Alternatives to Merging SAS Data Sets … But Be Careful

Michael J. Wieczkowski, IMS HEALTH, Plymouth Meeting, PA

Abstract

The MERGE statement in the SAS® programming language is a very useful tool in combining or bridging information from multiple SAS data sets. If you work with large data sets the MERGE statement can become cumbersome because it requires all input data sets to be sorted prior to the execution of the data step. This paper looks at two simple alternative methods in SAS that can be used to bridge information together. It also highlights caveats to each approach to help you avoid unwanted results.

PROC SQL can be very powerful but if you are not familiar with SQL or if you do not understand the dynamics of SQL "joins" it can seem intimidating. The first part of this paper explains to the non-SQL expert, the advantages of using PROC SQL in place of MERGE. It also highlights areas where PROC SQL novices can get into trouble and how they can recognize areas of concern.

The second section shows another technique for bridging information together with the help of PROC FORMAT. It uses the CNTLIN option in conjunction with the PUT function to replicate a merge. Caveats to this method are listed as well.

What's wrong with MERGE?

Nothing is wrong with using MERGE to bridge information together from multiple files. It is usually one of the first options that come to mind. The general technique is to read in the data sets, sort them by the variable you would like to merge on then execute the merge. Example 1A below shows a match merge between two data sets. One contains zip codes and states, the other contains zip codes and cities. The goal is to attach a city to each zip code and state.

Example 1A:

```
/* Read data set #1 – zip/state file */
DATA STATES;
  LENGTH STATE $ 2 ZIP $ 5;
  INPUT ZIP $ STATE $;
CARDS;
08000      NJ
10000      NY
19000      PA
RUN;

/* Sort by variable of choice */
PROC SORT DATA=STATES;
  BY ZIP;
RUN;

/* Read data set #2 – zip/city file */
DATA CITYS;
  LENGTH ZIP $ 5 CITY $ 15;
  INPUT ZIP $ CITY $;
 CARDS;
10000           NEWYORK
19000           PHILADELPHIA
90000           LOSANGELES
RUN;

/* Sort by variable of choice */
PROC SORT DATA=CITYS;
  BY ZIP;
RUN;

/* Merge by variable of choice */
DATA CITY_ST;
  MERGE STATES CITYS;
  BY ZIP;
RUN;

PROC PRINT DATA= CITY_ST;
RUN;
```

The following output is produced:

```
OBS   STATE    ZIP     CITY
  1     NJ     08000
  2     NY     10000   NEWYORK
  3     PA     19000   PHILADELPHIA
  4            90000   LOSANGELES
```

Notice above that the merged data set shows four observations with observation 1 having no city value and observation 4 having no state value. This is because not all of the records from the state file had a zip code in the city file and not every zip code in the city file had a matching zip code in the state file. The way the merge was defined in the data step allowed for these non-matches to be included in the output data set.

To only keep matched observations between the two files, the merge step can be altered using the **IN=** data set option where a variable is set to the value of 1 if the data set contains data for the current observation, i.e., a match in the BY variables. This is shown in the example below.

Example 1B:

```
DATA CITY_ST2;
  MERGE STATES (IN=A) CITYS (IN=B);
  BY ZIP;
  IF A AND B;
RUN;

PROC PRINT DATA= CITY_ST2;
RUN;
```

The following output is produced:

```
OBS   STATE    ZIP     CITY
  1     NY    10000   NEWYORK
  2     PA    19000   PHILADELPHIA
```

The two observations above were the only two situations where each data set in the MERGE statement contained common BY variables, ZIP=10000 and ZIP=19000. Specifying **IF A AND B** is telling SAS to keep observations that have by variables in both the STATES (a.k.a. **A**) and CITYS (a.k.a. **B**) files. Similarly you could include observations with ZIPs in one file but not in the other by saying either

IF A AND NOT B; *or*
IF B AND NOT A;

Introduction to PROC SQL

A similar result is possible using PROC SQL. Before showing an example of how this is done, a brief description of SQL and how it works is necessary. SQL (Structured Query Language) is a very popular language used to create, update and retrieve data in relational databases. Although this is a standardized language, there are many different flavors of SQL. While most of the different versions are nearly identical, each contains its own "dialect" which may not work the same from one platform to another. These intricacies are not an issue when using PROC SQL in SAS.

For the purpose of this paper using PROC SQL does not require being an expert in SQL or relational theory. The more you know about these topics the more powerful things you can do with PROC SQL. In my examples it is only required that you learn a few basic things about the SQL language and database concepts.

In SQL/database jargon we think of columns and tables where in SAS we refer to them as variables and data sets. Extracting data from a SAS data set is analogous, in SQL talk, to querying a table and instead of merging in SAS we perform "joins" in SQL.

The key to replicating a MERGE in PROC SQL lies in the SELECT statement. The SELECT statement defines the actual query. It tells which columns (variables) you wish to select and from which tables (data sets) you want them selected under certain satisfied conditions. The basic PROC SQL syntax is as follows:

```
PROC SQL NOPRINT;
  CREATE TABLE newtable AS
  SELECT col1, col2, col3
    FROM table1, table2
    WHERE some condition is satisfied;
```

The first line executes PROC SQL. The default for PROC SQL is to automatically print the entire query result. Use the NOPRINT option to suppress the printing. If you need the data from the query result in a data set for future processing you should use the CREATE TABLE option as shown on line 2 and specify the name of the new data set. Otherwise, the CREATE TABLE statement is not necessary as long as you want the output to print automatically.

Lines 2 through 5 are all considered part of the SELECT statement. Notice that the semicolon only appears at the very end. In the SELECT statement we define the variables we wish to extract separated by commas. If all variables are desired simply put an asterisk (*) after SELECT. Be careful if the data sets you are joining contain the same variable name. This will be explained more in the example to follow. The FROM clause tells which data sets to use in your query. The WHERE clause contains the conditions that determine which records are kept. The WHERE clause is not necessary for PROC SQL to work in general but as we are using PROC SQL to replicate a merge it will always be necessary. Also note that the RUN statement is not needed for PROC SQL.

The results from the merge in example 1B can be replicated with PROC SQL by eliminating the two PROC SORT steps, the MERGE data set step the PROC PRINT step and replacing it with the following code:

Example 2A:

```
PROC SQL;
  SELECT A.ZIP, A.STATE, B.CITY
  FROM STATES AS A, CITYS AS B
  WHERE A.ZIP=B.ZIP;
```

This results in:

```
STATE         ZIP       CITY
_______________________________________

NY            10000     NEWYORK
PA            19000     PHILADELPHIA
```

Notice that no CREATE TABLE line was used since all we did in our example was print the data set. If that's all that is needed then there is no use in creating a new data set just to print it if PROC SQL can do it. If desired, titles can be defined on the output the same as any other SAS procedure.

Also notice in the SELECT statement that each variable is preceded by A or B and a period. All variables in the SELECT statement can be expressed as a two-level name separated by a period. The first level, which is optional if the variable name selected is unique between tables, refers to the table or data set name. The second is the variable name itself. In the example, instead of using the table name in the first level I used a table alias. The table aliases get defined in the FROM clause in line 3. So the FROM clause not only defines which data sets to use in the query but in example 2A we also use it to define an alias for each data set. We are calling the STATES data set "A" and the CITYS data set "B" (the AS keyword is optional). Aliases are useful as a shorthand way of selecting variables in your query when several data sets contain the same variable name. In our example both data sets contain the variable ZIP. If we simply asked for the variable ZIP without specifying which data set to use, the print out would show both ZIP variables with no indication of which data set either one came from. If we used a CREATE clause without specifying which ZIP to use, an error would occur because SAS will not allow duplicate variable names on the same data set.

We could also code the SELECT statement several different ways, with and without two level names and aliases as shown below.

- *Two-level without aliases*

SELECT **STATES**.ZIP,**STATES**.STATE, **CITYS**.CITY

- *Two-level where necessary:*

SELECT **STATES**.ZIP, STATE, CITY

- *Two-level with alias where necessary:*

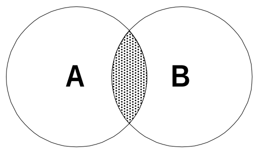
SELECT **A**.ZIP, STATE, CITY

All accomplish the same result. It's a matter of preference for the programmer.

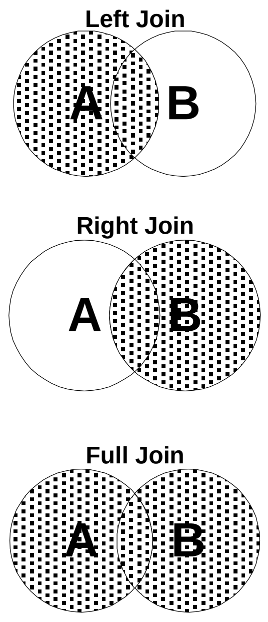
Joins versus Merges

In the last example (1B) we used a type of join called an *inner join*. Inner joins retrieve all matching rows from the WHERE clause. You may specify up to 16 data sets in your FROM clause in an inner join. Graphically (using only two data sets) the inner join

looks like this:



There are also joins called *outer joins* which can only be performed on two tables at a time. They retrieve all matching rows plus any non-matching rows from one or both of the tables depending on how the join is defined. There are three common types of outer joins: Left joins, right joins and full joins. Each type of join is illustrated below:



In the merge step from example 1A, the result was equivalent to performing a full join in PROC SQL where we ended up with not only matching records but also all non-matching records from each data set. The PROC SQL code would look as follows:

```
PROC SQL;
  SELECT A.ZIP, A.STATE, B.CITY
  FROM STATES A FULL JOIN CITYS B
  ON A.ZIP=B.ZIP;
```

Outer joins use a slightly different syntax. An ON clause replaces the WHERE clause but both serve the same purpose in a query. On the FROM clause, the words FULL JOIN replace a comma to separate the tables. Similarly for left and right joins, the word FULL would be replaced by LEFT and RIGHT respectively.

Comparing SQL joins with merging, the following IF statements used in the merge example would be equivalent to these types of outer joins:

IF A;	Left Join
IF B;	Right Join

How Joins Work

In theory, when a join is executed, SAS builds an internal list of all combinations of all rows in the query data sets. This is known as a Cartesian product. So for the STATES and CITYS data sets in our examples, a Cartesian product of these two data sets would contain 9 observations (3 in STATES x 3 in CITYS) and would look conceptually like the table below.

STATE	A.ZIP	B.ZIP	CITY
NJ	08000	10000	NEWYORK
NJ	08000	19000	PHILADELPHIA
NJ	08000	90000	LOSANGELES
NY	*10000*	*10000*	*NEWYORK*
NY	10000	19000	PHILADELPHIA
NY	10000	90000	LOSANGELES
PA	19000	10000	NEWYORK
PA	*19000*	*19000*	*PHILADELPHIA*
PA	19000	90000	LOSANGELES

The italicized rows would be the ones actually selected in our inner join example because the WHERE clause specified that the zips be equal. Conceptually, SAS would evaluate this pseudo-table row by row selecting only the rows that satisfy the join condition. This seems to be a highly inefficient way of evaluation but in reality SAS uses some internal optimizations to make this process more efficient.

Advantages to Using PROC SQL instead of MERGE.

There are several worthwhile advantages to using PROC SQL instead of data step merging. Using PROC SQL requires fewer lines to code.   In example 1B we eliminated two PROC SORTS (6 lines of code), an entire data step (5 lines) and a PROC PRINT (2 lines). We replaced it with 4 lines of PROC SQL code for a net loss of 9 lines of code. It doesn't sound like much but if you normally use MERGE a lot it can add up to something significant. Also, the fewer lines of code the better chance of error free code.

You also don't need to have a common variable name between the data sets you are joining. This can come in handy if you are dealing with SAS data sets created by someone who is not aware of how you are using them. If you were merging these data sets you may need to rename one of the variables on one of the data sets. The WHERE/ON clause in PROC SQL lets you specify each different variable name from each data set in the join.

Another benefit is that you do not need to sort each input data set prior to executing your join. Since sorting can be a rather resource intensive operation when involving very large data sets, replacing a merge with a PROC SQL join can potentially save you processing time. Note that factors such as data set indexes and the size of the data sets can also have effects on performance. Analysis of these different factors was not performed because it is not within the scope of this paper. Any references to potential efficiency improvements are qualitative, not quantitative.

PROC SQL also forces you to be more diligent with variable maintenance. If you MERGE two or more data sets and these data sets have common variable names (not including the BY variable), the variable on the output data set will contain the values from the right most data set listed on the MERGE statement. This may not be what you wanted and is a common oversight when using MERGE. There is a tendency in data step merging to carry all variables and to not specify only the necessary variables. In a PROC SQL join, if you are creating a new table from two or more data sets and those data sets have common variable names, SAS will give an error because it will not allow more than one variable with the same name on the output data set. It forces the programmer to either include only the necessary variables in the SELECT statement or rename them in the SELECT statement using the following syntax:

```
SELECT var1 as newvar1, var2 as newvar2
```

You can rename the variables as they are input to a MERGE but if you don't do this and variable overlaying occurs, there are no error or warning messages.

Another advantage is that once you become accustomed to using PROC SQL as a novice it will give you the confidence to explore many more of its uses. You can use PROC SQL to perform summaries, sorts and to query external databases. It's a very powerful procedure.

Things to be careful with in PROC SQL

If you are used to using MERGE and never had any SQL experience there are a few things you must be aware of so that you fully understand what happens in SQL joins. The entire Cartesian product concept is usually a new one for most veterans of the merge. Match merging in SAS does not work in the same manner.

Match merging is best used in situations where you are matching one observation to many, many observations to one, or one observation to one. If your input data sets contain repeated BY variables, i.e., many to many, the MERGE can produce unexpected results. SAS prints a warning message in the SASLOG if that situation exists in your data. This is a helpful message because we sometimes think our data is clean but duplication of by variables can exist in each data set without our knowledge.

PROC SQL, on the other hand, will always build a virtual Cartesian product between the data sets in the join. Analogously, situations of duplicate where-clause variables will not be identified in any way. It's important to understand the data.

Introduction to PROC FORMAT

We can also use PROC FORMAT along with some help from the PUT function to replicate a merge but first a review of PROC FORMAT.

PROC FORMAT is often used to transform output values of variables into a different form. We use the VALUE statement to create user-defined formats. For example, let's say we have survey response data consisting of three questions and all of the answers to our questions are coded as 1 for "yes", 2 for "no" and 3 for "maybe". Running a frequency on these values would show how many people answered 1,2 or 3 but it would be much nicer if we could actually see what these values meant. We could code a PROC FORMAT step like the following:

```
PROC FORMAT;
   VALUE ANSWER
      1='YES'
      2='NO'
      3='MAYBE';
RUN;
```

This creates a format called ANSWER. You could now run a frequency on the answers to Q1, Q2 and Q3 as follows:

```
PROC FREQ DATA=ANSWERS;
   FORMAT Q1 Q2 Q3 ANSWER.;
   TABLES Q1 Q2 Q3;
RUN;
```

We used the FORMAT statement here to apply the conversion in the PROC step. The output would show frequencies of “YES”, “NO” and “MAYBE” as opposed to their corresponding values of 1,2 and 3.

Using PROC FORMAT in this manner is fine if you are using a static definition for your conversion. Imagine now that you have data from another survey but this time the answer codes are totally different where 1 is “Excellent”, 2 is “Good”, 3 is “Fair” and 4 is “Poor”. You would have to re-code your PROC FORMAT statement to reflect the new conversion. Although it is not a lot of work to do here, it can become cumbersome if you have to do it every day. There is a convenient way to lessen the maintenance.

Using CNTLIN= option

There is an option in PROC FORMAT called CNTLIN= that allows you to create a format from an input control data set rather than a VALUE statement. The data set must contain three specially named variables that are recognized by PROC FORMAT: START, LABEL and FMTNAME. The variable named START would be the actual data value you are converting from, e.g., 1, 2 or 3. The LABEL variable would be the value you are converting to, e.g., YES, NO, MAYBE. The FMTNAME variable is a character string variable that contains the name of the format, e.g., ‘ANSWER’. You would simply read in the data set and appropriately assign the variable names. Then run PROC FORMAT with the CNTLIN option. See the following example:

```
DATA ANSFMT;
   INFILE external filename;
   INPUT @1 START 1.
            @3 LABEL $5;
   FMTNAME='ANSWER';
RUN;

/* The external input data set would look like:
1 YES
2 NO
3 MAYBE
*/

PROC FORMAT CNTLIN=ANSFMT;
RUN;
```

Both methods create the same format called ANSWER. In this example we would not have to go into the SAS code every time we had a new definition of answer codes. It allows the programmer more flexibility in automating their program.

Replicating a MERGE with PROC FORMAT using the PUT function.

Sometimes it may become necessary to create an entirely new variable based on the values in your format. It may not be enough to simply mask the real answer value with a temporary value such as showing the word “YES” whenever the number 1 appears. You may need to create a new variable whose permanent value is actually “YES”. This can be done using the PUT function along with a FORMAT. Showing how the PUT function is used, we’ll stay with our survey example. Our data sets look like this:

Survey Data: ANSWERS

ID	**Q1**
111	1
222	2
333	3

Format Data: ANSFMT

START	**LABEL**	**FMTNAME**
1	YES	ANSWER
2	NO	ANSWER
3	MAYBE	ANSWER

We will create a new variable using the PUT function that will be the value of the transformed Q1 variable. The format of the PUT function is: *PUT(variable name,format).* An example of the program code would look like:

```
DATA ANSWERS;
   SET ANSWERS;
   Q1_A = PUT(Q1,ANSWER.);
RUN;
```

The output data set would look like this:

ID	Q1	Q1_A
111	1	YES
222	2	NO
333	3	MAYBE

We have in effect "merged" on the Q1_A variable. An advantage to using this method in place of the merge would be in a situation where we had several variables (Q1, Q2, etc.) that all required the same transformation. You could just apply another PUT statement for each variable rather than perform multiple merges and renaming of merge variables.

Limitations and things to be careful with using the PUT function.

You must be careful that all expected values of your "before" variables have a corresponding value in the format list. If a value in the data exists that has no matching value in the format data set then the new variable will retain the value of the "before" variable as a character string. There are other options in PROC FORMAT such as OTHER and _ERROR_ that can be used to highlight these types of situations. These options will not be discussed here.

This method is best used when you would only need to merge on one variable from another data set. If we had a data set containing eight different attributes that we want added to a master file, it would require creating eight CNTLIN data sets and eight PROC FORMATS before executing eight PUT functions. A MERGE or PROC SQL would be better suited in such a case.

Conclusions

As you can see, there are other alternatives in SAS to using the traditional MERGE when combining data set information. It is not too difficult to learn a few beginner lines of PROC SQL code to execute what most merges do. The PROC FORMAT / CNTLIN / PUT method is also a quick way to get limited information from other data sets. Hopefully you will try out these new methods and expand you SAS knowledge in the process.

References

SAS Language Reference, Version 6, First Edition.

SAS Procedures Guide, Version 6, Third Edition.

SAS Guide to the SQL Procedure, Version 6, First Edition.
SAS® is a registered trademark of the SAS Institute, Inc., Cary, NC USA.

The Practical SQL Handbook, Third Edition; Bowman, Emerson, Darnovsky

Acknowledgements

Thanks to Perry Watts, Sr. Statistical Programmer, IMS Health and John Gerlach, SAS consultant at IMS Health, for providing various ideas, suggestions and technical assistance.

Author Information

Michael Wieczkowski
IMS HEALTH
660 West Germantown Pike
Plymouth Meeting, PA 19462
Email: Mwieczkowski@us.imshealth.com