DATA Step Merging Techniques: From Basic to Innovative

Arthur L. Carpenter
California Occidental Consultants, Anchorage Alaska

ABSTRACT

Merging or joining data sets is an integral part of the data consolidation process. Within SAS there are numerous methods and techniques that can be used to combine two or more data sets. We commonly think that within the DATA step the MERGE statement is the only way to join these data sets, while in fact, the MERGE is only one of numerous techniques available to us to perform this process. Each of these techniques has advantages and some have disadvantages. The informed programmer needs to have a grasp of each of these techniques if the correct technique is to be applied.

This paper covers basic merging concepts and options within the DATA step, as well as, a number of techniques that go beyond the traditional MERGE statement. These include fuzzy merges, double SET statements, and the use of key indexing. The discussion will include the relative efficiencies of these techniques, especially when working with large data sets.

INTRODUCTION

Merging two or more data tables is an essential data manipulation process. Known as a join when performed in a SQL step, in the DATA step the MERGE statement coordinates the process of bringing in the data from multiple tables to create a unified set of variables.

Merging two or more data sets in the DATA step is not limited to just the MERGE statement. Indeed a number of other techniques are available that have a number of advantages including performance enhancements and flexibilities that are not available using the tradition MERGE statement. This paper will introduce a number of these techniques, however since it is concentrating on the DATA step, techniques using alternate tools, such as SQL, will not be discussed.

The data used throughout this paper are four small clinical trial data sets. The first CLINDAT.PATIENT has patient identifying information, because this is a blinded study the identifying information must be stored separately until we are ready to un-blind the study. This data has one row per patient and PATID is a unique row identifier (primary key). The second data set (CLINDAT.TRIAL) has the study information. It has a different primary key PATID and DT_DIAG. The third data set (CLINDAT.CONMED) lists the medications being taken for each patient. The variables PATID and DRUG form its primary key. The fourth data set has adverse event information (CLINDAT.AE), and requires three variables as its primary key (PATID, AESTART, and AEDESC). The various data sets do not have the same number of observations.

REVIEW OF A MERGE

When two or more data sets are merged, their individual variable lists are joined into a single set of variables. Generally these two lists are fairly different with one or more key variables in common that allow the merging process to create associations among observations. In the simplest merge there are no key variables and the MERGE statement is used without a BY statement. This is known as a one-to-one merge, and is generally, except under very specific circumstances, a bad idea. Almost always the MERGE statement is immediately followed by a BY statement which lists the variables that form the key that coordinates the association between the observations that are read from the incoming data sets. This forms what is known as a match merge (shown here). The match merge assumes that both data sets have been sorted (or indexed – see below) for the BY variables.

```
data twomerge;
   merge clindat.patient
         clindat.trial;
   by patid;
   run;
```

Requirements

When the MERGE statement is used with a BY statement, the incoming data sets must be sorted (or indexed). This is known as a match merge and the BY variables should form a primary key (identify down to the row level) in all, but at most one, of the incoming data sets. If the BY variables do not form a primary key, a one-to-one merge will take place within that BY group.

Cautions

The merge process assumes that the variable lists of the incoming data sets are unique and that the only variables in common are those listed on the BY statement.

Because SAS builds the Program Data Vector (PDV) from left to right, the leftmost data set in the MERGE statement will be first to supply variable attributes. If there are overlapping variables, only one can reside on the PDV and it will have the attributes associated with the left most of the data sets that contains it. This includes the BY variables. It is always wise to ensure that the attributes of the BY variables are consistent across all of the data sets.

Behaviors

For a given combination of BY variables if a corresponding observation does not exist within a data set, the variables contributed by that data set will be missing.

When two or more observations have the same combination of BY variables in one data set and there is only one matching observation in the other data set the row in that data set will be duplicated.

MERGING WITH A DOUBLE SET

The MERGE statement coordinates the reads of the observations from the incoming data sets. It is possible to take charge of this process by using two SET statements. In its simplest form two SET statements in a DATA step will result in a unified PDV. The result, however is not necessarily the same as a merge. The example shown here is almost like a one-to-one merge, but it is not the same. When reading data using a SET statement, as is done here, as soon the last observation is read from *either* data set the reading process ends. This means that the total number of observations will be governed by the smaller of the incoming data sets. If we want to use this technique we will have to take control of the read process.

```
data twoset;
   set clindat.patient;
   set clindat.trial;
   run;
```

Because we are taking charge of the merging process, a DATA step constructed this way, even with the logic needed to control the reads, will often be faster than a merge managed by a MERGE statement. However the logic is more complex. Here an observation is read from the first data set and the patient id is renamed. The DO WHILE is then entered and observations are read until the patient codes match (this keeps the two data sets in sync with each other). Obviously as the data becomes more complex so too does the logic needed to read it successfully.

```
data doubleset;
  set clindat.patient(rename=(patid=code));
  * The following expression is true only
  * when the current CODE is a duplicate.;
  if code=patid then output;
  do while(code>patid);
    * lookup the study information using
    * the code from the primary data set;
    set clindat.trial(keep=patid symp dt_diag);
    if code=patid then output;
  end;
run;
```

MANY TO MANY MERGE

While a many to many merge is the default in a SQL join, it is often said that it is not possible to perform one in the DATA step. This is not correct. A many to many merge matches each observation from one data set to each in another data set. This tends not to be practical, but it is not uncommon to match all observations within a group with all of those in the same group in the other data set. As a general rule it is much more practical to do this in a SQL step. But there can sometimes be reasons that would make the DATA step a more attractive alternative.

In the code that follows merges a data set back on itself. Here we would like to match all the females in our study with all the males with the same symptom (SYMP) so that we can see how consistent the diagnosis codes (DIAG) are across gender. In this approach (there is a more efficient example of many to many merging in the section on using arrays), we read each observation several times. The first SET statement reads each female and then the interior DOW loop rereads the entire data set to find all observations that meet the stated criteria (males with the same symptom as the females). The key to rereading the data is the use of the POINT= and NOBS= options on the SET statement. These are used to control the looping.

```
data manymany(keep=f_patid m_patid f_symp m_diag f_diag
              rename=(f_symp=symp));
   set clindat.trial(rename=(patid=f_patid diag=f_diag
                             symp=f_symp sex=f_sex)
                     where=(f_sex='F'));
   do pt = 1 to nobs;
      set clindat.trial(rename=(patid=m_patid diag=m_diag
                                symp=m_symp sex=m_sex))
          point=pt nobs=nobs;
      if m_sex='M' and f_symp=m_symp then output manymany;
   end;
   run;
```

Remember, if you are considering a many to many merge, first look at a SQL step, but do not dismiss the DATA step out-of-hand.

TAKING ADVANTAGE OF INDEXES

One of the disadvantages of the MERGE is that both incoming data sets must be sorted in order to use the BY statement. For smaller data sets this may not be a very big consideration, but as data sets become large sorting itself can become problematic. If one or both of the data sets are indexed the sorting can be avoided. Indexes are a way to logically sort your data without physically sorting it.

Indexes must be created, stored, and maintained. They are most usually created through either PROC DATASETS (shown below) or through PROC SQL. The index stores the order of the data as if it had been physically sorted. Once an index exists, SAS will be able to access it, and you will be able to use the data set with the appropriate BY statement, even though the data have never been sorted. You are not limited to a single index on a data set, so any given data set can be effectively sorted multiple ways at any given time.

Obviously there are some of the same limitations to indexes that you encounter when sorting large data sets. Resources are required to create and maintain the index, and these can be similar to the SORT itself. The index(es) is stored in a separate file, and the size of this file can be substantial, especially as the number of indexes, observations, and variables used to form the indexes increases.

Indexes can substantially speed up processes, but they can also *SLOW* things down if they are not constructed effectively. Be sure to read and experiment carefully before investing a lot in the use of indexes. Indexes are best suited to be applied to large stable data sets. Large and stable will depend on your organization and how it processes its data, but if a data set is constantly changing, it may not be a good candidate as the index would need to be recreated each

time the data set changes. For most of my work the definition of large is: 'Sigh, I have to sort that puppy' and stable data sets are usually not recreated much more often than nightly.

Creating Indexes

When a merge is performed on an indexed data set, the BY statement can be used as if the data were sorted.

The index can be created using a PROC DATASETS step. Here an index is created for the primary key of each of the two data sets of interest. Indexes are named although we rarely if ever use the name of the index. Indexes consisting of a single variable (PATID for CLINDAT.PATIENT) are automatically named for the variable. When the two or more variables are used to form the index as they are for CLINDAT.TRIAL, the index must be given a name (here the name is KEYNAME).

```
proc datasets library=clindat nolist;
  modify patient;
    index create patid / unique;
  modify trial;
    index create keyname=(patid dt_diag);
quit;
```

Merging without Sorting

Once indexes have been created the data set will no longer need to be physically sorted as long as it is virtually sorted (indexed) using the appropriate key variables. You do not need to do anything different to take advantage of the indexes. Notice that the composit index for CLINDAT.TRIAL is available, eventhough in this example we have only used the first of the two variables in the index.

```
data indexmerge;
   merge clindat.patient
         clindat.trial;
   by patid;
   run;
```

Using the Key= Option

You can also perform a merge when an index only exists on one of the data sets. This is very important when one of the data sets cannot be sorted or for whatever reason cannot be indexed. In the example shown here, the data set CLINDAT.TRIAL represented the unsorted and unindexed large data set. We only need to know that it contains the variable PATID, which is an index for the second (usually smaller) data set. As each observation is read from CLINDAT.TRIAL the value of PATID is used by the KEY= option to perform a random access read of the corresponding of the indexed data set.

```
data keymerge(keep=patid sex lname fname symp diag);
   set clindat.trial; *Master data;
   set clindat.patient key=patid/unique;
   if _iorc_ ne 0 then do;
      * clear variables from the indexed data set;
      lname=' ';
      fname=' ';
   end;
   run;
```

It is possible that a PATID in the primary data set will not exist in the secondary (indexed) data set. When this happens a read of the secondary data set cannot take place and SAS sets the temporary / automatic variable _IORC_ to something other than zero. Here we use this information to reset the secondary table's variables that would otherwise be retained from the previous observation that was successfully read from CLINDAT.PATIENT. There are a number of other functions that utilize this return code to help you during the debugging phase.

The resulting data set will be in the same order as the incoming master data set (CLINDAT.TRIAL).

MERGING WITH ARRAYS

Another solution to the merging problem, which does not require any sorting or indexing, is achieved through the use of arrays. Because arrays are held in memory, their access times, even for very large arrays, are extremely fast. Once understood, variations on the use of array can be used to solve a wide variety of programming problems.

Many to Many Merge Using Arrays

The key to any of the various types of merges with arrays is to store some portion of the data in one or more arrays. Usually you should pick the smaller data set to load into the array, but that really does not matter too much. In this example we return to the many-to-many merge problem. One of the problems with the previous solution was that we had to reread the data multiple times. In this solution each observation is read only once, and only a portion of the data is held in memory.

Remember we want to match all males to all females in the study that have the same symptom code. First we use a DO UNTIL loop to load all of the information on the males into a series of arrays. In this case each male observation goes into the next position (MCNT) of the array.

```
data manymany2(keep=f_patid m_patid f_symp m_diag f_diag
               rename=(f_symp=symp));
   * Use arrays to hold the retained (male) values;
   array mpatids {100000}    _temporary_;
   array mdiags  {100000} $1 _temporary_;
   array msymps  {100000} $2 _temporary_;
   do until(done);
      set clindat.trial(keep=patid diag symp sex
                        where=(sex='M')) end=done;
      * Save male data to arrays;
      mcnt+1;
      mpatids{mcnt} = patid;
      mdiags{mcnt}  = diag;
      msymps{mcnt}  = symp;
   end;
   do until(fdone);
      set clindat.trial(rename=(patid=f_patid diag=f_diag
                                symp=f_symp sex=f_sex)
                        where=(f_sex='F')) end=fdone;
      do i = 1 to mcnt;
         * Retrieve male values;
         if msymps{i}=f_symp then do;
            m_patid = mpatids{i};
            m_diag  = mdiags{i};
            output manymany2;
         end;
      end;
   end;
   stop;
   run;
```

Once all of the data for the males has been read into the arrays, we can then read the female data. For each female we check all of the male data for the same symptom. For each match we write out an observation.

The dimension of the arrays needs to be at least as big as the number of males. There is essentially no penalty for making the dimension too big. A numeric array with a dimension of a million elements only uses about eight megabytes of memory. Not much by today's standards.

Using Key Indexing Techniques

Key indexing is similar to the technique used with the many to many merge with a very important 'key' difference. In both techniques primary information is loaded and held in memory through the use of arrays. In both cases the information is then retrieved from the arrays based on data read from a second data set. The difference is in how the data are stored and retrieved from the arrays. In the previous example the data for the first male went into the first array position, the second male into the second position, and so on. Retrieval required us to step through the whole array to find the information of interest. In key indexing the data are not stored sequentially but rather according to some other data value. This means that retrieval can be much more efficient. It also means that neither data set needs to be sorted or indexed. When applicable this will tend to be the fastest possible merge of two data sets.

In this example we want to retrieve the last and first name from the patient data set (CLINDAT.PATIENT). Two arrays are created to hold all of the potential values. In this technique the dimension of the array must be as large as the largest patient number. The dimension is not necessarily the number of patients, but the largest patient number. Say for a one hundred patient study the patient numbers ranged from 301 to 400, the arrays would either be dimensioned for 400 items or dimensioned for one hundred (with the index ranging from 301 to 400). Here we just choose an arbitrarily large enough number (100,000).

The first DO UNTIL loop reads all of the patient data into the arrays (one array for each variable to be saved). It is very important to notice that the information is placed into the array by using PATID as the index. If the patient identifier takes on the value of 4,057 then his information will go into the 4,057th array location. This makes the information retrieval especially quick.

```
data keyindex(keep=patid lname fname symp diag);
   * Use arrays to hold the retained (patient) values;
   array lastn  {100000} $10 _temporary_;
   array firstn {100000} $6  _temporary_;
   do until(done);
      * read and store the patient data;
      set clindat.patient(keep=patid lname fname) end=done;
      * Save Patient data to arrays;
      lastn{patid}  = lname;
      firstn{patid} = fname;
   end;
   do until(tdone);
      set clindat.trial(keep=patid symp diag) end=tdone;
      * retrive patient data for this patid;
      lname = lastn{patid};
      fname = firstn{patid};
      output keyindex;
   end;
   stop;
   run;
```

The second DO UNTIL loop reads the trial data, including the patient identifier (PATID). This variable is then used as the array index to retrieve the appropriate values.

This technique is unaffected by patients with no trial data, and patients with no patient history data will automatically have missing values returned from the array.

WITHOUT A PRIMARY KEY (THE FUZZY MERGE)

A merge in which there is no combination of key variables available that will match up the observations to be joined is known as a fuzzy merge. The MERGE statement requires BY variables that match in both data sets in order to align the observations so that it can perform a match merge, when these are not available a match merge will not be successful.

One solution, although not particularly practical is to expand the BY variables in such a way as to create alignment in the BY values. This approach can result in extremely large data sets and correspondingly slow merge times. A more efficient approach is to perform a merge that is a variation on the array approach shown in the previous example.

In the following example patients have recorded when they took certain medications. The data (CLINDAT.CONMEDS) has one observation per medication with the start and end dates being noted. For any given patient there may be a number of drugs with overlapping, but not necessarily concurrent date ranges. The patients have also reported various adverse events (AEs). The adverse event data (CLINDAT.AE) records the date and type of event. We would like to merge these two data sets to find out what if any medications the patient was taking on the date of the adverse event, but date cannot be used as a BY variable because the medication data only has a date range.

This type of problem can be approached in a number of ways, several of which require multiple reads of the data. The use of doubly subscripted arrays will allow us to read each observation exactly once.

The doubly subscripted arrays will be indexed by patient number and date. The dimension of the array will correspond to the number of patients by the number of dates in the study (two years). Here the patient numbers range from 200 to 215, and these provide the first array dimension. The study start and stop dates are entered into macro variables, which are in turn used to provide the index range for the dates in the array.

```
%let start = %sysevalf('01jan2006'd);
%let stop  = %sysevalf('31dec2007'd);
data ae_meds(keep=patid aestart aedesc aemeds);
   array drgs {200:215, &start.:&stop.} $150 _temporary_;
   do until(donemed);
      * Read in the meds;
      set clindat.conmed(keep=patid drug cmstart cmstop)
          end=donemed;

      cmstart = max(&start,cmstart);
      if cmstop gt &stop or cmstop=. then cmstop=&stop;
      do date = cmstart to cmstop by 1;
         * assign this drug to each day in its date range;
         drgs{patid,date} = catx('*',drgs{patid,date},drug);
      end;
   end;
   do until(doneAE);
      * Read in AE data and check for previous meds;
      set clindat.ae(keep=patid aestart aedesc)
          end=doneae;
      if &start. le aestart le &stop. then do;
         AEMeds = drgs{patid,aestart};
         output ae_meds;
      end;
   end;
run;
```

The first DO UNTIL loop is used to read the medication data for each patient and to load the array with the medication names. Since the medication data has the date range for each medication this range is used as the index to fill the array. A given drug name is stored into each date that that drug was taken. In this solution the names of the drugs are concatenated into one name using the CATX function.

Each element in this character array takes up 150 bytes. Since there are fourteen patients (the index has left room for 16) and around 730 days in the study, the whole array takes up less than one megabyte of storage.

Once all the medication data has been loaded the AE data can be read one observation at a time. The AE start date (AESTART) becomes the index to the array which will return the medications taken by that patient on that date. By using this approach multiple medications across varying date ranges can be easily applied (merged) to a given AE start date, and neither of the two data sets has to be ordered (sorted) or indexed.

MERGING USING HASH TABLES

One of the limitations of array based approaches to merging is the necessity to use a numeric value as the array index. A number of techniques have been developed to work around this limitation, however in the current versions of SAS a better solution exists. Hash objects are in many ways like super arrays. They utilize indexes, but the indexes can be either character or numeric. Like an array they can store values in memory, but unlike an array any number of numeric and character values can be stored in each cell.

Essentially the HASH object defines (instantiates) an array in memory. The hash object is named and it can be preloaded with data. The object can also have one or more numeric or character indexing (KEY) variables. In this example the hash object is being used to store the last and first names of each patient. Hash objects are instantiated through the use of the DECLARE statement and they are operated against by the application of methods. These methods are used to assign attributes to the object as well as to read from it and to write to it.

On the DECLARE statement the hash object is declared and named, in this case it is named HMERGE, and this name is used with each method that is to be applied to the hash object. Naming an object is important as it is not uncommon to declare more than one hash object within a DATA step. By using the DATASET: constructor on the DECLARE statement we can preload the hash object with the values of a data set (CLINDAT.PATIENT). In order to use the DATASET: constructor, the variables in the incoming dataset have to have been predefined on the PDV. We can do this by using a compilation only SET statement.

```
data hashmerge(keep=patid lname fname symp diag);
   if 0 then set clindat.patient;
   declare hash hmerge(dataset: 'clindat.patient',
                       hashexp: 6);
      rc1 = hmerge.defineKey('patid');
      rc2 = hmerge.defineData('lname', 'fname');
      rc3 = hmerge.defineDone();

   do until(done);
      set clindat.trial end=done;
      rc4 = hmerge.find();
      if rc4 = 0 then output hashmerge;
   end;
   stop;
   run;
```

Methods are then used to define the key variables (DEFINEKEY) and the variables to be stored in the hash object (DEFINEDATA). After the hash object has been fully defined, and in this example filled with data, the declaration is closed using the DEFINEDONE method.

Once defined the hash object is ready to use. In this example a second DO UNTIL loop is used to read the data set (CLINDAT.TRIAL). Each observation that is read contains a value of PATID, which is the index variable used to access the hash object. The FIND method uses it to retrieve the corresponding LNAME and FNAME from the hash object. If a matching PATID is found (a successful execution of the FIND method returns a value of 0, in this case making RC4=0) the observation is written out to the resultant data set.

A merge is accomplished even though neither of these two data sets has been sorted or indexed. Usually this type of merge will be one of the fastest. Especially when taking time to sort or index the data sets is taken into consideration.

SUMMARY

Usually the merging of two data sets in a DATA step is as simple as using a MERGE statement to perform a match merge. True the incoming data sets must be sorted in order to use the BY statement, but for most data sets this is not much of a constraint. Sometimes, however the incoming data sets do not meet the criteria needed to successfully use a MERGE statement. Perhaps one or both of the data sets cannot be sorted. Perhaps there is not an adequate set of KEY variables to use with the BY statement. Perhaps the process of sorting and merging just takes too much time. Whatever the reason, when this happens you need to be aware that alternative merging techniques exist. These techniques may or may not require more advanced coding skills. Some may not be practical in all situations, but if you have not familiarized yourself with these techniques, at least to the level of understanding that you know what they can do, you will not be able to successfully apply them when the need arises.

ABOUT THE AUTHOR

Art Carpenter's publications list includes five books, and numerous papers and posters presented at SUGI, SAS Global Forum, and other user group conferences. Art has been using SAS® since 1977 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Advanced Professional programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

REFERENCES

Some of the examples in this paper have been borrowed (with the author's permission) from the book *Carpenter's Guide to Innovative SAS® Techniques* by Art Carpenter (SAS Press, 2012).

TRADEMARK INFORMATION