Proc SQL – A Primer for SAS Programmers

Jimmy DeFoor Citi Card Irving, Texas

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 well the complete syntax and the full 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 DB2, T-SQL, PL-SQL and other SQL code.

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 first example involves creating a simple output listing in Proc SQL vs Proc Print.

Creating an Output Listing with Proc SQL vs a Data Step

<pre>Filename out `C':\temp.txt' new;</pre>	Filename out 'C':\temp.txt' new;
/* assign output to a file */	<pre>/* assign output to a file */</pre>
Proc Printo print=out;	Proc Printo print=out;
Run;	Run;
<pre>/* print contents of variables */</pre>	<pre>/* print contents of variables */</pre>
Proc SQL;	<pre>Proc Print data = Datamart.Measures;</pre>
Select montonic() as obs,	Label City = 'Town';
a.State, a.City as Town,	Var State City Store
a.Store, a.Year,	Year Month Sales
a.Month, a.Sales,	VarCost FixedCost;
a.VarCost, a.FixedCost	Run;
from Datamart.Measures a;	
quit;	

Since Proc SQL is a procedure, it requires a Proc Printto to be used before it is 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. Only by adding the little documented function of monotonic can observation numbers be generated. Notice the 'as' reference that follows the function. An 'as' reference uses the name that follows to name the result of the function. It can also be used to rename variables: here, it is used to rename City to Town. A Label statement in the Proc Print example accomplishes the same result. Compare the syntax of the Proc Print to the 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.

The variables listed in Proc SQL can be defined relative to the data set in which they occur. The programmer can create an alias by following the table name with a shorter name ('a', in the examples below). 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 a later example when a data set is created by joining two data sets.

Our next example shows the simplest way of creating a SAS data set: reading into it all the variables and records of another SAS data set. Notice that the Select * statement is used to specify that all variables are to be retrieved 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.

Proc SQL;	Data Work1;
Create Table Work1 as	Set Datamart.Measures;
Select *	Output;
from Datamart.Measures a;	Run;
guit;	

Creating a Table with Proc SQL compared to Creating It with a Data Step, Example 1

The syntax of the SAS Data Step states that it creates a data set, while the syntax of Proc SQL states that it creates a table, but SAS treats them as equals. The SAS Data Step can read a Proc SQL table and Proc SQL can read a SAS Data Set. Both entities are called tables by SAS when viewed in its explorer window (when details is tuned on).

The example that follows has the Select statement retrieving particular variables from the Measures data set and writing them to the table Sales. As discussed earlier, the Data Step must accomplish the same result with a keep action - in this case, 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. Keep, in any form, does not specify order. Only Length statements or Format statements set order 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.

On the other hand, though the Data Step cannot match the ease of variable ordering that can be done in Proc SQL, it can perform multiple record outputs to the same data set or record outputs to multiple data sets with great ease. For example, the data statement below 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);

Creating a Table with Proc SQL compared to Creating It with a Data Step, Example 2

Proc SQL;	Data Sales;
Create Table Sales as	Length State \$12 City \$20
Select	Store \$06 Year \$04
a.State, a.City,	Month \$02 Sales 8;
a.Store, a.Year,	Set Datamart.Measures
a.Month, a.Sales	(keep= State City Store
from	Year Month Sales);
work.Datamart.Measures a;	Output;
quit;	Run;

The Match-Merge Syntax of Proc SQL Compared to the Data Step

Proc SQL;	Data Joined;
Create Table Joined as	Merge Particular.Cities
Select	(in=c keep=City)
a.City, b.State,	Datamart.Measures
b.Store, b.Year,	(in=m
b.Month, b.Sales,	keep=State Store Year
b.VarCost, b.FixedCost	Month Sales
from	VarCost FixedCost)
work.ParticularCities a	By city;
left outer join	If c then
Datamart.Measures b	output;
On a.City = b.City;	run;
auit:	

These SAS code match-merges a list of selected cities with the Measures data set and keeps only the cities in Measures that match the cities in ParticularCities. The SQL code does the same thing, but the action is called an *equijoin* instead of a match-merge. This purpose of both sets of code is to create a data set that has sales and cost values for only particular cities. The same result could have been achieved in each example by using a Where statement that named each city, but that would have required more coding and, potentially, more errors.

Notice, however, that the SQL code and Data Step do more than just keep the records in Measures that match the cities from ParticularCities. They also keep the cities that aren't found in Measures in order that the user can know if sales and cost values were not found for a particular city.

The SQL code does this because of the left outer join clause. The Data Step does it because the If statement will output a record only if city in that record was also on Particular.Cities. It use the special variable, C, that has a value of 1 only when the value of City comes from that data set.

Below are examples of the coding needed to obtain particular results from a match-merge or an equijoin. The desired result is shown under the heading of function. Notice that the Match-Merge code references the special variable of M when records from Measures are being output.

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

Comparison of SQL Equijoins to Data Step Match-Merges

Based upon the coding above, Proc SQL and the SAS Data Step join records on a matching variable with almost equal ease of coding. They also join such records with almost equal efficiency, as long as the right outer data set is not indexed.

Index the right data set, however, and Proc SQL executes an inner join with far more speed than does the SAS Data Step. The reason: the matchmerge of the SAS Data Step will not use the index to retrieve the records on the right (second) data set while Proc SQL will.

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;
```

No matter the method of creation, the index is created in the same way. A data set is built that has the locations of each record matching a particular value of the indexed variable. That data set is accessed first whenever the indexed variable is used to retrieve particular values of the indexed variable. Where statements that select particular values of the indexed variable will likely use the index, as will sorts on the indexed variable, as will SQL joins. In most cases, the SAS programmer needs do nothing special to cause SAS to use the index. If the data set is indexed on that variable, SAS will evaluate whether the index will save processing time and use it if it will. To make certain, however, that SAS even considers using the index in Proc SQL, however, the indexed data set must be placed last in the join.

```
Select *
from
work.ParticularCities a
inner join
Datamart.Measures b
On a.City = b.City;
```

If both data sets are indexed by the same variable, then SAS may use both indexes in the join, but only if number of records in the first data set are numerous enough to warrant using the index. In that case, it will join the indexes and then join the records from each data set. Indexes can speed selection of particular records by a factor of 10 to 20, but they require additional processing time to create and additional storage space to store. In addition, each record retrieved via the index takes more processing time than retrieving the same record directly from the data set. Thus, indexes improve the overall speed of accessing data only when they can be used to avoid reading the majority of the records in the data set. Creating and using an index is, therefore, only warranted if normal process is to pull only subsets of the data set and not the full data set.

Because a data set has been indexed, however, the programmer doesn't have to use Proc SQL in order to use the index. Besides coding Where statements with the values desired of the indexed variable, the programmer can use the Set statement with the Key= option to pull records that match a list of values. The code below pulls cities from the Datamart Measures data set that match the values in 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;
```

But this approach requires special handling using the automatic _IORC_ variable when the value passed to the index is not found. Proc SQL makes no such demand on the programmer.

Finally, it should be noted that this discussion of match-merges and equijoins has assumed that the data sets were sorted by the variable being matchmerged and that the match-merge involved oneto-one matching or one-to-many matching. That is, it has assumed that the at least the first data set in the merge had unique values and that both data sets were sorted. If either of those assumptions is incorrect, then a match-merge will either fail or it will create an incorrect result. Proc SQL will not fail, however, because it will internally sort the data sets before merging and because it can do many-to-many matching.

Simple Sort, Computation, Assignment, and Selection

```
Proc SOL;
                                         Proc Sort Data = Joined
   Create Table Stats as
                                              (where=(State='TX')
                                                    Out = Subset;
     Select
       a.City, a.State, a.Store,
                                           By City Store Year Month;
       a.Year,
                a.Month, a.Varcost,
                                         Run;
       sum(a.VarCost, a.FixedCost) as
                                         *;
        TotCost,
                                         Data Stats;
     from
                                           Set Joined
       work.Joined
                                              (Keep=State City Store Year
        Where state = 'TX'
                                                    Month VarCost TotCost);
   Order by City, Store, Year, Month
                                           By City Store Year Month;
                                           Totcost =
quit;
                                                 sum(Varcost, FixedCost);
                                         Run;
```

The above examples perform four actions:

- 1. Selects only particular variables such as City, Store, Year, and Month.
- 2. Retrieves values for those variables from only the state of 'TX'.
- 3. Orders the results by City, Store, etc..
- 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.

<u>SQL</u>

1.30 * Sum(a.VarCost, a.FixedCost) as Price Year||Month as Yearmo Substr(Store,1,3) as StoreArea

Data Step

Price = 1.30 * Sum(a.VarCost, a.FixedCost); Yearmo = Year||Month; Storearea = Substr(Store,1,3);

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 the next example builds on this example and it requires that the sort be done before the summing.

Proc SOL; Data Stats; Create Table Stats as Drop FixedCost SumVarCost SumTotCost; Set Joined(where=(State='TX') select State, City, Year, Month, keep=State City Year Month sum(VarCost) as VarCost, VarCost FixedCost); sum(TotCost) as TotCost By State City Year Month; If first.month then from (select SumTotcost = 0;Totcost = Sum(Varcost, FixedCost); a.State, a.City, a.Year, a.Month, a.VarCost, SumTotCost + TotCost; sum(a.VarCost, a.FixedCost) SumVarCost + VarCost; as TotCost If last.month then from work.Joined a do; Where state = 'TX') TotCost = SumTotCost; group by State, City, Year, Month VarCost = SumVarCost; order by State, City, Year, Month; output; quit; end; run;

Summarization Combined with Computation, Assignment, and Selection

The above example repeats the actions taken in the previous example, but also uses sum statements or functions to create totals by State and City. Accordingly, the Store variable is dropped. Now the reason for sorting the data before executing the Data Step becomes obvious. The Data Step could not have summed the TotCost and VarCost fields by City, Year, and Month without the preceding sort. The SQL code, however, can sum without a Proc Sort because it will internally sort the data before it calculates a sum for each grouping of City, Year, and Month. This happens because both a Group By and an Order By are present. The Order By sorts and the Group By indicates that only the summary values will output for each grouping. Had the Order By not been present, the SQL code would still have summed for each grouping, but it would not have sorted the data before the summing by group. Instead, it would have just ordered each group as it encountered it, which could have resulted in a data order that was not sorted.

The Data Step uses the By variable statement to establish the first.month and last.month variables that mark the first and last occurrence of each By grouping. It then uses sum statements to calculate the sum for each grouping. Then, it writes out those sums under the original names of the variables via the last.month reference.

The SQL step accomplishes the same result with an in-line view, which – essentially – is a SQL step within a SQL step. Notice that the Select statement that sums over the Group By retrieves data from another Select statement and not a Table. That statement retrieves Texas cities and stores from the data set Joined and then creates the TotCost variable. Thereafter, it passes its results to the next Select statement, which sums VarCost and TotCost over the Order By variables and then writes the result into the variables VarCost and TotCost.

The use of an in-line view wasn't required. The same result could have been achieved this way:

```
Proc SQL;
Create Table Stats as
Select State, City, Year, Month,
sum(VarCost) as VarCost,
sum(sum(a.VarCost, a.FixedCost))
as TotCost
from work.Joined a
Where state = 'TX')
group by State, City, Year, Month
order by State, City, Year, Month;
```

But in-line views can be very useful in 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 this case. The SQL code in this last example works because SQL knows *by context* that it should for 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 easily and what SQL understands easily may not always be the same.

Two final notes should be made before going onto the next subject. 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 added to both sets of code.

SAS Data Step

```
If last.month then
   do;
    TotCost = SumTotCost;
    VarCost = SumVarCost;
    ExpRev = Totcost * 1.30;
    output;
end;
```

SQL Code

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

The key point to notice is that the SQL code applied the 1.30 rate to the same math calculation for TotCost as was used in the creation of the value of TotCost. Normal SQL code cannot, within the same view, use the value of TotCost to calculate ExpRev. This is because TotCost does not exist until SQL completes that view. Until then, that column is an address space that only SQL can address. TotCost can be accessed in the Data Step because it was 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 can avoid this issue with the Calculated reference, however. Adding Calculated in front of a column will make the SQL processor create a reference to the result that can be employed in the same view. In this case, the above code would be changed to:

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

Remerging calculations onto the same data set

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 remerged 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.

The Proc Summary above computes the average of VarCost and then that average is added to each row of the original data with a second Set statement that was executed only once. In that way, the average value is retained for all subsequent rows of read from the original data set Joined.

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 to back onto Joined. This joining caused SQL to issue a warning that a Cartesian join was performed, which is when all rows of the second data set are joined to each row of the first data set. This is exactly what was wanted. The single row of the in-line view of Joined, which has the average of VarCost was joined to each row of Joined. This isn't the usual coding approach by which remerging is done in SQL, however, but it is executed in the same way as the usual approach and it demonstrates the underlying mechanics of standard method of remerging. The typical SQL coding for remerging is as follows:

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

SQL recognizes that the above coding requires that the average be merged with all records, so it does so and issues a warning that the results of its calculation are being remerged with the original data set, which – by the way – is a Cartesian join. In fact, the CPU time needed to average the values for 1 million records and merge that average back onto the million records is very similar for the two SQL coding methods. The inline view is slightly faster than the remerge, but it is just a bit slower than the combined time of the Proc Summary and the Data Step.

The next examples will cover sub-queries and 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 have resulted from the summary of a group of records, perhaps even all the records in the table. Thus, the Having clause is either used with an Group By clause or a function that calculates a mean, a sum, or some other measure over all records of the table.

Using a Having Clause to evaluate records summed with a Group By Clause

Proc SQL;	Proc summary data = Joined nway;
Create Table Stats as	Class City State Year Month;
select	Var VarCost;
a.City, a.State,	Output out = summary
a.Year, a.Month, a.VarCost,	Mean = AvgVarCost;
mean(a.VarCost) as AvgVarCost	Run;
from	*;
work.joined a	Data Stats;
having Varcost gt AvgVarCost	Merge Joined
Group by City, State, Year, Month	<pre>Summary(drop= type freq);</pre>
Order by City, State, Year, Month	By City State Year Month;
quit;	If VarCost gt AvgVarCost then
	Output;
	Run•

The above example shows a Having clause selecting records from grouping of City, State, Year, and Month that have a variable cost greater than AvgVarCost. The mean function calculates the average and, then, the Having clause outputs only the City and State values of VarCost that exceed the average. This happens during the remerging of mean value with the Joined table. As stated earlier, it is the Group By clause that causes the calculation of the average 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 second example above shows SAS code accomplishing the same result with a Proc Summary and a match-merge by City, State, Year, and Month.

Using a Having Clause to evaluate records summed over all records

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

On the left above is another example of using a Having clause to control the records that will output to the Stats table. In this case, only the

records with the lowest variable costs are passed to Stats table.

As stated above, the Having clause is executed after the calculation of the measure, which means it is executed during the remerge. Since the remerge, in truth, is a separate view from the calculation, the Having clause can reference either the variable that is created from the calculation, as in the earlier example, or the function that creates the measure, as shown here. This means that the Having clause does not ever need to use the Calculated reference described earlier. It also means that the Having clause is the equivalent of a Where clause from the standpoint of the outer view. This is shown by the second example, which accomplishes the same result with an in-line view.

This in-line view executes somewhat more efficiently than the remerge as the number of observations increase, just as it did in an earlier example. The inline view used 1 less CPU second for 100,000 observations and 19 fewer CPU seconds for 1 million observations.

This application of in-line view with a Where, or a Having clause, is called a subquery. Subqueries can return one value, as it does here, or multiple values. To return multiple values, the in-line Select must be structured to return multiple values and the Where or Having clause must have an operator that deals with multiple values, such as In or Exists. The In operator is shown in the example below and is the preferred method since it is more efficient.

```
Proc SQL;
Create Table Stats as
Select
City, State,
Year, Month, VarCost
from
work.joined a
where
a.VarCost in
(Select Varcost from
work.joined b
having Varcost gt
(.125 * avg(varcost)))
;
quit;
```

The subquery retrieves a list of values of variable costs from Joined that are more than 25% larger than the average variable costs of all Cities. The Where clause then selects only the Cities from Joined that have variable costs that exceed the average variable cost by 25%. When it executes, the Where clause looks something like this:

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

The last examples in this paper cover Case expressions and user-built formats, which are used to assign values to a variable based upon the values of other variables.

```
Proc SOL;
  Create Table Stats as
    Select
      City, State,
      Year, Month, VarCost,
      Case
        When Year 1t '2003' then
             'PreviousYear'
        Else
            'CurrentYear'
      end
        as Descrptn
          from
             work.joined a
      ;
  quit;
```

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 the field that will be named Descprtn when the results of the Select are written to the Stats table.

The Case expression is closed with an End reference. The Else references are recommended, but optional. SQL will assign missing values in their absence, just as would the If Then Else statements a Data Step if a 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'
```

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

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

Assuming the user-built format looked something like below, the put function 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;
```

Final Words

This paper has attempted to provide an understanding of the simpler capabilities of Proc SQL by comparing and contrasting Proc SQL code with typical SAS code that would perform the same function as the SQL code being presented. It has also contrasted the SQL code to other SQL to both reinforce the specific functions of the code and to indicate which coding techniques could be more efficient when processing larger numbers of records.

It is hoped that this paper compliments, rather than repeats or rewords, recent SUGI papers on Proc SQL or the <u>SAS Guide to the SQL</u> <u>Procedure</u>. 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', SAS Users Group International (SUGI) 27, 2002

Lafler, Kirk Paul., 'Undocumented and Hard-to-Find SQL Features', SAS Users Group International (SUGI) 28, 2003

Sherman, Paul D., 'Creating Efficient SQL – Four Steps to a Quick Query', SAS Users Group International (SUGI) 27, 2002

Whitlock, Ian., 'Proc SQL – Is it a Required Tool for Good SAS® Programming?', SAS Users Group International (SUGI) 26, 2001

Trademark Citation

SAS and all other products and service names of the SAS Institute Inc. are registered trademarks of SAS Institute in the USA and other countries. ® indicates registration.

Contact Information

The author, Jimmy DeFoor, can be contacted at

972-653-5928, and jimmy.a.defoor@citigroup.com