Paper 250-31

Proc SQL – A Primer for SAS® Programmers

Jimmy DeFoor
Associate Project Manager
Data Base Marketing
CitiCard
Irving, Texas

Abstract

The Structured Query Language (SQL) has a very different syntax and, often, a very different method of creating the desired results than the SAS Data Step and the SAS procedures. Only a very thorough manual, such as the *SAS® Guide to the Proc SQL Procedure*, could even begin to describe the complete syntax and capabilities of Proc SQL. Still, there is value in presenting some of the simpler capabilities of Proc SQL, especially those that are more efficient or easier to code than the SAS Data Step. The reasons:

1. The tables created by Proc SQL can be read by the SAS Data Step or SAS procedures, so the SAS programmer can choose to use only some SQL code without impacting the rest of his or her SAS code.
2. Understanding Proc SQL can aid the programmer in understanding other SQL code, such as DB2 and Oracle.

Approach

This paper will assume that the reader is a capable SAS programmer, but is fairly uninformed about Proc SQL. It will use familiar coding techniques in the SAS Data Step and SAS procedures to teach the syntax and function of Proc SQL as it shows some of the unique capabilities of the Proc SQL. The method of instruction, in general, will be to define a particular coding objective and then show how that objective could be accomplished with typical SQL code and then with typical SAS code. Thereafter there will be a discussion of the purpose of portions of the code and the reason for the syntax that was used.

General Proc SQL Syntax

```
Proc SQL options;
  Select column(s)
    From table-name | view name
       Where expression
      Group by column(s)
        Having expression
      Order By column(s)
       ;
 Quit;
```

Proc SQL works with columns, rows, and tables, but they are the equivalent of variables, observations, and SAS data sets. In fact, SAS uses only the terminology of columns, row, and tables in the SAS explorer window for both SAS data sets and tables. The browser makes no reference to variables, observations, and data sets.

The Select statement determines the columns that will be retrieved and the operations that will be performed on them. The From clause states the source of the rows and columns. The Where clause limits the rows 'read' from a table, while the Having clause limits the output of the summary operations executed upon the rows. Having clauses must be used with Group By clauses or with functions that operate on all rows that were retrieved. A Group By clauses put the rows into the unique groupings of column values, but it does not order those groupings. If a descending or ascending order of columns is required, an Order By clause must be used. Ascending is the default.

Examples of SQL Code and Comparable SAS Code

Example 1: Creating an output listing with Proc SQL versus a Data Step

```
Filename out 'C':\temp.txt' new;
*;
Proc Printo print=out;
Run;
*;
Proc SQL;
  Select monotonic() as obs,
         a.State, a.City as Town,
         a.Store, a.Year,
         a.Month, a.Sales,
         a.VarCost, a.FixedCost
      from Cost a;
  quit;
```

```
Filename out 'C':\temp.txt' new;
*;
Proc Printo print=out;
Run;
*;
Proc Print data = Cost;
  Label City = 'Town';
  Var State City Store
      Year  Month Sales
      VarCost FixedCost;
Run;
```

Since Proc SQL is a procedure, it requires a previous Proc Printto to be executed if the listing output is to be directed to a text file instead of the default list file SAS uses. Proc Print has the same requirement

Proc Print generates observation numbers automatically; Proc SQL doesn't. Observation numbers can be generated only by adding Number as an option on the Proc SQL statement or by adding the undocumented function of monotonic to the Select statement. The Number option is ignored when a table is created, however, while the monotonic function isn't. The monotonic function is shown here.

Notice the reference of 'as' that follows 'monotonic'. It provides names to the results of functions, but it can also be used to rename variables. In this Select statement, it names the results of the monotonic function as Obs and renames City to Town. The Label statement in the Proc Print code accomplishes the same result as far as printing is concerned.

Compare the syntax of the Proc SQL to syntax of the Proc Print. Doing so reveals the basic syntax of Proc SQL. Commas separate the variables listed in the Select statement of Proc SQL. The Quit statement is the terminator of Proc SQL, not the Run statement. The semi-colon does not end every instruction, as it does in usual SAS code. Instead, it ends only the SQL Select statement, which is really the only statement in SQL.

When SAS encounters the semi-colon that ends the Select statement, it passes all of the preceding SQL code to the SQL compiler. The compiler separates the Select statement into individual SQL clauses as it encounters such key words as From, Where, and On. It then evaluates the clauses and passes them to the execution module if they are syntactically correct. Thereafter, SQL processes all of the clauses and waits for another Select statement unless it encounters a Quit statement or another SAS step. Thus, Proc SQL can create multiple tables and/or listings until it is closed.

One unique and important feature of Proc SQL is that columns can be defined relative to the data set in which they occur. This is shown by the a.column references above, such as a.State. The programmer creates an alias by following the table name with a shorter name (as in Cost a). He or she can then attach the alias to the front of the variable name with a period. This ability to reference variables by their data sets allows multiple variables with the same name to be manipulated by Proc SQL. This will be shown in Example 11 when a table is created by joining two other tables.

Example 2: Creating a table with Proc SQL versus creating it with a Data Step

The simplest way of creating a SAS data set or table is to read into it all the variables and records of another SAS data set. In the code below, the **Select *** syntax retrieves all variables from the Measures data set. The SAS Data Step retrieves all variables by default. Only the presence of a Keep or Drop statement as a Data Set option prevents all variables in the input data set from being read into the SAS Data Step. SQL either retrieves all variables or only the specified variables. It does not explicitly drop variables.

Keeping all columns from the source table regardless of order

```
Proc SQL;
  Create Table Work1 as
     Select *
      from Datamart.Measures a;
  quit;
```

```
Data Work1;
  Set Datamart.Measures;
  Output;
Run;
```

The code below shows the Select statement retrieving particular variables from the Measures data set and writing them to the table Sales. As discussed above, the Data Step must accomplish the same result via a Keep, which in this case is the Keep= data set option. However, the Select statement has another result that cannot be easily duplicated in the SAS data step. It orders the variables as it retrieves them. The Keep statement, in any form, does not specify order. Only Length statements or Format statements will set the order of variables in a Data Step and they must be placed before the Set statement if they are going to determine the order of all variables in the data set. The problem is, however, that they require some knowledge of the variables' content and existing formats if they are to be used effectively. The SQL Select statement does not require such knowledge.

Keeping and ordering particular columns from the source table

```
Proc SQL;
   Create Table Sales as
     Select
       a.State, a.City,
       a.Store, a.Year,
       a.Month, a.Sales
    from
      work.Datamart.Measures a;
quit;
```

```
Data Sales;
  Length State $12 City $20
         Store $06 Year $04
         Month $02 Sales  8;
  Set Datamart.Measures
      (keep= State City  Store
             Year  Month Sales);
  Output;
Run;
```

On the other hand, though the Data Step cannot match the ease of variable ordering that can be done in Proc SQL, it can output multiple records to the same data set or to multiple data sets very easily. For example, the data statement above could easily be modified to write part of the record to Sales and part to a data set named CityList.

```
Data Sales(keep=Sales) CityList(keep=City);
```

Proc SQL would require two Create Table clauses with different Select statements to accomplish the same result.

The next example discusses sorting data with Proc SQL and computing new variables.

Example 3: Creating and assigning columns and selecting and sorting rows

```
Proc SQL;
   Create Table Stats as
     Select
       a.State, a.City,  a.Store,
       a.Year,  a.Month, a.Varcost,
       sum(a.VarCost, a.FixedCost) as
        TotCost,
     from
       work.Cost a
        Where state = 'TX'
   Order by State, City, Store,
            Year, Month
   ;
quit;
```

```
Proc Sort Data = Cost
    (keep=State City Store Year
          Month VarCost FixedCost
     where=(State='TX'))
          Out  = Subset;
  By State City Store Year Month;
Run;
*;
Data Stats(drop=FixedCost);
  Set Subset;
  Totcost = sum(Varcost, FixedCost);
  output
Run;
```

The above examples perform four actions:

1. Selects only particular variables such as City, State, Store, Year, Month, and Varcost.
2. Retrieves values for those variables from only the state of 'TX'.
3. Orders the rows by State, City, Store, Year, and Month.
4. Creates a Total Cost field.

In both examples, the sum function was used to create the values of TotCost from the values of VarCost and Fixed Cost. Other calculations and assignments could have been done with very similar coding. The only difference would have been the form of the syntax. In SQL, the form is Action > Target. In the Data Step, the form is Target < Action. This is shown below.

SQL	**Data Step**
`(1.30 * Cost) as Price`	`Price = 1.30 * Cost;`
`Year||Month as Yearmo`	`Yearmo = Year||Month;`
`Substr(Store,1,3) as StoreArea`	`Storearea = Substr(Store,1,3);`

In this example, the SQL code performed the sorting with an Order clause after it executed the selection, calculation, and assignment. The SAS code selected, sorted, calculated, and then assigned. For the SAS code, this was by choice. The sort could just as easily been executed after the Data Step as before. But Example Six builds on this code and it requires that the sort be done before the summing.

TotCost is calculated in both the SQL and Data Step code via the Sum function. The capabilities of the SQL Sum function are more extensive than the Sum function in the SAS Data Step, however. In the Data Step, the Sum function always sums over columns. In SQL, it **sums over rows** if only one column is listed in the Sum function and **sums over columns** if two or more columns are present. Since functions can be nested, a Sum function in SQL can sum over rows the result of a Sum function that summed over columns (see below).

```
 sum(sum(a.VarCost, a.FixedCost)) as GrandTotal
```

In this case, GrandTotal would be the sum of all rows of variable and fixed costs in the entire table.

Example 4: Creating columns with Case expressions and user-built formats

In SQL, case expressions and user-built formats are very useful methods of assigning values to a variable based upon the values of other variables.

```
Proc SQL;
  Create Table Stats as
    Select
      City, State, Year, Month, VarCost,
      Case
        When Year lt ‘2003’ then
            ‘PreviousYear’
        Else
            ‘CurrentYear’
      End
        As Descrptn
          From
             work.Cost a
     ;
```

The Case expression assigns values to a field in the same way an If-Then-Else statement would do in a SAS Data Step. In this example, it assigns the strings **PreviousYear** or **CurrentYear** to Descrptn.

Case expressions are closed with End references. Use of the Else assignment is recommended, but optional. SQL will assign missing values in its absence - just as would the If-Then-Else statements in a Data Step if the final Else statement was not used.

Case statements can also be nested.

```
Case
    When Year lt ‘2003’ then
        Case
              When Month le ‘06’ then
                  ‘First6Mos’
              Else
                  ‘Second6Mos’
         End
   Else
       ‘PreviousYear’
End
   As Descrptn
```

User-built formats can also be used to assign values with the same coding ease and operational speed they provide in Data Steps. They would be used like this:

```
Select
   City,
   Put(State,$State.) as StateName,
   VarCost,
   FixedCost
```

User-built formats are especially valuable in Proc SQL because of their flexibility in assigning many values with a small amount of code. Case statements, though useful, seem to clutter SQL code more than If-then-Else statements clutter SAS data steps.

Assuming the user-built format looked something like below, the put function in the Select statement would assign to StateName the values ‘Texas’ for ‘TX’, ‘Louisiana’ for ‘LA’, and ‘New York’ for ‘NY’.

```
Proc format;
   Value $State
      ‘LA’ = ‘Louisiana’
      ‘TX’ = ‘Texas’
      ‘NY’ = ‘New York’
       ;
run;
```

Example 5: Concatenating tables and restricting the number of rows

```
Proc SQL;
  Create Table Cost as
     Select *
        From
           ThisYearsCost(obs=max)
              Union All
     Select *
        From
           LastYearsCost(obs=max)
     ;
Quit;
```

```
Data Cost;
  Set ThisYearsCost(obs=max)
      LastYearsCost(obs=max)
      ;
  Output;
Run;
```

The Data Step code on the right concatenates two tables, stacking the second underneath the first and writing out all rows in both data sets. The Union All operator in Proc SQL accomplishes the same result – but only because the positions of the columns are the same in both data sets. If they were in a different order, the SQL code would concatenate the columns incorrectly.

There are five Union operators in Proc SQL. Four perform very different operations than the simple concatenation seen here – and several of those are very difficult to code in the SAS Data Step.

Union	- matches by column position, not name, and drops duplicate rows
Union All	- matches by column position, not name, and doesn’t drop duplicate rows
Union Corresponding	- matches by column name and drops duplicate rows
Except	- matches by column name and drops the rows found in both tables
Intersection	- matches by column name and keeps the unique rows in both tables

The Unions can be combined also. Two tables could be concatenated with a Union Corresponding and then that result intersected with another table to find the records in the last table that were also in either the first table or the second.

Also shown in this example is the use of the **Obs=** data set option to control the number of input records pulled from each table. This option is most important when testing code that will be run against large tables. It allows the coder to pull only some records from the first table and all records that match it from another, which is very helpful in testing joins. Proc SQL doesn't have any other coding construct that controls the number of input records pulled from just one table. In short, the following type of coding structure is not allowed in Proc SQL:

```
Select *
   From Cost
      Where rownum le 100
      ;
```

Example 6: Selecting and assigning columns and summarizing rows

SQL

```
Proc SQL;
  Create Table Stats as
    select                                /* keep only these variables in output */
        State, City, Store, Year,
        sum(VarCost)   as VarCost,        /* sum column values over rows */
        sum(a.TotCost) as TotCost
      from                                /* accessing a select statement within */
        (select                           /*   a from clause is an in-line view  */
          State, City,  Store,
          Year,  Month, VarCost,
          sum(VarCost,  FixedCost)        /* sum row values over columns */
             as TotCost
           from work.Cost
             Where state = 'TX') a        /* retrieve columns for only 'TX' */
    group by State, City, Store, Year     /* sum rows over a group of columns */
    order by State, City, Store, Year     /* order the output by column values */
    ;
 quit;
```

Data Step

```
Data Stats;
 Drop Month      FixedCost                /* drop unneeded columns from output */
      SumVarCost SumTotCost;
 Set Cost(where=(State='TX')              /* keep only rows for 'TX' */
            keep=State City Store Year
                 Month VarCost FixedCost);
 By State City Store Year;                /* group data by these columns */
 If first.year  then
   do;
      SumTotcost = 0;                     /* set summing columns to zero at */
      SumVarcost = 0;                     /*   the first row of each group  */
   end;
 Totcost = Sum(Varcost,FixedCost);        /* sum row values over columns */
 SumTotCost + TotCost;                    /* sum column values over rows */
 SumVarCost + VarCost;
 If last.year  then                       /* find the last row of each group */
   do;
      TotCost = SumTotCost;               /* load values from summing columns */
      VarCost = SumVarCost;               /*   into cost columns              */
      output;                             /* output last row of each group */
    end;
run;
```

This code repeats the actions taken in Example Three, but also uses sum statements or functions to create group totals by State, City, and Year. Comparing the SQL and the Data Step code shows the very explicit coding that must be executed in the Data Step to accomplish the actions taken by the SQL code. The objective is to create a Total cost value for the cities and stores of the state of Texas and to summarize that cost and Variable cost by State, City, Store, and Year.

The reason for sorting the data before executing the Data Step in Example Three now becomes obvious. The Data Step could not have summed the TotCost and VarCost fields by State, City, Store, and Year without the preceding sort. The SQL code, however, can sum without a Proc Sort because it will internally group the data as it is calculating sums for State, City, Store, and Year. This is accomplished via the Group By clause. The Order By only serves to sort the grouped columns so that Fort Worth will not precede Dallas and 2005 will not come before 2004.

The Data Step retrieves the data for the cities in Texas via a **Where=** option on the Set statement. It then uses the By variable statement to establish the first.month and last.month variables that mark the first and last occurrence of each grouping. Next, it uses **a sum function** to compute the total of variable and fixed cost and then uses **two sum statements** to calculate the group totals of Variable and Total cost. Last, it writes out those sums under the original names of the variables via the last.month reference.

In-Line Views

The SQL step in Example Six creates the Total cost column via an in-line view, which – essentially – is a SQL step within a SQL step. An In-line view exists when the outer Select statement retrieves its data from another Select statement and not a table. In this case, the inner Select statement creates the Totcost column as the sum of the Varcost and Fixedcost columns. It then passes that result and Varcost to the outer query, which sums Totcost and Varcost over the rows of each Group By and thereafter writes the results of that summing into new columns that it will name Totcost and Varcost when it writes the results to the Table Stats.

The use of an in-line view wasn't required in order to perform this 'multi' summing, however. It could have been accomplished using the nested summing shown next, which was discussed in Example Three.

```
Proc SQL;
  Create Table Stats as
    Select State, City, Store, Year,
       sum(VarCost) as VarCost,
       sum(sum(VarCost, FixedCost))  /*   nested summing: columns are   */
          as TotCost                 /* summed and then rows are summed */
       from work.Cost a
          Where state = 'TX'
    group by State, City, Store, Year
    order by State, City, Store, Year
    ;
  quit;
run;
```

Yet, in-line views can be very useful even when not necessary, particularly when the programmer wants to break-up complex code so that it can be tested independently or, perhaps, just to make the function of the code more obvious – as it does in Example Six. The nested summing in SQL code above works because SQL knows *by context* that it should be summing over rows after it sums over the columns. It knows this because only one value is present in the external sum function after the internal sum has completed its task of summing columns. Yet, what a programmer understands and what SQL understands may not always be the same.

Two final notes should be made before going onto the next subject. First, further calculations associated with TotCost could have been easily added to both the Data Step code and the SQL code. Here is how the Expected Revenue calculation could have been added to SQL code and to a Data Step.

SQL	Data Step

`sum(VarCost) as VarCost,`
`sum(sum(a.VarCost, a.FixedCost))`
`   as TotCost`
`sum(sum(a.VarCost, a.FixedCost)) * 1.30`
`   as ExpRev`
	`If last.year then`
	`  do;`
	`    TotCost = SumTotCost;`
	`    VarCost = SumVarCost;`
	`    ExpRev  = Totcost * 1.30;`
	`    output;`
	` end;`

Notice, however, that the SQL code does not calculate the Expected Revenue by applying the 1.30 rate to Totcost. Instead, it applies the 1.30 rate to the calculation for TotCost. Normal SQL code cannot, within the same view, use the value of TotCost to calculate ExpRev. This is because the new column does not exist until SQL completes that view. Until then, that column cannot be referenced in the SQL code. On the other hand, a new column can be accessed during the Data Step because it is established at the beginning of the Data Step and has its own position in the program data vector. Thus, its value in the current loop of the program is available as long as it is accessed after it is populated.

The SAS version of SQL, however, bypasses this issue if the coder uses the Calculated reference. Adding the word ‘calculated’ in front of a column will make the SQL processor create a column that can be used by the coder in the same view. As a result, the code below

```
sum(sum(a.VarCost, a.FixedCost))as TotCost,
sum(sum(a.VarCost, a.FixedCost))* 1.30 as ExpRev
```

would become

```
sum(sum(a.VarCost, a.FixedCost))as TotCost,
Calculated TotCost * 1.30 as ExpRev
```

The last line is now very similar to the Data Step code used to calculate Expected Revenue.

```
ExpRev = Totcost * 1.30;
```

Again, though, an in-line view will create the same result and that method can be employed in any SQL code, such as Oracle or DB2.

Example 7: Joining statistical measures onto the same data set

<pre>Proc SQL;
  Create Table Stats as
    Select a.State, a.City, a.Store,
           a.Year, a.Month,
           a.VarCost, b.AvgVarCost,
          (a.VarCost/b.AvgVarCost)*100
            as PctAvgVarCost
      from work.Cost a,
           (select
               mean(b.VarCost)
                 as AvgVarCost
             from
                work.Cost b)
             ;
  quit;</pre>
	<pre>Proc summary data = Cost;
	  Var VarCost;
	  Output out = summary
	              (Drop=_type_ _freq_)
	     Mean = AvgVarCost;
	Run;
	*;
	Data Stats;
	  Set Cost;
	  If not eof1 then
	    Set Summary end=eof1;
	  PctAvgVarCost =
	    (VarCost/AvgVarCost)*100;
	  Output;
	Run;</pre>

Analysts often compare the values of a variable. This is frequently accomplished by comparing the individual values of the variable to the average of all values of that variable. This requires that the average be calculated for the variable and then joined with the individual values so that they can be compared. Typical SAS programming requires that the average be calculated in a previous step and then merged with the original data so that percentages can be calculated.

On the right above is a Proc Summary that computes the average of VarCost. That average is then added to each row of the original data via a Data Step in which there are two Set statements. The first is the original data set; the second is the data set containing the average cost. The use of two Set Statements, with the second executed only once, causes the average cost value to be read once into the program data vector and then retained for all subsequent reads from the original data set Cost.

The SQL code accomplishes the same result in much the same way: the average of VarCost is calculated with an in-line view and then joined back onto the data from the table Cost. This joining causes SQL to issue a warning that a Cartesian join is being performed, which is when all rows of the second data set are joined to each row of the first data set.

Remerging computed values

Though the Cartesian join accomplished the objective, it isn't the usual way in SQL to add a computed value onto its original table. The typical method is a remerge and it has the following structure.

```
Proc SQL;
  Create Table Stats as
     select
       a.State, a.City, a.Store,
       a.Year, a.Month, a.VarCost,
       mean(a.VarCost) as AvgVarCost,
      (a.VarCost/calculated AvgVarCost)*100
          as PctAvgVarCost
         from
           work.Cost a
          ;
  quit;
```

Notice that the mean function is now part of the outer select. Notice also that the Calculated reference has been added to the computation of PctAvgCost. Because the in-line view has been removed, SQL must now be told to create a reference for AvgVarCost. Otherwise, the SQL code will fail with a message similar to 'AvgVarCost does not exist'.

Example Eight shows the use of the Having clause. While the Where statement is used in evaluating the values of fields on individual records, the Having clause is used to evaluate fields that result from the summary of a group of records, perhaps even from all of the records in the table.

Example 8: Using Having clauses to evaluate records summed over a Group By

```
Proc SQL;
  Create Table Stats as
    select
       a.State, a.City, a.Store,
       a.Year, a.VarCost,
       mean(a.VarCost) as AvgVarCost
    from
         work.Cost a
       having Varcost gt AvgVarCost
    Group by State, City, Store, Year
    Order by State, City, Store, Year
  quit;
```

```
Proc summary data = Cost;
  By State City Store Year;
  Var VarCost;
  Output out = summary
        Mean = AvgVarCost;
Run;
*;
Data Stats;
  Merge Cost
        Summary(drop=_type_ _freq_);
  By State City Store Year;
  If VarCost gt AvgVarCost then
    Output;
Run;
```

The SQL code shows a Having clause selecting records from grouping of State, City, State, and Year that have a variable cost greater than AvgVarCost. The mean function calculates the average over the Group By columns and then the Having clause outputs only the City and Stores that have a VarCost that exceeds the average. This happens during the remerging of the mean values with the Cost table.

As stated earlier, it is the Group By clause that causes the calculation of the average cost by City, State, Year, and Month. The Order By clause only ensures that the order of the fields will be in the expected sort order.

The Data Step in Example Eight shows the SAS code accomplishing the same result using

1. a Proc Summary that computes the average value of variable cost for each combination of State, City, Store, and Year.
2. a Data Step match-merge that joins the average variable cost to the original data set and then outputs only the stores with a variable cost that exceeds the average variable cost.

Example 9: Using Having clauses versus subqueries to evaluate values summed over a table

<table>
<tr>
<td>
<pre>
Proc SQL;
  Create Table Stats as
    Select
      State, City, Store,
      Year, Month, VarCost
        from
           work.Cost a
    having
       VarCost lt (1.30 *
                   min(VarCost))
     ;
  quit;
</pre>
</td>
<td>
<pre>
Proc SQL;
  Create Table Stats as
    Select
      State, City, Store,
      Year, Month, VarCost
        from
           work.Cost a
     where
       VarCost lt (1.30 *
          (Select min(Varcost)
             from
               work.Cost b))
             ;
  quit;
</pre>
</td>
</tr>
</table>

On the left is SQL code using a Having clause to control the records that will output to the Stats table. On the right is SQL code using a Where clause with a subquery to accomplish the same result, which is to output only the Cities and Stores with the lowest variable costs to the Stats table.

The Having clause will execute after the calculation of the measure, which means it is executed after the remerge. The Where clause will also execute after the calculation of the measure, but only because it is operating against a subquery.

Subqueries

A subquery is a Select statement that is executed as part of a Where clause, while an in-line view is a Select statement that executed as part of the From clause. Subqueries create a value or a list of values. In-line views create one or more rows of one or more columns. Since the subquery here will execute before the Where clause, the Where clause will operate on the values retrieved by the subquery. That means that the Where clause would look something like this when it executes.

```
Where VarCost lt (1.30 * 20)
```

Subqueries can return one value, as it does in Example Nine, or multiple values. When multiple values are returned, the Where or Having clause must have an operator that deals with multiple values, such as In or Exists. The In operator is shown next and is the preferred method since it is more efficient than an Exists operator.

```
Select State, City, Store, Year, Month, VarCost
    from
       work.Cost a
 where
   a.VarCost in
      (Select Varcost from
         work.Cost b
            having Varcost gt (1.25 * avg(varcost)))
   ;
```

The subquery is this code retrieves a list of stores that have variable costs 25% larger than the average variable costs of all stores. When it executes, the Where clause might look something like this:

```
Where a.VarCost in (60, 70, 90, 100, 120)
```

Example 10: More on Having clauses versus Where clauses with subqueries.

```
Proc SQL;
  Create Table Stats as
    Select
      State, City, Store,
      Year, Month, VarCost,
      Mean(VarCost) as AvgVarCost
        from
           work.Cost a
    having VarCost lt AvgVarCost))
     ;
  quit;
```

```
Proc SQL;
  Create Table Stats as
    Select
      State, City, Store,
      Year, Month, VarCost,
      Mean(VarCost) as AvgVarCost
        from
           work.Cost a
      where VarCost lt
        (select mean(VarCost)
           from work1)
      ;
  quit;
```

Compare the two code forms. Will they produce the same result? Almost, but there will be one important difference. That value of AvgVarCost in the Stats table will not be the same. Why?

The AvgVarCost value calculated in the outer Select statement of the Proc SQL on the left is the average variable cost of the entire Cost table. The AvgVarCost calculated in the outer Select statement on the right is the average of a subset of the Cost table: it was computed only from the rows of variable cost values that were retrieved by the Where clause. The values of the other columns and the total number of rows retrieved were not affected, however. Why? The average variable cost value actually used to retrieve the rows is the same.

Now, consider the in-line view that was used in Example Seven. Suppose a similar Where clause had been added to that code. Would it have produced the same discrepancy? No, because the average variable cost calculated there was computed on all of the rows in the table and then joined onto all rows of the result table – just as did the remerge with the Having clause above. Thus, if the value used in sub-setting a table must be carried to the result table, then the choice is either the use of a Having clause with a remerge or the use of a Where clause with an in-line view that executes a Cartesian join. The use of a Where clause with a subquery will give a misleading result.

The next example compares the Data Step match-merge to the Proc SQL equijoin.

Example 11: Comparing a Data Step Match-Merge to a Proc SQL Equijoin

```
Proc SQL;
   Create Table Cost as
     Select
       a.State, b.City,
       b.Store, b.Year,
       b.Month, b.Sales,
       b.VarCost, b.FixedCost
    from
      work.ParticularCities a
        left outer join
      Datamart.Measures b
        On a.City = b.City;
quit;
```

```
Data Cost;
  Merge Particular.Cities
           (in=c keep=City)
        Datamart.Measures
          (in=m
           keep=State City Store
                Year Month Sales
                VarCost FixedCost)
  By city;
  If c then
    output;
run;
```

The Data Step code match-merges cities from Particular.Cities with the Measures data set and keeps only the cities in Measures that match the cities in ParticularCities. This is accomplished by the **By City** and the **If C then output** statements. The latter statement outputs only the records for which C has a value of one (1). That happens when the row is from Particular.Cities, even if that city isn't found on Datamart.Measures. The variable C was created by the **in=c** data set option on Particular.Cities.

The SQL code on the left does the same thing, but the action is called an *equijoin* instead of a match-merge. An equioin, like a match-merge, is the joining of two or more tables on common columns. This SQL code does it, however, through the *left outer join* clause.

Example 12 will show each type of coding structure available in SQL equijoins and Data Step match-merges and the function performed by each. The Match-Merge code will reference the variable M and/or the variable C to create the same output as the equijoins.

Before looking at that Example 12, however, several special features of the Proc SQL join should be mentioned. When Proc SQL executes an equijoin, it can perform a many-to-many merge, whereas the Data Step match-merge can only 'correctly' execute a one-to-one or one-to-many merge. When a many-to-many merge is executed, the log for the Data Step will indicate that the resulting data set is suspect.

Proc SQL can also join two tables without a matching column. When it does, the result has every row of the first table joined to every row of all subsequent tables. This is called a Cartesian product. (Example Seven also showed the use of the Cartesian product). It is used in self-joins (when the table is joined with itself) so that a value of a column in a previous time frame or for another product or at another store can compared to the value of the current product, store, or time frame. A self-join could even be used to compare the current salary of an employee to his salary in a previous job in the same company.

Finally, it should be noted that this example assumed that the input data sets were sorted by the variable being match-merged. If they aren't, then the Data Set match-merge will fail. The Proc SQL equijoin won't, however, since it will internally order the data sets before merging.

Example 12: A list of SQL Equijoins and their equivalent Match-Merges

Function	Equijoin code	Match-Merge code
Output all values of the merging variable from the first file and matching records from the second.	Left outer join	If C then output;
Output all values of the merging variable from the second file and matching records from the first.	Right outer join	If M then output;
Output only the values of the merging variable that are on both files.	Inner join	If C and M then output;
Output all values of the merging variable from both files regardless of whether they match.	Full Join	If C or M then output;

This table shows the coding needed to implement each type of equijoin in Proc SQL or match-merge in the SAS Data Step. The complete code that would be executed would be that in Example 11, with the only change being the substitution of the code in this table depending on the output that was desired.

Example 13: Indexes Can Speed Equijoins In Proc SQL

If the table on the right-side of the equijoin is indexed on the joining columns, or if all tables in the index are joined on the joining columns, then Proc SQL can execute an equijoin with far more speed than will the SAS Data Step. The reason: the match-merge of the SAS Data Step cannot use the index to retrieve the records in either data set while Proc SQL can.

For Example 11, this means that the joining of Datamart.Measures and Particular.Cities could execute much faster than a match-merge of the same tables if either Datamart.Measures is indexed or both tables are indexed.

Creating an Index is SAS is easy to code. There are several methods. Below is the Proc Datasets approach.

```
Proc Datasets library = Datamart;
   Modify Measures;
   Index Create City;
   Quit;
Run;
```

No matter the approach used, SAS creates the index in the same way: it determines the locations of each record on the primary data set that match a particular value of the indexed variable and then builds another data set that has the locations of those values on the primary data set. This ‘indexed’ data set is accessed first whenever the indexed variable is used to retrieve rows from the primary data set.

Using an Index does create additional overhead, however, even though it can speed selection of a single record by a factor of 10 to 20. Building an index requires additional processing time to create the index and additional space to store it. Also, the use of the index adds processing time for each record retrieved. Thus, indexes improve the overall speed of reading a table only if their use avoids the retrieval of the majority of the records in the data set. Creating and using an index is only warranted if the usual process is to pull subsets of the data set and not the full data set. The general rule is that the index should never be used to pull more than 20% of the table.

Also, as noted earlier, Proc SQL isn't the only way to invoke the use the index, but it is the easiest way to use an index. Where statements will use an index, but the value(s) being retrieved must be specified. Another option is using the Set statement with the Key= option, but that method requires special handling of the automatic _IORC_ variable when the value passed to the index is not found. The next Data Step code uses the key=option to pull rows from the Datamart.Measures table that match the cities in the table Particular.Cities.

```
Data Subset;
   Set Work.ParticularCities;
   Set Datamart.Measures key=City;
   If _iorc_ = 0 then
      Found = 'Yes';
   Else
      Do;
          _error_ =  0;
          Found   = 'No';
      End;
  Output;
Run;
```

Final Words

This paper has attempted to provide an understanding of the functionality of Proc SQL by comparing and contrasting Proc SQL code with typical SAS code that would perform the same function. It has also contrasted the SQL code to other SQL code to both reinforce the specific functions of the code and to indicate which coding techniques could be more effective.

It is hoped that this paper compliments, rather than repeats or rewords, recent SUGI papers on Proc SQL or the *SAS® Guide to the SQL Procedure*. The Guide and those papers that were read (and appreciated) by the author are listed in the references. All are available, thanks to SAS, under the SAS support website: http://support.sas.com/index.html

References

SAS® Guide to the SQL Procedure, Usage and Reference, Version 6, First Edition, SAS Insitute, Cary, NC, USA; 1990

Harrington, Timothy J., 'An Introduction to SAS® Proc SQL', *Proceedings of the SAS Users Group International 27, 2002,* <http://www2.sas.com/proceedings/sugi27/p070-27.pdf>

Lafler, Kirk Paul., 'Undocumented and Hard-to-Find SQL Features', *Proceedings of the SAS Users Group International 28, 2003,* <http://www2.sas.com/proceedings/sugi28/019-28.pdf>

Foley, Malachy J, 'Merging vs Joining: Comparing the DATA Step with SQL', *Proceedings of the SAS Users Group International 30, 2005,* <http://www2.sas.com/proceedings/sugi30/249-30.pdf>

Sherman, Paul D., ‘Creating Efficient SQL – Four Steps to a Quick Query’, *Proceedings of the SAS Users Group International 27, 2002,* <http://www2.sas.com/proceedings/sugi27/p073-27.pdf>

Whitlock, Ian., ‘Proc SQL – Is it a Required Tool for Good SAS® Programming?’, *Proceedings of the SAS Users Group International 26, 2001,* <http://www2.sas.com/proceedings/sugi26/p060-26.pdf>

Winn, Tom, ‘Intermediate PROC SQL’, *Proceedings of the SAS Users Group International 23, 1999,* <http://www2.sas.com/proceedings/sugi23/Advtutor/p35.pdf>

Trademark Citation

Contact Information

The author, Jimmy DeFoor, can be contacted at 972-652-4392 and jimmy.a.defoor@citigroup.com