Old Wine, New Wineskins: Why the SQL Procedure Will Change the Way You Think

Peter Kretzman, McCaw Cellular Communications, Inc.

ABSTRACT

Release 6.06 of the SAS System is NOT just a new release. Instead, in the increasingly diverse world of software development using the SAS System, Release 6.06 dramatically ups the ante. Especially considered in tandem with the new ability to index SAS data sets, PROC SQL immediately jumps into the category of Mandatory Things to Know for the advanced SAS programmer.

This advanced tutorial compares and contrasts some standard SAS tools (most notably, MERGE) with equivalent ways to achieve the same effects using PROC SQL. Astounding improvements in performance can often be achieved by using PROC SQL instead of more traditional SAS tools.

Concrete examples are provided, along with discussion of efficiency considerations. After this tutorial, your way of thinking about a SAS solution to a data combination problem will simply not be the same.

This tutorial is intended primarily for relatively experienced SAS programmers who are interested in the latest tools and techniques for improving their productivity. No previous knowledge of SQL is assumed.

INTRODUCTION

Over the years, numerous “mini-languages” have been added or grafted on to the basic SAS System: the macro language, Screen Control Language (SCL), the Interactive Matrix Language (IML), and, most recently, the Structured Query Language (SQL). Each of these mini-languages exists for a reason; each, unfortunately, has its own rules, syntax, and set of “gotchas” for the hapless SAS programmer. The sheer number of these additional “mini-languages” makes it fairly difficult for anyone anymore simply to claim that they “know” SAS; instead, one “knows” the base SAS product, or “knows” each of these subsystems or sublanguages individually. Few people (myself included) know all of the mini-languages now available under the SAS System.

Given the number of these mini-languages, the burden of proof for learning any or them has essentially been thrown on the SAS System itself. Users have the right (and the need) to ask: what are the tangible benefits of learning a particular subsystem? What can I do using this subsystem that will vastly increase my productive use of the SAS System overall? These issues are the chief thing to consider with respect to the latest kid on the block, PROC SQL.

Perhaps more than with any of the other mini-languages within the SAS System, reasons abound to take a solid look at what PROC SQL can do for you. Unlike the others (with the exception of the macro facility), PROC SQL is included with the base SAS System, and is thus available to all SAS users on all platforms as of Release 6.06. And, of course, the reason it is included with the base product is that, like the macro facility and unlike a specialized subsystem such as SAS/IML, it is a general-purpose tool, one which can be used to advantage by every SAS programmer.

Secondly, PROC SQL, by embodying the emerging Structured Query Language standard, represents a standardized way to formulate a query on (in other words, to retrieve data from) existing data. Given the availability of SQL within the SAS System, programmers can now migrate quickly from any of the numerous database management packages that also employ the SQL standard as a query language. Although each of these packages contains extensions to the SQL standard (and the SAS System is no exception), there is at least a common ground now among them. Most importantly, the availability of SQL in all these packages means that their metaphor of data representation is common across them: a relational design featuring tables with rows and columns. Aside from the use of SQL within the SAS System itself, this commonality means it is considerably easier to conceptualize, and to code, access to data across database platforms, using the SAS/ACCESS tools. Since the SAS System can ship SQL queries directly to external DBMS packages, it requires no break in stride for the programmer to code a piece of SQL for internal SAS System use, combined in the same routines as SQL code that is intended to be stripped elsewhere at runtime.

The argument for using PROC SQL that will doubtlessly have the greatest weight with current SAS System users, however, is pure and simple efficiency. PROC SQL provides, at the very least, an alternative (and more efficient) way of achieving the results of one of the most common tools in the SAS System: a simple match-merge. In many circumstances, PROC SQL can provide data combination results that would be extremely difficult to obtain using conventional SAS System tools.

This introductory paper is designed to whet your appetite, and to make you realize the kinds of situations where you might use PROC SQL to great advantage. It is NOT intended to be a thorough investigation into the vast array of data combination possibilities using SQL constructs.

The style of this paper will be to reveal SQL constructs incrementally, rather than attempt to show them all at once in a kind of catalog format. However, a basic template of a “join,” or what SAS users refer to as a “match-merge,” is a useful starting point for most of the examples in this paper.

SQL SYNTAX: THE BASICS

The style of this paper will be to reveal SQL constructs incrementally, rather than attempt to show them all at once in a kind of catalog format. However, a basic template of a “join,” or what SAS users refer to as a “match-merge,” is a useful starting point for most of the examples in this paper.
PROC SQL FEEDBACK NOPRINT;
CREATE TABLE newtable AS
SELECT *
FROM data.master AS m,
data.lookup AS l
WHERE m.key = l.key
ORDER BY idnumber;

The above SQL code is a basic construct that shows how to combine, for example, a large table (data.master) with a smaller lookup table (data.lookup), using the equivalent of a BY variable, in this case named key. The equivalent conventional SAS code would look as follows:

DATA newtable;
  MERGE data.master(in=m) data.lookup(in=l);
  BY key;
  IF m AND l;
RUN;

Let's go through the SQL construct now and discuss its implications. First, the procedure is invoked with a simple PROC SQL call: think of this as the gateway into the mini-language. It turns control over to the SQL processor, away from the conventional SAS DATA step and PROC steps. Second, SQL is a query language, intended at least partially to provide printed results from each query. It turns out, of course, that much of the time you will want to use SQL just for its combinatorial capabilities, rather than to obtain a printed list of its results (this is especially true when you are combining thousands or millions of observations!). Hence, more often than not, I find myself using the NOPRINT option to turn off the printing of the results. In the above example, the FEEDBACK option is also used. This provides extra information at runtime in the SAS log, pertaining to how the query was reformulated by the SQL query optimizer, and I find it useful to examine.

Note, in the above SQL code, that the syntax is quite different from conventional SAS System syntax. Most notably, only one semi-colon is contained in the query, at the very end. Lists of things (variables, data sets, etc.) have commas separating the things: in the above example, commas separate the names of the two data sets that will be joined. Finally, although the two data sets specified above contain periods in their names (in the conventional form of database.name), the periods in the WHERE clause are something else indeed, and are discussed further below.

Ignore the CREATE TABLE statement for now. The meat of most SQL code is one or more SELECT clauses. A SELECT clause has two parts, one where you specify what columns you want and the second where you specify what data sets will be used. In the above example, we have specified an asterisk for a variable list, meaning that we want ALL variables from each of the data sets. (NOTE: PROC SQL takes that quite literally; if you are obtaining printed output, for example, and a variable of the same name appears in more than one of the data sets named in the FROM portion of the SELECT clause, you will get a separate column for each of those variables, labeled identically. SAS output data sets, of course, can contain only one appearance of any given variable).

After the variable list is specified, the keyword FROM appears, followed by a list of one or more data set names, separated by commas. Each of these data sets may be assigned an arbitrary alias by using the keyword AS, followed by the desired alias. In our example, we have assigned single letter aliases to each of the two data sets. Assignment of aliases is typically done so that column names that match in the two data sets may be "qualified," (that is, stated unambiguously) during operations such as processing of the WHERE clause.

The Tricky Part

Now here is the tricky part. Logically speaking (and it is important to remember that this does NOT happen physically), the SQL procedure builds a table that combines each row from the first table with every row from the second table. In short, if nothing further were specified in the query, your result would be every possible combination of the rows. This "all combination" result is what is referred to as the Cartesian product of the two tables. If your first table has eight rows and your second table has three rows, the Cartesian product would contain 24 rows: the first row of the first table combined, in turn, with the first, second, then third rows of the second table, then the second row of the first table combined with the first, second, then third rows of the second table, and so on.

It's actually fairly seldom, of course, that you really want the Cartesian product of any two tables. The whole concept of a match-merge, for example, is built upon the word "match." That means you only care about cases where the key variable agrees in the two tables you are joining. What this means is Cartesian product terms is that you only want to keep the rows (in that product) where the variable key (in this case, the variable named key) matches. This match is specified in the WHERE clause of the SQL query; our example features the equivalent of the match-merge, known as an "equijoin" since the key variables must be equal for the row to be kept in the output table.

Typically, you "qualify" the references to the key variable by the alias for one table, equating it to the qualified name of the key variable for the other table. Typically, there are (N-1) conditions in the WHERE clause for N tables being joined together. Since we are logically proceeding from a table (the Cartesian product, that is) that features every possible combination of rows, it is not necessary (unlike with a match-merge) that the two input tables data.master and data.lookup be sorted on the key variable. Note, however, that the use of periods in the WHERE clause is specific to the SQL procedure, and represents use of the "aliasing" explained above.

Finally, since the input tables are not necessarily sorted, the results of the query are not in any particular guaranteed order. For that reason, you may specify an ORDER BY clause to the query, which serves to place them in the order you desire. Note that the variable specified in the ORDER BY clause does not have to be the same variable on which the equijoin was performed. In general, PROC SQL gives you a great deal of freedom in variable selection: variables listed in your WHERE clause, for example, do not need to be in the variable list that appears immediately after the SELECT statement.

272
Piquing Your Interest

Now that we have gotten through the rudiments of the syntax involved in an equijoin, let's look at some timings and comparisons, stacking up a conventional match-merge against the equivalent SQL construct. Keep in mind that this is only an example, and is not intended by any means to show that SQL is always better than a conventional match-merge. Depending on your situation, actual needs and results may vary considerably. All timings shown are from running the code on a Northgate 486/25 PC, using the SAS System under OS/2 (beta test version).

First, here is the standard match-merge. In this case, RAW01 is a data set containing approximately 501,000 observations, INTEREST is a data set containing approximately 55 occurrences of the key variable SYMBOL. Please observe the recorded run times carefully.

```
DATA MONTH1;
   MERGE RAW01 {in=raw}
      INTEREST {in=wanted};
   BY SYMBOL;
   IF RAW AND WANTED;
RUN;
```

Time: 1 minute, 44.82 seconds (1,173 obs. extracted)

Hmmm. Not bad for a PC: processing over 100,000 observations in only 105 seconds. Now let's try it with PROC SQL:

```
PROC SQL NOPRINT;
   CREATE TABLE result1 AS
      SELECT *
      FROM raw01 AS a,
      INTEREST AS w
      WHERE a.symbol=w.symbol
   QUIT;
```

Time: 8 seconds (1,173 obs. extracted). NOT a typo!

Hmmm. Hmmm. 8 seconds compared to 105. How would you feel about driving a car that got you 200 miles to the gallon? At least ostensibly, it appears that PROC SQL, in this instance, can give you an order of magnitude better performance over a standard match-merge.

Not so fast, buster

But let's slow down a bit and run a few more tests before we get too excited. Here's a couple more examples with slightly different results:

```
DATA MONTH1;
   MERGE RAW01 {in=raw}
      INTEREST {in=wanted};
   BY SYMBOL;
   IF RAW AND WANTED;
RUN;
```

Time: 3 minutes, 24.44 seconds (1,173 obs. extracted).

Whaaaaat? This is the exact same code! Where did all our time savings go? In fact, this result is now considerably worse than before. The CPU time was greater (444.25 seconds) than the match-merge of one minute, 44 seconds. Well, it turns out that this most recent example represents the same job run without an index having been created on the master data set (in this case, raw01). The index, then, allows the SQL query optimizer to be considerably more intelligent about how it processes the query. Without that index, it simply has to sort both input data sets through the scenes, at great cost in both CPU time and in disk resources, and proceed to slog through the data.

So no problem, right? Just create an index on all your data and you'll be in great shape, right? Wrong. Here's the match-merge example, the one that ran in one minute, 44 seconds in our previous result:

```
DATA MONTH1;
   MERGE RAW01 {in=raw}
      INTEREST {in=wanted};
   BY SYMBOL;
   IF RAW AND WANTED;
RUN;
```

Time: 1 minute, 3 seconds (1,173 obs. extracted)

Here we go again. This time, the code ran in only 63 seconds, not 105. So the last time was without an index, right? Wrong: in fact, just the opposite. In a conventional match-merge where the data is already sorted, the presence of an index actually slows things down. The reason for this is that the DATA step processor is not nearly as intelligent as the SQL query optimizer. It simply has to go through each and every observation in the large data set (raw01) and can only check the one that has been reduced by the WHERE clause. The reason for this is that when there is an index present, it goes out and uses it, but since it has to go through all the data in the large data set anyway, it doesn't help. In fact, it hinders, enough so that a 63 second job becomes 67% longer as a result.

Creating an Index

Now that you've seen the potential benefits of indexes, however, it would be helpful to know how to create them. There are two ways to create an index in the SAS System: use PROC DATASETS or use PROC SQL itself.
Here's a summary of U1e timings and the pitfalls, in the SAS System.

Again, it is hard to provide hard-and-fast rules of thumb now that the SAS System is so complex (with indexes, multiple platforms, etc.), but for the simple kinds of joins I have outlined here, it seems usually advantageous to use PROC SQL and combine it with the use of an index. If that same indexed data set is going to be used in conventional DATA steps, however, you may wish to delete the index. In future SAS releases, there may be a way to instruct the SAS System not to use an index in a particular step, but there is currently no such method available.

Remember as well that there is a considerable cost, depending on the nature of the data, to creating an index in the first place. This cost must be weighed against how often (and how) the data in question are going to be used.

Why Use SQL Views?

Moving away now from the topic of pure efficiency, the rest of this paper will deal with another major reason to use SQL: the ability to create SQL Views.

The SAS System, compared to most third-generation languages, has always had a weak point in its lack of ability to "bottle," as I like to put it. Other than with the macro language, there haven't been many constructs in the SAS System for designing something once, then using that something as a building block for other purposes. Most languages have user-written procedures; the SAS System does not, at least within its own framework. SQL now provides SQL views, which are a kind of "bottling" tool, not for user-written procedures per se, but for designing "recipes." Rather than storing a particular version of the data, it is now possible to store its "recipe," so to speak, or how that version may be constructed. At the time you need that version of the data, you may refer to it as if it actually exists, employing its name in SAS procedures and so on. At runtime, the SAS System will go out, process the recipe, and feed the resulting cake to the procedures that are referring to it.

Of course, there are pros and cons to having a recipe, especially at times when you really want the cake and you want the cake mighty fast. In short, the downside is that every time you want the data in question, the SAS System has to go out and construct it. But there are many advantages to keeping the views instead of many subsets or alternative versions of data. Users may want several different versions (literally, views) of the data, and keeping several physical sets of data in synchronization is notoriously difficult. With a view, the data are always current since data are obtained from a single source and manipulated into relevant forms. Finally, views can shield your users from the messy details involved in large and complex merges. To the user, they may not even be aware that a given "data set" is really a recipe and not a cake at all. Once the view is coded, you can be certain that all of its users will be performing the merge correctly, since it is being done for them; you can also be sure that the users are getting the most recent form of your primary data. And, of course, if you want to change where and how you have implemented the physical form of your data base, it is a good deal easier to change a view (which is, after all, the public view of the data) than to change the data structures themselves.

Invisible Mortar

SQL Views, as other words, are a way of providing users with "invisible mortar," by which bricks are slapped together and passed on together, transparently to the view's user. See my SUGI 13 paper (Krotzman, 1980) on the "Mortar and Brick" theme in general.

Let's look at a typical view constructed with PROC SQL. As with our other examples, it represents the equivalent of a match-merge.

```sas
PROC SQL NOPRINT;
CREATE VIEW thismon AS
SELECT a.symbol, date, high, low, close, volume
FROM raw01 AS a,
     interest AS w
WHERE a.symbol=w.symbol
QUIT;
```

You will note that this is virtually the exact same code as I gave in the timing examples earlier in this paper. The key difference, however, is that CREATE VIEW is being used, rather than CREATE TABLE. In short, I'm telling PROC SQL to build and file a recipe here, not to construct the actual cake. The recipe, in this instance, will be a "work" (not permanent) view called thismon.

Let's now examine a case where we use this view to do some work. Note how the reference to thismon appears in a place where an ordinary data set reference could appear. This example also happens to be using more SQL, but it wouldn't have to be. The key point about views is that they may be used anywhere you could refer to a SAS data set (with the exception that they may not be written to in the current release of the SAS System).

The point of this particular query, aside from its use of an SQL view, is to obtain a set of stock symbols, ordered by descending average closing price in the current month. The purpose of the
view is to obtain only those stocks that are of interest, as contained in a SAS data set of key symbols.

```sql
PROC SQL FEEDBACK;
SELECT symbol,
    mean(close) AS avgclose
FROM thismon
GROUP BY symbol
ORDER BY avgclose DESC;
QUIT;
```

First, before we discuss the role of views in resolving this query, let's examine the additional SQL features that emerge in this code. First, note that we have specified neither CREATE TABLE or CREATE VIEW. That means that our output will be printed only. Secondly, we have instructed PROC SQL to compute a new variable, which we have told it to call avgclose, the value of which will be the mean of the variable called close. The GROUP BY clause tells PROC SQL over what groups the mean will be calculated (in short, we want one mean for each value of the variable symbol). Finally, we would like our results sorted in descending order of the newly computed variable.

Imagine doing this same operation using conventional SAS code. You would need a PROC SUMMARY, then a PROC SORT, then a PROC PRINT. SAS combines all of those things into one.

Note once again that there are syntactical anomalies to watch out for. The "DESC" option on the ORDER BY clause, for example, appears after the name of the variable to which it applies, unlike how it appears on an ordinary SAS BY statement. It is also abbreviated differently.

The FEEDBACK option on the PROC SQL statement itself instructs the PROC SQL processor to show us how it is expanding the query. In short, how it is resolving the recipe that it has stored in the SQL view. Here is the result of that resolution:

```sql
SELECT raw.symbol,
    AVG(raw.close) AS avgclose
FROM (SELECT raw.symbol, raw.date, raw.high, raw.close
    FROM WORK.RAW01 RAW, WORK.INTEREST WANTED
    WHERE raw.symbol=wanted.symbol)
GROUP BY raw.symbol
ORDER BY avgclose DESC;
```

Note how the reference in the code to thismon has been resolved to another SELECT clause, one which was obtained from the "recipe" specified in the view. The SQL processor has done some other "cosmetic" things; note that MEAN was changed to AVG, variables were qualified in every instance, and so on.

The results, as printed, of this query look like the following:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>AVGCLOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERICY</td>
<td>196.16</td>
</tr>
<tr>
<td>IBM</td>
<td>105.59</td>
</tr>
<tr>
<td>MOT</td>
<td>62.13</td>
</tr>
<tr>
<td>MSFT</td>
<td>59.40</td>
</tr>
<tr>
<td>GE</td>
<td>58.07</td>
</tr>
</tbody>
</table>

Additional Complexities

As the reader has no doubt surmised by now, the complexity in the SQL procedure far exceeds the scope of this one paper. Topics of interest that I have not covered include the following:

- Complex joins: outer joins, reflexive joins, etc.
- Subqueries (within WHERE clauses, etc.)
- Updating data sets using SQL
- Set operators (UNION, EXCEPT, INTERSECTION, etc.)
- Querying multiple tables
- Other arithmetic expressions in the SELECT clause

CONCLUSION

The SQL Procedure can and should be a vital tool in the arsenal of the accomplished SAS programmer. For performance reasons alone on basic constructs, it is worth taking the time to learn and to learn well.

Comments, suggestions, and questions are welcome. The author may be reached at the following address:

Peter Kretzman
McCaw Cellular Communications, Inc.
P.O. Box 90685
Kirkland, WA 98033-9765
Voice: (206)628-1344
FAX: (206)628-1300

In addition, the author may be contacted via Internet, using the address:

kretzman@uwam.scs.washington.edu.

ACKNOWLEDGEMENTS

The author wishes to thank the following people for their contributions (wittingly or not) to the thought that went into this paper: Tim Featham, Patricia Gerend, Gene Hart, and Paul Kent. In addition, numerous ideas were obtained from the SAS-L conference on BITNET.

BIBLIOGRAPHY


SAS, SAS/ACCESS and SAS/IML are registered trademarks of SAS Institute, Inc., Cary, NC.