PhUSE 2009

Paper PO06

The Perfect Match - an appreciation of the functionality of the dataset MERGE

Diana Stuart, Takeda, London, UK

ABSTRACT

Performing a match-merge using the MERGE statement can be a versatile and efficient method of combining SAS datasets, whilst providing clear code readability for debugging or code review. The purpose of this paper is to review the functionality that can be achieved using the MERGE statement to combine datasets within a datastep, and give programmers a better understanding of the power of the MERGE. Being aware of the options available using a MERGE statement, could lead to efficient use of code, fewer datasteps and proper consideration of the structure of the data being merged.

INTRODUCTION

Match-merging is regularly used for combining datasets. Using a basic 4-statement match-merge with no options, for example:

```
data demo_vital;
  merge demo
        vital;
  by subject;
run;
```

and carrying out all other processing in separate datasteps, is a sound “better safe than sorry” strategy. It has the benefits of being quick, easy and little consideration of the data is required. Legacy programs often contain such code, using multiple datasteps for merging, when in actual fact one will do. The benefits of using the options available in a MERGE statement and streamlining code into one datastep are; code efficiency, less processing time, but requiring more familiarity with the data.

It is accepted that there are some pitfalls associated with match-merging, however, a programmer that is familiar with their data and understands SAS processing will appreciate the efficiency and quantity of manipulation that can be achieved in one datastep using the options available, as presented in this paper.

KEEP/DROP

Using KEEP and DROP statements both on the input datasets and the output dataset(s), means that the output dataset contains only exactly what you require. It demonstrates that you, the programmer know what you want to select from the input datasets and what you need in the dataset output. Otherwise, it’s a bit like going to the supermarket knowing you need to buy some ingredients to bake a cake but you’re not sure exactly what, so you buy one of everything in the store. You end up with a lot of things you don’t need and will never use, plus you have to store them somewhere.
Streamlining the data going into your merge, improves the efficiency of your merge.

This example demonstrates creating a dataset containing demography and vital signs data, selecting only the variables required from the input datasets, and after creating a new variable within the datastep, dropping the variable that is no longer required.
The source datasets are demo, a dataset containing demographic data with one observation per subject, and vital, a dataset containing vital signs data, with one observation per subject, per visit, per parameter, demo and vital have been sorted into *subject* order:

```
data demo_vital(drop=age);
  merge demo(keep=subject sex race age height)
        vital(keep=subject visit parameter value);
  by subject;
  if age le 65 then age_group=1;
  else age_group=2;
run;
```

WHERE

Using a WHERE clause is efficient, because it subsets the data prior to the data entering the input buffer. Having selected only the variables we require, we are now selecting only the rows of data we require. Going back to the cake and supermarket analogy, you wouldn't buy all the different types of flour available to bake a cake, you would only select the type of flour you require to bake the cake.

This example demonstrates merging only vital signs observations that have a value:

```
data demo_vital;
  merge demo(keep=subject sex race age height)
        vital(keep=subject visit parameter value
              where=(value ne .));
  by subject;
run;
```

RENAME

Programmers often overlook the benefit of merging different subsets from the same source dataset. Doing this makes good use of the WHERE clause, but often you will need to RENAME some variables to prevent overwriting.

The example below demonstrates this, creating a dataset containing demography and weight assessment data, and a baseline weight variable is also created:

```
1.    data demo_weight(drop=parameter visit1);
2.      merge demo(keep=subject sex race age height)
3.            vital(keep=subject visit parameter value
4.                  rename=(value=weight)
5.                  where=(weight ne . and parameter eq "Weight"))
6.            vital(keep=subject visit parameter value
7.                  rename=(value=b_weight visit=visit1)
8.                  where=(b_weight ne . and visit1 eq 1 and parameter eq "Weight"));
9.      by subject;
10.   run;
```

Looking at this datastep in more detail:

It is important to know that the options are processed in alphabetical order, and therefore it is considered good practice to write your code in the same order, which will allow you to easily see how they are applied.

```
1.    data demo_weight(drop=age parameter visit1);
```

A dataset called demo_weight is created and the variables *age*, *parameter* and *visit1* are dropped.

```
2.     merge demo(keep=subject sex race age height)
```

A dataset called demo is the first dataset to be read into the merge. Remember the order of this dataset dictates the order of the output dataset. The variables *subject, sex, race, age* and *height* are selected to be kept in this dataset.

```
3.           vital(keep=subject visit parameter value
```

The next dataset to be merged is vital. Three options are used on this dataset. Firstly, variables *subject, visit, parameter* and *value* are selected to be kept.

```
4.                 rename=(value=weight)
```

Although it is not necessary to use RENAME here, it is done for clarity so the variable name reflects the data in the variable. Remember, old=new.

```
5.                 where=(weight ne . and parameter eq "Weight"));
```

A where condition is applied, and as one of the variables in this condition has been renamed the new variable name must be used in the code. The where condition selects observations where the parameter name is Weight and the associated assessment value is not missing.

```
6.           vital(keep=subject visit parameter value
```

The last dataset is vital again, and again three options are used on this dataset. Firstly, variables *subject, visit, parameter* and *value* are selected to be kept.

```
7.                 rename=(value=b_weight visit=visit1)
```

Next, two of the variables are renamed. Here a bit of forward thinking is required as we are merging 2 different subsets of the same dataset. We need to think about each of the 4 variables in the keep statement in the previous line. Ultimately we want a variable that is the baseline measurement of weight, when the WHERE is applied, *value* holds that data so we rename it to something appropriate, in this case *b_weight*. *Subject* is our BY variable and *parameter* is dropped in the output statement, which leaves *visit*. We need this variable because it is used to subset the input dataset, however, we need to bear in mind that there is a variable of the same name in the 2nd dataset in the merge statement that we want to keep in our output dataset. Even though we don't need the data from this variable, it is good practice to rename it to differentiate from the other variable *visit* and more importantly to prevent it from overwriting the other variable *visit*.

```
8.                 where=(b_weight ne . and visit eq 1 and parameter eq "Weight"))
```

Lastly, a where condition is applied. One of the variables in this condition has been renamed and therefore, the new variable name must be used in the code. The where condition selects observations where the parameter name is

Weight and the associated assessment value is not missing and the visit number is 1, which is the visit that is used for baseline in this example.

```
9.      by subject;
```

The variable *subject* is used to match observations from demo and vital. It is important to note that in order to avoid problems with repeats of BY variables that the BY variable or combined BY variables must be unique in all but one of the datasets being merged.

```
10.   run;
```

It is good programming practice to always have a run statement at the end of a datastep.

IN

This creates a temporary flag, which can be used to determine which input dataset the data came from.

In the example below two datasets are merged and IN has been used to create the temporary flags *a* and *b* (the flags can be called any name of your choosing, although the nomenclature of using single letters is a common one).

```
data demo_vital;
  merge demo(in=a)
        vital(in=b);
  by subject;
  if a;
run;
```

Remember, *subject* is the BY variable that must be matched. In this example, the output dataset contains all the subjects from demo, and those in vital if they match those in demo.

These temporary flags can give a programmer a lot of power in terms of selecting exactly which observations they wish to keep in a dataset. The merge might be used to provide a subset even when no variables are being added to the output dataset. For example:

```
data demo_subset;
  merge demo(in=a)
        status(keep=subject in=b);
  by subject;
  if a and not b;
run;
```

Here, demo is merged with the dataset status which contains subjects that are positive for a particular status. If we wished to exclude those subjects from the original dataset demo in a new dataset, we can, by assigning the temporary flags *a* and *b* and using an IF statement.

CONCLUSION

Combining datasets using a match-merge is a basic piece of coding, usually learned by programmers early on. Properly understanding what is going on in the code means being able to take full advantage of the options available. Efficiency of the merge can be greatly increased by only reading in the data required by using DROP/KEEP and WHERE and using WHERE, RENAME and IN to manipulate data. A full understanding of these can help programmers avoid unnecessary use of additional datasteps and reduce processing time which can be particularly invaluable when dealing with large data.

CONTACT INFORMATION

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

Diana Stuart
Takeda Global Research & Development Centre (Europe) Ltd.
61 Aldwych
London WC2B 4AE

d.stuart@takeda.com