PharmaSUG 2014 - Paper AD03

Table Lookup Techniques: From the Basics to the Innovative

Arthur L. Carpenter, California Occidental Consultants

ABSTRACT

One of the more commonly needed operations within SAS® programming is to determine the value of one variable based on the value of another. A series of techniques and tools have evolved over the years to make the matching of these values go faster, smoother, and easier. A majority of these techniques require operations such as sorting, searching, and comparing. As it turns out, these types of techniques are some of the more computationally intensive, and consequently an understanding of the operations involved and a careful selection of the specific technique can often save the user a substantial amount of computing resources.

Many of the more advanced techniques can require substantially fewer resources. It is incumbent on the user to have a broad understanding of the issues involved and a more detailed understanding of the solutions available. Even if you do not currently have a BIG data problem, you should at the very least have a basic knowledge of the kinds of techniques that are available for your use.

KEYWORDS

Indexing, Key-indexing, Lookup techniques, Match, Merge, Select, KEY=, hash objects

INTRODUCTION

Along with the proliferation of very large data sets have come increasing powerful computers and data base management capabilities. Even with the increase in computing power, the need for a more thorough understanding of efficient lookup techniques has not diminished. Since publishing an earlier paper/presentation on table lookup techniques (Carpenter, 2000), the need to understand the more efficient and advanced techniques has not diminished. This paper is an update of the earlier one, with a concentration on the more advanced techniques. Readers that do not have a strong working understanding of the basic techniques should first read the earlier paper.

Techniques given only cursory attention in this paper, and more attention in the previous paper, include the following:

- IF-THEN
- IF-THEN-ELSE
- SELECT statement
- MERGE
- SQL joins

The bulk of the discussions in this paper will concentrate on the use of the following techniques:

- user defined formats with the PUT and INPUT functions
- merging with two SET statements
- using the KEY= option
- use of ARRAYS
- Key-Indexing
- the use of hash objects

Anyone who performs table lookups should have at least a general understanding of these techniques. Even if you do not learn them now, know that they exist so that you can learn and use them later. All users will be confronted with lookup situations and they need to be able to compare, and contrast techniques so that they can select the technique that is appropriate for their particular situation.

A table lookup is performed when you use the value of one (or more) variables (*e.g*. a patient identification code) to determine the value of another variable or variables (*e.g*. patient name). Often this second piece of information must be

'looked up' in some other secondary table or location. The process of finding the appropriate piece of information is generally fast, however as the number of items and/or observations increases, the efficiency of the process becomes increasingly important. Fortunately there are a number of techniques for performing these table lookups.

The simplest form of a table lookup makes use of the IF-THEN statement. Although easy to code, this is one of the slowest table lookup methods. Substantial improvement can be gained merely by the inclusion of the ELSE statement. The SELECT statement has a similar efficiency to the IF-THEN/ELSE, however there are efficiency differences here as well.

The use of FORMATS allows us to step away from the logical processing of assignment statements, and allows us to take advantage of the search techniques that are an inherent part of the use of FORMATS. For many users, especially those with smaller data sets and lookup tables (generally less than 30,000 items), the efficiency gains realized here may be sufficient for most if not all tasks.

Merges and joins are also used to match data values. The MERGE statement (when used with the BY statement - as it usually is) requires sorted data sets, while the SQL step does not. There are advantages to both processes. Depending on what you need to determine, it is usually also possible to build a merge process yourself through the use of arrays and without using the SQL join or the MERGE statement. Substantial performance improvements are possible by using these large temporary arrays as they can eliminate the need to sort the data.

Users with very large data sets are often limited by constraints that are put upon them by memory or processor speed. Often, for instance, it is not possible/practical to sort a very large data set. Unsorted data sets cannot be merged by using BY statements. Joins of very large data sets using SQL may be possible by using the BUFFERSIZE option, but this still may not be a useful solution. Fortunately there are a number of techniques for handling these situations as well.

THE DATA

The data used throughout this paper are small data sets taken from a clinical trial. The first CLINDAT.PATIENT has patient identifying information. Because this is a blinded study, the data 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, and it has a different primary key (PATID and DT_DIAG). The two data sets do not have the same number of observations.

- CLINDAT.PATIENT   A list of patient names, their SSN value, and a unique patient identifier (PATID).
- CLINDAT.TRIAL   Study information for a small clinical trial. PATID is the only patient identifier.

We cannot assume that any data set has been sorted and for most of the examples we will want to look up the patient name given the patient identifier (PATID). To give you an idea of their contents, small portion of each data set is shown below.

VIEWTABLE: Clindat.Patient

	lname	fname	ssn	name	patid
1	Moon	Rachel	375363500	Moon, Rachel	51
2	Cranston	Rhonda	287463500	Cranston, Rhonda	18
3	Karson	Shawn	297854321	Karson, Shawn	36
4	Carlile	Patsy	578854321	Carlile, Patsy	11
5	Holmes	Donald	315674321	Holmes, Donald	30
6	Nolan	Terrie	298456241	Nolan, Terrie	54
7	Hermit	Oliver	471094671	Hermit, Oliver	29
8	Candle	Sid	468729812	Candle, Sid	10
9	Panda	Mery	387549812	Panda, Mery	56

VIEWTABLE: Clindat.Trial

	sex	dob	race	edu	symp	dt_diag	diag	patid
1	M	15JAN30	1	12		.		3
2	M	25JAN32	1	12		.		6
3	M	26JAN37	1	10	10	.		21
4	M	29JAN37	1	10	10	.		7
5	M	21MAY37	1	12		.		19
6	M	23MAY38	1	12		.		73
7	M	26FEB39	2	16	05	.	2	20
8	M	15OCT40	1	13		.		48

BASIC TECHNIQUES

The basic techniques described in the following section of this paper are fairly straightforward to master and are sufficient for small to medium size problems. They are not discussed in great detail here as they have been thoroughly covered in some of the other referenced papers such as Carpenter (2001). For the most part these techniques do not scale well, and as the size of the data sets or the number of items to look up increases so can the time to perform the look up. If you are using one of these techniques and experience substantial slowdowns due to increase processing requirements, you may wish to look into some of the more advanced techniques discussed later.

LOGICAL PROCESSING

Logical processing, which is accomplished through the use of IF-THEN/ELSE or SELECT statements is not particularly practical when the number of values to look up exceeds one's tolerance for typing repeated code. My tolerance is about three lines of repeated code (hence the size of the example shown here).

```
data logical;
   set clindat.trial;
   length lname $10 fname $6;
   if patid = 1 then do;
      lname='Adams'; fname='Mary'; end;
   else if patid = 2 then do;
      lname='Adamson'; fname='Joan'; end;
   else if patid = 3 then do;
      lname='Alexander'; fname='Mark'; end;
   else * others not shown;
   run;
```

In terms of the efficiency of the look up process, this is about your worst choice, the only worse one is to use the IF-THEN statements without the ELSEs. Of course efficiency is relative, if there really are only three or four items to look up and your data sets are small, it is quite possible that little or no performance gain can be achieved with the techniques described later in the paper.

Effectively we are '*hard coding*' the values to be recovered by making them a part of the code. Regardless of its relative efficiency this is just not practical for large numbers of items to be looked up.

```
data select;
   set clindat.trial;
   length lname $10 fname $6;
   select (patid);
      when (1) do; lname='Adams'; fname='Mary'; end;
      when (2) do; lname='Adamson'; fname='Joan'; end;
      when (3) do; lname='Alexander'; fname='Mark'; end;
      otherwise do; lname=' '; fname=' '; end;
   end;
   run;
```

In terms of computer efficiency and processing time, the use of the SELECT statement is roughly equivalent to the use of IF-THEN/ELSE statements. Like the IF-THEN/ELSE statements this form of look up is just not practical for more than just a few items.

Relative to the efficiency of the computer, each of these types of searches is sequential; they are applied in order. When the list is long the average number of comparisons goes up quickly, even when you carefully order the list. A number of other search techniques are available that do not require sequential searches. Binary searches operate by iteratively splitting a list in half until the target is found, and on average these searches tend to be faster than sequential searches. SAS formats use binary search techniques.

MERGES AND JOINS

Perhaps the most common way of getting the information contained in one table into another is to perform either a merge or a join. Both techniques can be useful, and of course, each has both pros and cons. The MERGE statement is used to identify two or more data sets. For the purpose of this discussion, one of these data sets will contain the information that is to be looked up. The BY statement is used to make sure that the observations are correctly aligned. The BY statement should include sufficient variables to form a unique key in all but at most one of the data sets. For our example the PATIENTS data PATID is a primary key (each PATID is unique), however in the TRIAL data some patients have return visits and thus there are duplicate PATID values.

```
proc sort data=clindat.patient
          out=patient;
   by patid;
   run;

proc sort data=clindat.trial
          out=trial;
   by patid;
   run;

data merged;
   merge patient(keep=patid lname fname ssn)
         trial;
   by patid;
   run;
```

Because the BY statement is used, the incoming data must be sorted. When the data are not already sorted, the extra step of sorting can be time consuming or even on occasion impossible for very large data sets or data sets on tape. In this example PROC SORT is used to reorder the data into temporary (WORK) data sets. These are then merged together using the MERGE statement. By using a DATA step merge, additional variables can be added. Here the SSN has been included along with the first and last names.

Sorting can be IO intensive and as a result time consuming. The sorting can be avoided through the use of any of a variety of types of SQL joins. SQL joins load both of the incoming tables into memory and the merging process is called a join.

In this example the requirement has been added that the PATID be in both data tables before the match is made. Although the use a SQL join can avoid the use of sorts, it can also cause memory and resource problems as the size of the data tables increase. A side advantage of the LEFT JOIN and the ON clause, which is shown here, is that the resultant data set is sorted; without resorting to a PROC SORT.

```
proc sql noprint;
  create table joined as
    select p.patid,p.lname,p.fname,t.dob,t.diag, t.symp
      from clindat.patient as p left join clindat.trial as t
        on p.patid=t.patid;
  quit;
```

The MERGE or SQL join will do the trick for most instances, however it is possible to substantially speed up the lookup process by using one or more of the advanced techniques shown below.

TECHNIQUES BEYOND THE BASICS

You may find that having mastered the basic techniques shown above is sufficient for your processing needs – and that is great. If however you find that your data sets and look up problems are large enough to cause your program run times to increase to an unreasonable point, then it may well be time to look into some of these more advanced techniques. Some are code intensive, but for the most part for most of them the hardest part is knowing that they exist.

USING FORMATS

Of all of the advanced techniques discussed below, the use of the user defined formats to perform the look up is probably the most practical for the largest range of problems. For most situations where the look up table has fewer than somewhere around 40,000 items this technique will provide all the muscle that you will need.

This technique uses a user defined format to make the look up assignment. Format look ups are fast because the internal search is based on a binary tree approach. This means that you can return one item out of over a hundred items with 7 or fewer queries ($2^7 = 128$), where on average IF-THEN/ELSE processing would require 50 queries.

The process of creating a format is both fast and straight forward. Formats can be built and added to a library (permanent or temporary) through the use of PROC FORMAT. Two formats (PATLNAME. and PATFNAME.) are created in this PROC FORMAT. Each contains an association between the patient number (PATID) and their first and last names.

```
proc format;
   value patlname
      1 ='Adams'
      2 ='Adamson'
      3 ='Alexander';
   value patfname
      1 ='Mary'
      2 ='Joan'
      3 ='Mark';
   run;
```

Of course typing in a few values is not a ‘big deal’, however as the number of entries increases the process tends to become tedious and error prone. Even for the example data sets used in this paper, entering the 80 or so patients in the study is just not practical.

Fortunately it is possible to build formats directly from a SAS data set. The CNTLIN= option on the PROC FORMAT statement identifies a data set that contains specific variables. These variables store the information needed to build the format, and as a minimum must include the name of the format (FMTNAME), the incoming value (START), and the value which the incoming value will be translated into (LABEL). The data step shown here builds the data set CONTROL, which is based on the look up data (CLINDAT.PATIENT). This data set is then used by PROC FORMAT to build the formats. One advantage of this technique is that the control data set does not need to be sorted within a format, however if the data contains more than one format definition, as it does here, it must at least be grouped by the format name.

```
data control(keep=fmtname start label);
   set clindat.patient(keep=patid lname fname
                       rename=(patid=start));
   * format for last name;
   fmtname='patlname';
   label=lname;
   output control;

   * Format for first name;
   fmtname='patfname';
   label=fname;
   output control;
   run;
proc sort data=control;
   by fmtname;
   run;
proc format cntlin=control;
   run;
```

```
data fmtname;
   set clindat.trial(keep=patid sex);
   lname=put(patid,patlname.);
   fname=put(patid,patfname.);
   run;
```

Once the format has been defined, the PUT function can be used to assign a value to the variable using the format. The PUT function always returns a character string; when a numeric value is required, the INPUT function can be used with an INFORMAT.

Using these formats in a DATA step to look up the first and last names will be substantially faster than the IF-THEN/ELSE or SELECT processing steps shown above. The performance improvement becomes even more dramatic as the number of items in the lookup list increases. Notice that there is only one executable statement for each look up, and the look up itself will use a format, and hence will employ a binary search.

REPLACING MERGE WITH TWO SET STATEMENTS

As was shown earlier the typical match merge requires two sorted data sets and results in a list of variables that is the union of the variable lists of the two data sets. Because this technique requires that both data sets be sorted, it may not be practical for very large data sets. Also there is a nontrivial amount of overhead used behind the scenes by the MERGE statement to coordinate the observations to make sure that they are aligned correctly. In this particular example this merge will fail unless both the data sets are sorted by PATID.

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

A DATA step with two SET statements can also be used to perform a merge like operation. Because of the overhead associated with the MERGE statement, the double SET statement merge can often be faster than the MERGE statement. However it becomes the programmer’s responsibility to take over the functionality of the merge process through coding and logic statements.

Although a DATA step with two SET statements can be used to perform a merge operation, a simple data step with just two SET statements is not sufficient to successfully complete the operation. In this simplistic two SET statement DATA step, the list of variables will be the same as in the previous step with the MERGE statement, but this is about the only similarity. Effectively this would mostly perform a one-to-one merge, however when the last observation from the shorter data set is read, the DATA step will terminate.

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

The following DATA step also uses two SET statements to perform a merge. Although no BY statement is used, this technique expects both of the incoming data sets to be sorted. In this DATA step an observation is first read from the TRIAL data set to establish the patient ID (PATID) that is to be looked up (notice that a rename option is used). The lookup list is then read sequentially until the codes are equal and the observation is written out. Although not shown here, logic can be included to handle observations that are in one data set and not the other. One restriction of the code, as it is shown here, is that any duplicate patient codes would have to be in the TRIAL data set. This code expects the lookup data (PATIENT) to have unique values of PATID.

```
proc sort data=clindat.patient out=patient;
   by patid;
   run;
proc sort data=clindat.trial out=trial;
   by patid;
   run;

data doubleset(drop=code);
  set trial(keep=patid symp dt_diag
            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 patient(keep=patid lname fname);
    if code=patid then output;
  end;
  run;
```

Although the sorting restrictions are the same as when you use the MERGE statement, the advantage of the double SET can be a substantial reduction in processing time. This improved performance does however come at a cost. The code and logic is more complicated and more prone to error.

USING INDEXES

Indexes are a way to logically sort your data without physically sorting it. While not strictly a lookup technique, if you find that you are sorting and then resorting data to accomplish your various merges, you may find that indexes will be helpful.

Indexes must be created, stored, and maintained. They are most usually created through either PROC DATASETS (shown here) or through PROC SQL. The index stores the order of the data had it 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 physically sorted. Indexes are named and one variable (simple) indexes have the same name as the variable forming the index. In this example CLINDAT.PATIENT has a single simple index on PATID. An index formed by two or more variables is known as a composite index. Although the composite index is named, the name itself is not really used. Here a composite index named KEYNAME is created for the CLINDAT.TRIAL data set.

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

Obviously there are some of the same limitations to indexes that you encounter when sorting large data sets. Resources are required to create the index, and these can be similar to the SORT itself. The indexes are 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, however they can also SLOW things down (Virgle, 1998). Be sure to read and experiment carefully before investing a lot of effort in the use of indexes. The topic of indexes can be complex enough that an entire book has been written on the subject (Raithel, 2006).

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

Although neither of the two data sets that we have been working with have been sorted, they have now been indexed and we can now perform a merge using the BY statement through the use of their indexes. Because the index information is stored as a part of the data set's meta data, the index is detected and utilized when the data set is used with a BY statement. The merge takes place as if the data sets were sorted and the programmer has to do nothing in order for the indexes to be utilized.

USING THE KEY= OPTION

It is often not practical to create an index for both data sets. If one of the data sets is unindexed you can still perform a look up operation using the index on the other data set. In this example we will assume that the master data set (CLINDAT.TRIAL) is unindexed. As you examine this DATA step you will notice that this is essentially a two SET statement merge. However an index is used instead of logic to coordinate the reads of the observations.

The KEY= option on the SET statement option identifies an index that is to be used for reading that data set. In this example the data set CLINDAT.PATIENT has an index for PATID. As each observation is read from the master data set (CLINDAT.TRIAL), a value of PATID is loaded into the PDV. This value is then used in the second SET statement to read an observation with a matching PATID from CLINDAT.PATIENT. This retrieval is fast because of the PATID index.

```
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;
```

When an indexed read is performed using the KEY= option, a return code indicating the success or failure of the read is stored in the temporary variable _IORC_. This variable will be equal to 0 when the index value is found. Because the variables that we are retrieving (LNAME and FNAME) are retained variables, they must be set to missing when a particular value of PATID does not exist in the indexed data set (_IORC_ will not be 0).

USING ARRAYS FOR KEY-INDEXING

Sometimes when the use of indexes or sorting is not an option, or when you just want to speed up a search, the use of arrays can be just what you need. Under the current versions of SAS you can build arrays that can contain millions of values (Dorfman, 2000a, 2000b). However arrays are not always an option. An array can contain a single value in each cell and the index to the array must be numeric. If these are not limitations for your particular problem, there are no faster look up techniques than those that utilize key-indexing techniques to access arrays.

When we use the term key-indexing we are not referring to a data set index as was described in the previous section, but rather the term refers to the way that the array is accessed. In key-indexing the array is accessed by a variable that is used in the look up process. This variable is used as the array index. By utilizing arrays in this manner, we can go directly to the value of interest rather than searching for it through a list one item at a time.

A simple example of key-indexing can be demonstrated through the problem of selecting unique values from a data set. In terms of our data sets, we would like to make sure that the patient codes in the data set CLINDAT.PATIENT are unique – that the data set has at most one observation for each value of PATID. One solution for this type of problem would be to use PROC SORT with the NODUPKEY option as is done here. However an alternative to sorting is to use an array and key-indexing techniques.

```
proc sort data=clindat.patient
          out=patient
          nodupkey;
   by patid;
   run;
```

To avoid sorting, we somehow have to "remember" for any given PATID whether or not it has already been found. The way to do this is to use an array. There are a couple of ways to use an array to 'remember' the values of the PATID variable that have been encountered. The first is a brute force approach that does not use key-indexing. Here we store each new PATID value in the next available array element. The first PATID found is stored in CHECK{1} and the next PATID that is different from the first is stored in CHECK{2}. When we want to see if a given PATID has already been found we must check all the elements of the array, which we can do using the WHICHN function. The primary limitation of this approach is in the way that we search across the array. As the size of the array increases, and the array has to be as large as the number of distinct PATID values, the search slows down. This limitation is avoided by the key-indexing technique shown next.

```
data Brute;
   array check {10000} _temporary_;
   retain foundcnt 0;
   set clindat.patient;
   * Check if already found;
   if whichn(patid,of check{*})=0 then do;
      * First time this PATID;
      foundcnt+1;
      check{foundcnt} = patid;
      output brute;
   end;
   run;
```

The beauty of the key-indexing technique is that the search is very quick because regardless of the size of the array, only one item has to be checked. We accomplish this by using the PATID code itself as the index to the array. As an observation is read from the incoming data set, the numeric PATID code is used as the index for the array CHECK. If the array value is missing, this is the first occurrence of this PATID. The array element is then marked as found (the value is set to 'x'). Notice that this particular array will allow a range of PATID values from 1 to 10,000. Larger ranges, into the 10s of millions, are easily accommodated.

```
data unique;
   array check {10000} $1 _temporary_;
   set clindat.patient;
   * check if this patient has been
   * found before;
   if check{patid}=' ' then do;
      * First occurance for this patient;
      output unique;
      * mark this patient as found;
      check{patid}='x';
   end;
   run;
```

This process of looking up a value is exactly what we do when we merge two data sets. In this DATA step the list of patient identification codes are read into an array. The order that the values are read in does not matter as each value is loaded using the PATID as an index. Since in this example we are looking up two values (LNAME and FNAME) two arrays have been created. In both arrays the PATID will be used as the array subscript (index).

The second DO UNTIL then reads the data set of interest. Again the order that this data set is read makes no difference. In this loop the values of LNAME and FNAME are recovered from the arrays, again using the PATID as the array index, and assigned to the appropriate variables.

Each observation from the two incoming data sets is read exactly once. Neither data set need to be sorted. The two arrays together take up only about 1.6 million bytes of memory, and this is fairly small for modern machines.

```
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;
```

This technique is known as Key-indexing because the index of the array is the value of the variable that we want to use as the look up value. Unfortunately this technique will not work in all situations. As the number of array elements increases the amount of memory used also increases (Paul Dorfman, 2000a, discusses memory limitations). Certainly most modern machines should accommodate arrays with the number of elements in the millions. For situations where this technique requires unreasonable amounts of memory, other techniques such as bitmapping and hashing are available. Again Paul Dorfman is the acknowledged expert in this area and his cited papers should be consulted for more details.

Other limitations of this technique as well as solutions to these limitations are discussed in the section on hash objects below.

USING HASH OBJECTS

Limitations of KEY-INDEXING, and the use of arrays in general, include the necessity to use a numeric value as the array index, the inconvenience of multidimensional arrays, the inability to mix types of variables stored in a single array, fixed dimensionality of the array, and the inability to store more than one item in a given array position. While techniques have been developed to work around these limitations, the hash object directly addresses and negates each of these issues.

You can think of the hash object as a super array. The values stored in a hash table can be loaded with data in one of several ways. One or more index variables can be specified and these can be either numeric or character. Essentially the size of a hash table is dynamically allocated, so you do not need to know either the number of elements to be stored or the number of bytes needed to store an item ahead of time. Hash tables are defined and used in a DATA step, and a given DATA step can have as many hash objects defined as necessary. Since these tables are stored in memory it is possible to fill the available memory allocated to SAS, but like with arrays, enough memory is generally available to hold very large amounts of information.

Within the DATA step, the hash object must first be defined (declared) before it can be used. Rather than address the object directly like we do with arrays, a series of predefined tools have been written for us to use. Known as methods and constructors, these tools are used, among other things, to load data into and to retrieve values out of the hash table. Since the declaration process utilizes variables on the PDV, we have to make sure that these variables exist prior to the declaration process. In this example the variables that are to be used in the hash object are manually added to the PDV using the LENGTH statement. The hash object itself is then defined once (_N_=1) and named (HMERGE) using the DECLARE statement. The attributes of the object are then defined using the DEFINEKEY (the look up variable), DEFINEDATA (the values stored in the array and to be retrieved), and DEFINEDONE (closes the DECLARE block) methods. The DATASET: constructor loads the selected variables of the incoming table (CLINDAT.PATIENT) into the hash object HMERGE.

```
data hashmerge(keep=patid lname fname symp diag);
   length patid 8 lname $10 fname $6;
   if _n_=1 then do;
      declare hash hmerge(dataset: 'clindat.patient',
                          hashexp: 6);
         rc1 = hmerge.defineKey('patid');
         rc2 = hmerge.defineData('lname', 'fname');
         rc3 = hmerge.defineDone();
   end;
   set clindat.trial end=done;
   rc4 = hmerge.find();
   if rc4 = 0 then output hashmerge;
   run;
```

For each observation and its associated PATID value in the data set CLINDAT.TRIAL, the FIND method uses the value of PATID and retrieves (looks up) the corresponding values LNAME and FNAME in the hash table and loads these values into the PDV. The values stored in the PDV can then be written out to the new data set HASHMERGE using the DATA statement.

The previous example works fine, but it is not as efficient as it could be. And because the variables LNAME and FNAME are never directly read into the PDV, the LENGTH statement causes UNINITIALIZED variable notes in the LOG. In this version of this same DATA step we overcome both of these limitations. The LENGTH statement is replaced with a compilation only SET statement (during execution IF 0 is false). This loads the variables LNAME and FNAME onto the PDV. Technically we do not need to add PATID to the PDV, since during compilation it will be added from the SET statement that names CLINDAT.TRIAL as an incoming data set.

```
data hashmerge(keep=patid lname fname symp diag);
   if 0 then set clindat.patient(keep=lname fname);
   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;
```

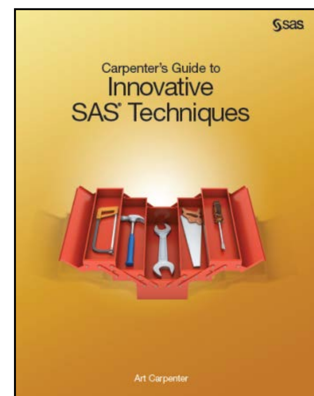
The incoming SET statement for CLINDAT.TRIAL is now inside of a DO UNTIL loop. This improves the efficiency of the read, and it also means that we no longer need to conditionally execute the DECLARE block. The STOP statement is used to terminate the DATA step. Whenever you use a DO loop to surround a SET statement it is a good idea to also use a STOP statement to close the DATA step.

SUMMARY

There are a number of techniques that can be applied whenever you need to “look up” a value in another table. Each of these techniques has both pros and cons and as SAS programmers we must strive to understand the differences between them. As our data tables become larger or as our retrievals (look ups) become more complex, it becomes increasingly more important that we understand and utilize those techniques that are most offer the best efficiencies. Since none of these techniques is appropriate for all situations, it is incumbent that we have a good grasp of each of these methods so that we can choose the appropriate one. Some of the commonly applied techniques, such as IF-THEN/ELSE, MERGEs and JOINS have alternate methods that can be used to improve performance and may even be required under some conditions.

REFERENCES

Many of the examples in this paper are adapted from *Carpenters Guide to Innovative SAS® Techniques*. https://support.sas.com/pubscat/bookdetails.jsp?pc=62454



Other references include:

Aker, Sandra Lynn, 2000, “Using KEY= to Perform Table Look-up”, published in the conference proceedings for: SAS Users Group International, SUGI, April, 2000.

Carpenter, Arthur L., 1999, “Getting More For Less: A Few SAS® Programming Efficiency Issues”, published in the conference proceedings for: Northeast SAS Users Group, NESUG, October, 1999; Pacific Northwest SAS Users Group, PNWSUG, June, 2000; Western Users of SAS Software Inc., WUSS, September, 2000.

Carpenter, Arthur L., 2001, “Table Lookups: From IF-THEN to Key-Indexing,” presented at the Ninth Western Users of SAS Software Conference (September, 2001) and the Twenty-Sixth Annual SAS Users Group International Conference, SUGI, (April, 2001), and the Pacific Northwest SAS Users Group Conference (November, 2005). The paper was published in the proceedings for each of these conferences. http://www2.sas.com/proceedings/sugi26/p158-26.pdf

Carpenter, Art, 2012, *Carpenter’s Guide to Innovative SAS Techniques*, Cary, NC: SAS Institute Inc.

Dorfman, Paul, 1999, “Alternative Approach to Sorting Arrays and Strings: Tuned Data Step Implementations of Quicksort and Distribution Counting”, published in the conference proceedings for: SAS Users Group International, SUGI, April, 1999.

Dorfman, Paul, 2000a, "Private Detectives In a Data Warehouse: Key-Indexing, Bitmapping, And Hashing", published in the conference proceedings for: SAS Users Group International, SUGI, April, 2000.

Dorfman, Paul, 2000b, "Table Lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing", published in the conference proceedings for: Pacific Northwest SAS Users Group, PNWSUG, June, 2000.

Raithel, Michael, 2006, *The Complete Guide to SAS Indexes*, Cary, NC: SAS Institute Inc

Virgle, Robert, 1998, *Efficiency: Improving the Performance of Your SAS® Applications*, Cary, NC: SAS Institute Inc., 256pp.

ACKNOWLEDGMENTS

Paul Dorfman, Mike Raithel, and Bob Virgle are internationally acknowledged experts in the fields of SAS programming efficiencies and large data set techniques. Through conversations and their written work, each has provided me with valuable insights into the techniques used here, and it has been a pleasure to continue to learn from them.

ABOUT THE AUTHOR

Art Carpenter's publications list includes two chapters in *Reporting from the Field*, five books, and numerous papers and posters presented at SAS Global Forum, SUGI, PharmaSUG, WUSS, and other regional conferences. Art has been using SAS since 1977 and has served in various leadership positions in local, regional, national, and international user groups.

Art is an Advanced SAS Certified Professional, 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) 86567
art@caloxy.com
www.caloxy.com

View my paper presentations page at:
http://www.sascommunity.org/wiki/Presentations:ArtCarpenter_Papers_and_Presentations

TRADEMARK INFORMATION

SAS and SAS Certified Professional are registered trademarks of SAS Institute, Inc. in the USA and other countries.
® indicates USA registration.