation we can turn to hexadecimal display to reveal what is really (as opposed to apparently) in the table:

```
  SELECT DISTINCT LEFT( UPCASE( sex ) ) ,
                  LEFT( UPCASE( sex ) ) FORMAT = $hex4.
   FROM Messy;
```

Notice that there are two elements (comma-separated, of course) in the SELECT clause. The DISTINCT reduction operates on them jointly. In this case the two columns are generated with the same formula, which avoids pointlessness only because the second element is associated with a different format. The result is:

```
  ----------
  F     4600
  F     4620
  M     4D00
  M     4D20
```

We can see that hexadecimal 46 must be the ASCII code for an upper case “F” and 4D the counterpart for an “M”. A little research can confirm that hexadecimal 20 represents a blank, the expected padding character. The troublemaker is the hexadecimal 00. It can be eradicated with the COMPRESS function:

```
  SELECT DISTINCT COMPRESS( LEFT( UPCASE( sex ) ), '00'X )
   FROM Messy;
```

This code at last produces the expected:

```
  ----
  F
  M
```

so we can finally:

```
  QUIT;
```

All of the functions used in this exercise come from the SAS function library, and are not exclusive to PROC SQL. So even though this exercise has been presented as a mere investigation, the final formula:

```
  COMPRESS( LEFT( UPCASE( sex ) ), '00'X )
```

can used for the needed cleanup, either in PROC SQL or in a DATA step.

Matching

IN SQL, the primary tool for matching data from multiple sources is the "join". To illustrate, suppose that we have two tables. The first, ***Visitors***, contains the results of a survey conducted by a public park district determine the recreational activities which park visitors prefer:

Obs	Visitor	Golf	Swimming	Tennis
1	Aaron	B		A
2	Carol		A	B
3	James	A	C	B
4	Susan		A	

The letters in the table reflect the preference ranking, "A" indicating the individual's favorite activity, "B" the second choice, etc.

The second table, ***Parks***, lists the parks in the district and indicates (by an "X") the facilities available in each park:

Obs	Park	Golf	Swimming	Tennis
1	Harbor		X	X
2	Uptown	X	X	
3	Valley	X		X

Unrestricted Joins

As a preliminary exercise, we'll ignore the information content of these two tables and merely combine the visitor names with the park names. Here is the code:

```
PROC SQL;
CREATE TABLE List_All AS
 SELECT Visitor , Park , 'Hi' AS Placeholder
  FROM Visitors CROSS JOIN Parks
  ORDER BY Visitor , Park
;
QUIT;
```

Notice that the SELECT clause designates the identification columns from the two tables, as well as a new column containing a literal. The AS clause assigns the name "Placeholder" to this new column. The FROM clause is more complicated than in the earlier examples. It names both source tables and specifies that they be combined via a CROSS JOIN process, which means that each and every row in the first source is matched with each and every row in the second, with no regard to content. Here is the result (called ***List_All***):

Obs	Visitor	Park	Placeholder
1	Aaron	Harbor	Hi
2	Aaron	Uptown	Hi
3	Aaron	Valley	Hi
4	Carol	Harbor	Hi
5	Carol	Uptown	Hi
6	Carol	Valley	Hi
7	James	Harbor	Hi
8	James	Uptown	Hi
9	James	Valley	Hi
10	Susan	Harbor	Hi
11	Susan	Uptown	Hi
12	Susan	Valley	Hi

This table is not very informative, but it might be useful, for example as a foundation for recording detailed data on facility usage.

NORMALIZATON

Notice that the given tables (***Parks*** and ***Visitors***) both have a relatively short and wide “grid” structure. In contrast, the derived table ***List_All*** has a long and narrow “list” structure. Neither structure is universally superior, and programmers have to be prepared to deal with both.

The list arrangement is sometimes called a “normalized” structure, and the grid called “denormalized”. The concepts of normalization, which underlie data base design, are actually a lot more complicated and nuanced, but for the purposes of this paper we’ll simplify and just refer to the list and grid structures as being, respectively, normalized and denormalized.

SQL is definitely **not** neutral with regard to normalization. Its features are designed to operate on normalized data, and its outputs tend to materialize in normalized structures. Unfortunately, data aren’t always provided to us that way, and requirements may mandate that end results be denormalized.

As a consequence, we must be able to convert back and forth between list and grid arrangements. PROC SQL itself is not particularly adept at this, but PROC TRANSPOSE is. To facilitate the rest of the examples in this section, we’ll utilize two macros. The first, LIST2GRID, is for denormalization:

```
%MACRO list2grid(data=, out=, by=, id=, var=);
   PROC TRANSPOSE DATA = &data OUT = &out(DROP=_name_);
   BY &by;
   ID &id;
   VAR &var;
   RUN;
   %MEND list2grid;
```

The second, GRID2LIST, normalizes:

```
%MACRO grid2list(data=, out=, by=, id=, var=, colname=);
   PROC TRANSPOSE
     DATA = &data
     OUT=&out(RENAME=( _name_=&id col1=&colname)
              WHERE=( NOT MISSING(&colname) )
             );
   BY &by;
   VAR &var;
   RUN;
   %MEND grid2list;
```

Since PROC TRANSPOSE and the macro language aren't *per se* subjects of this paper, we won't go through the details of just how these two macros do what they do.

To illustrate the use of LIST2GRID, we can call it thusly:

```
%list2grid( data=List_All
           ,out=Grid_all
           ,by=Visitor
           ,id=Park
           ,var=Placeholder)
```

This will call PROC TRANSPOSE to take ***List_All*** and rearrange it as ***Grid_All***:

Obs	Visitor	Harbor	Uptown	Valley
1	Aaron	Hi	Hi	Hi
2	Carol	Hi	Hi	Hi
3	James	Hi	Hi	Hi
4	Susan	Hi	Hi	Hi

Now suppose that we want to use the information we have about people's favored recreational activities, and about the available facilities for these activities. The main reason we didn't do that earlier is that the given denormalized tables made it difficult. So we'll use the GRID2LIST macro once:

```
%grid2list( data=visitors
           ,out=visitors_N
           ,by=Visitor
           ,id=Activity
           ,var= Golf Swimming Tennis
           ,colname=Rank)
```

to create ***Visitors_N***:

Obs	Visitor	Activity	Rank
1	Aaron	Golf	B
2	Aaron	Tennis	A
3	Carol	Swimming	A
4	Carol	Tennis	B
5	James	Golf	A
6	James	Swimming	C
7	James	Tennis	B
8	Susan	Swimming	A

and a second time:

```
%grid2list( data=parks
           ,out=parks_N
           ,by=Park
           ,id=Activity
           ,var= Golf Swimming Tennis
           ,colname=X)
```

to produce ***Parks_N***

Obs	Park	Activity	X
1	Harbor	Swimming	X
2	Harbor	Tennis	X
3	Uptown	Golf	X
4	Uptown	Swimming	X
5	Valley	Golf	X
6	Valley	Tennis	X

With the normalized tables ***Visitors_N*** and ***Parks_N***, we are prepared to do some more interesting joins.

RESTRICTED JOINS

Now that both data sources (pertaining to park facilities and to visitor interests) have been normalized, we can easily code a join which will use all of the information:

```
PROC SQL;
CREATE TABLE List_Match AS
 SELECT   Visitor
        , Park
        , Visitors_n.Activity
        , Rank
  FROM Visitors_n INNER JOIN Parks_n
   ON Visitors_n.Activity = Parks_n.Activity
 ORDER BY Visitor , Activity , Park
;
QUIT;
```

The CROSS JOIN used earlier has been replaced by an INNER JOIN. That requires the inclusion of a restrictive ON clause, which in this example excludes all non-matching pairs. The SELECT clause now includes the type of activity and the preference ranking. The result (***List_Match***) looks like this:

Obs	Visitor	Park	Activity	Rank
1	Aaron	Uptown	Golf	B
2	Aaron	Valley	Golf	B
3	Aaron	Harbor	Tennis	A
4	Aaron	Valley	Tennis	A
5	Carol	Harbor	Swimming	A
6	Carol	Uptown	Swimming	A
7	Carol	Harbor	Tennis	B
8	Carol	Valley	Tennis	B
9	James	Uptown	Golf	A
10	James	Valley	Golf	A
11	James	Harbor	Swimming	C
12	James	Uptown	Swimming	C
13	James	Harbor	Tennis	B
14	James	Valley	Tennis	B
15	Susan	Harbor	Swimming	A
16	Susan	Uptown	Swimming	A

There is a row for each location having a facility for each activity of interest of each person. For the sake of compact presentation, the LIST2GRID macro can be invoked:

```
%list2grid( data=List_Match
           ,out=Grid_Match
           ,by=Visitor Activity
           ,id=Park
           ,var=Rank)
```

Here is the output (***Grid_Match***):

Obs	Visitor	Activity	Uptown	Valley	Harbor
1	Aaron	Golf	B	B	
2	Aaron	Tennis		A	A
3	Carol	Swimming	A		A
4	Carol	Tennis		B	B
5	James	Golf	A	A	
6	James	Swimming	C		C
7	James	Tennis		B	B
8	Susan	Swimming	A		A

Perhaps this is too much information for the immediate needs, and that instead of identifying each activity of interest for each person at each park, we really want only the **number** of activities of interest for each person at each park. To get that, we can alter the code as follows:

```
PROC SQL;
CREATE TABLE List_Summary AS
 SELECT   Visitor
        , Park
        , N(Parks_n.Activity) AS Activity_Count
  FROM Visitors_n INNER JOIN Parks_n
   ON Visitors_n.Activity = Parks_n.Activity
  GROUP BY Visitor , Park
 ORDER  BY Visitor , Park
;
QUIT;
```

The detail on specific activities and their preference ranks has been removed from the SELECT clause, in favor of a summary function (in this case, the N function, which provides simple counts). The GROUP BY clause has been inserted to stratify the process and produce a separate count for each park/person pair. Here is the output (***List_Summary***):

Obs	Visitor	Park	Activity_Count
1	Aaron	Harbor	1
2	Aaron	Uptown	1
3	Aaron	Valley	2
4	Carol	Harbor	2
5	Carol	Uptown	1
6	Carol	Valley	1
7	James	Harbor	2
8	James	Uptown	2
9	James	Valley	2
10	Susan	Harbor	1
11	Susan	Uptown	1

Notice that the summarization has reduced the number of rows in the output from 16 to 11.

As before, we can call the LIST2GRID macro to re-shape the table:

```
%list2grid( data=List_Summary
            ,out=Grid_Summary
            ,by=Visitor
            ,id=Park
            ,var=Activity_Count)
```

giving us ***Grid_Summary***:

Obs	Visitor	Harbor	Uptown	Valley
1	Aaron	1	1	2
2	Carol	2	1	1
3	James	2	2	2
4	Susan	1	1	.

To illustrate another important SQL feature, let's suppose that we require a bit more focus. Specifically, let's exclude the park/person pairs which don't include the person's favorite (A-ranked) activity. That's accomplished by including a HAVING clause:

```
PROC SQL;
CREATE TABLE List_A_Only AS
 SELECT   Visitor
        , Park
        , N(Parks_n.Activity) AS Activity_Count
  FROM Visitors_n INNER JOIN Parks_n
   ON Visitors_n.Activity = Parks_n.Activity
  GROUP BY Visitor , Park
  HAVING MIN(Rank) = 'A'
 ORDER BY Visitor , Park
;
QUIT;
```

HAVING clauses are filters, as are the WHERE clauses we saw in an earlier example. The difference is that HAVING clauses refer to summary functions (MIN in this case), and are necessarily performed later in the sequence of processing. Here is the result (***List_A_Only***):

Obs	Visitor	Park	Activity_Count
1	Aaron	Harbor	1
2	Aaron	Valley	2
3	Carol	Harbor	2
4	Carol	Uptown	1
5	James	Uptown	2
6	James	Valley	2
7	Susan	Harbor	1
8	Susan	Uptown	1

The additional restriction has further reduced the size of the result, from 11 rows to 8.

Once again we'll use %LIST2GRID to denormalize:

```
%list2grid( data=List_A_Only
           ,out=Grid_A_Only
           ,by=Visitor
           ,id=Park
           ,var=Activity_Count)
```

That produces ***Grid_A_Only***:

Obs	Visitor	Harbor	Valley	Uptown
1	Aaron	1	2	.
2	Carol	2	.	1
3	James	.	2	2
4	Susan	1	.	1

CONCLUSION

The examples we've explored have been pretty realistic, and most of the code shown is at least adaptable for real-world tasks. However, the features of SQL, and the nuances of its usage, go far beyond what was presented here. See the references for sources of additional information.

REFERENCES

- SAS Institute Inc. 2008. *Base SAS® 9.2 Procedures Guide*. Cary, NC: SAS Institute Inc.
- Schreier, Howard. 2008. *PROC SQL by Example: Using SQL within SAS®*. Cary, NC: SAS Institute Inc.
  (http://tinyurl.com/sqlbook)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Howard Schreier
Howles Informatics
Arlington VA

703-979-2720

hs AT howles DOT com
http://howles.com/saspapers/
http://sascommunity.org/wiki/Howard_Schreier