Table Lookups: From IF-THEN to Key-Indexing

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.

This workshop will demonstrate, compare, and contrast the following techniques:
IF-THEN, IF-THEN-ELSE, SELECT, user defined formats (PUT function), MERGE, SQL joins, merging with two SET statements, using the KEY= option, use of ARRAYS, and Key-Indexing.

KEYWORDS

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

INTRODUCTION

A table lookup is performed when you use the value of a variable (*e.g.* a part code) to determine the value of another variable (*e.g.* part 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 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, 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 in 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 primary discussion in this workshop will be to compare and contrast the more common methods and then to look at some of the simpler of the specialty techniques.

THE DATA

Two data sets will used throughout this workshop. The first contains the lookup information which is a list of automobile part codes and the name of the associated parts (LOOKUP.CODES). The second is a list of number of orders for the respective parts. The order information contains the part code but not the part name (LOOKUP.PARTORDR). Neither data set is sorted, nor is there any guarantee that all ordered parts will be in the CODES data set. The PARTORDR data set may also have duplicate observations.

```
Part Codes and Associated Names
LOOKUP.CODES

Obs    partname            partcode
 1     Hubcap                 1
 2     Wheel Cover            2
 3     Tire Type              3
 4     Valve stem             5
 5     Tail Light, Left      47
```

```
 6     Rim                    4
 7     Generator             41
 8     Wiring Harness        42
 9     Fuse Assembly         43


Part Ordering Information
LOOKUP.PARTORDR

Obs    partcode    ordernum
 1         1           1
 2         2          21
 3         3           4
 4         5           7
 5         4           7
 6        49           8
 7        42          11
 8        42          11
 9        43           4
```

The assumption throughout this paper will be that the list of parts orders (PARTORDR) can not be fully processed with out the appropriate part name. This means that we somehow need to add the PARTNAME variable and determine its value.

LOGICAL PROCESSING

One of the easiest solutions to the lookup problem is to ignore it and just create the new variable with its associated value through the use of IF-THEN processing. Effectively we are '*hard coding*' the part name within the program. This technique is demonstrated in the following DATA step.

```
* Use IF-THEN processing to assign partnames;
data withnames;
set lookup.partordr;
if partcode=1 then partname='Hubcap           ';
if partcode=  2 then partname='Wheel Cover    ';
if partcode=  3 then partname='Tire Type      ';
if partcode=  5 then partname='Valve stem     ';
if partcode=  4 then partname='Rim            ';
if partcode= 41 then partname='Generator      ';
if partcode= 42 then partname='Wiring Harness ';
if partcode= 43 then partname='Fuse Assembly  ';
run;
```

The problem with this approach is that it is not practical if there are more than a very few codes to lookup. Besides this is VERY inefficient. SAS must execute each IF statement even if an earlier IF statement was found to be true. To make matters worse, IF statements require a fair bit of processing time.

A substantially faster method is to use the IF-THEN-ELSE statement combination. The following DATA step executes more quickly than the previous one because as soon as one IF statement is found to be true, its ELSE is not executed consequently none of the following IF-THEN-ELSE statements are executed. This technique is fastest if the more likely outcomes are placed earlier in the list.

```
* Use IF-THEN-ELSE processing to assign partnames;
data withnames;
set lookup.partordr;
if partcode= 1 then partname='Hubcap          ';
else if partcode= 2 then partname='Wheel Cover';
else if partcode= 3 then partname='Tire Type  ';
else if partcode= 5 then partname='Valve stem ';
else if partcode= 4 then partname='Rim        ';
else if partcode=41 then partname='Generator  ';
else if partcode=42 then
                     partname='Wiring Harness';
else if partcode=43 then
                     partname='Fuse Assembly';
run;
```

The SELECT statement is on par with the IF-THEN-ELSE combination when performing table lookups (actually it might be a bit faster - Virgle, 1998). Again processing time is minimized when the most likely match is located early in the list.

```
* Use SELECT() processing to assign partnames;
data withnames;
set lookup.partordr;
select(partcode);
 when( 1) partname='Hubcap           ';
 when( 2) partname='Wheel Cover      ';
 when( 3) partname='Tire Type        ';
 when( 5) partname='Valve stem       ';
 when( 4) partname='Rim              ';
 when(41) partname='Generator        ';
 when(42) partname='Wiring Harness   ';
 when(43) partname='Fuse Assembly    ';
 otherwise;
end;
run;
```

Interestingly Virgle (1998) has found that the efficiency of the SELECT statement can be enhanced by placing the entire expression on the WHEN statement.

```
* Use SELECT processing to assign partnames;
data withnames;
set lookup.partordr;
select;
 when(partcode= 1) partname='Hubcap           ';
 when(partcode= 2) partname='Wheel Cover      ';
 when(partcode= 3) partname='Tire Type        ';
 when(partcode= 5) partname='Valve stem       ';
 when(partcode= 4) partname='Rim              ';
 when(partcode=41) partname='Generator        ';
 when(partcode=42) partname='Wiring Harness   ';
 when(partcode=43) partname='Fuse Assembly    ';
 otherwise;
end;
run;
```

The primary problem with each of these techniques is that the search is sequential. 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. On average these searches tend to be faster than sequential searches. SAS formats use binary search techniques.

USING FORMATS

Formats can be built and added to a library (permanent or temporary) through the use of PROC FORMAT. The process of creating a format is both fast and straight forward. The following format (PARTNAME.) contains an association between the part code and its name.

```
* Building a Format for partnames;
proc format;
value partname
  1='Hubcap        '
  2='Wheel Cover   '
  3='Tire Type     '
  5='Valve stem    '
  4='Rim           '
 41='Generator     '
 42='Wiring Harness'
 43='Fuse Assembly ';
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. Fortunately it is possible to build a format directly from a SAS data set. The CNTLIN= option 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 too (LABEL). The following data step builds the data set CONTROL which is used by PROC FORMAT. Notice the use of the RENAME= option and the RETAIN statement. One advantage of this technique is that the control data set does not need to be sorted.

```
* Building a Format from a data set;
data control(keep=fmtname start label);
 set lookup.codes(rename=(partcode=start partname=label));
 retain fmtname 'partname';
```

```
 run;

proc format cntlin=control;
run;
```

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

```
* Using the PUT function.;
data withnames;
 set lookup.partordr;
 partname = put(partcode,partname.);
 run;
```

The previous data step will be substantially faster than the IF-THEN-ELSE or SELECT processing steps shown above. The difference becomes even more dramatic as the number of items in the lookup list increases. Notice that there is only one executable statement rather than one for each comparison. The lookup itself will use the format PARTNAME., and hence will employ a binary search.

MERGES AND JOINS

Another 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 CODES data has exactly one observation for each value of PARTCODE.

Because the BY statement is used, the data must be sorted. When the data are not already sorted, this extra step can be time consuming or even on occasion impossible for very large data sets or data sets on tape. In the following steps PROC SORT is used to reorder the data into temporary (WORK) data sets. These are then merged using the MERGE statement.

```
* Using the MERGE Statement;
proc sort data=lookup.codes
          out=codes;
  by partcode;
  run;

proc sort data=lookup.partordr
          out=partordr;
  by partcode;
  run;

data withnames;
  merge partordr codes;
  by partcode;
  run;

proc print data=withnames;
  title
  'Part Orders with Part Names After the Merge';
  run;
```

The following listing of the merged data shows that two part codes (41 and 47) are in the codes list but did not have any orders (absent from the data set PARTORDR). Also part code 49 was ordered, but because it is not on the parts list we do not know what part was actually to be ordered.

```
Part Orders with Part Names After the Merge

Obs    partcode    ordernum    partname
  1        1           1       Hubcap
  2        2          21       Wheel Cover
  3        3           4       Tire Type
  4        4           7       Rim
  5        5           7       Valve stem
  6       41           .       Generator
  7       42          11       Wiring Harness
  8       42          11       Wiring Harness
```

```
 9       43           4       Fuse Assembly
10       47           .       Tail Light, Left
11       49           8
```

It is also possible to use SQL to join these two data tables. When using SQL, the merging process is called a join. There are a number of different types of joins within SQL, and one that closely matches the previous step is shown below.

```
* Using the SQL Join;
proc sql;
create table withnames as
 select *
  from lookup.partordr a, lookup.codes b
   where a.partcode=b.partcode;
quit;
```

In this example we have added the requirement that the code be in both data tables before the match is made. Notice that SQL does not require the two data sets to be sorted prior to the join. This can avoid the use of SORT, but it can also cause memory and resource problems as the data table sizes increases.

```
Part Orders with Part Names Using SQL Join

Obs    partcode    ordernum    partname
 1         1           1       Hubcap
 2         2          21       Wheel Cover
 3         3           4       Tire Type
 4         5           7       Valve stem
 5         4           7       Rim
 6        42          11       Wiring Harness
 7        42          11       Wiring Harness
 8        43           4       Fuse Assembly
```

As in the above SQL step, you can also select only those observations (PARTCODE) that meet certain criteria when using the MERGE. The IN= option creates a temporary numeric variable that is either true or false (1 or 0) depending on whether the current observation (for the current value of the BY variables) is in the specified data set. This allows you to use IF-THEN-ELSE logic to eliminate or include observations that meet specific criteria. In the example below, only observations that contain a part code that is in both data sets will contribute to the new data set.

```
* Selecting only existing values;
data withnames;
  merge partordr(in=inordr)
        codes(in=incodes);
  by partcode;
  if inordr & incodes;
  run;

proc print data=withnames;
  title
  'Part Orders with Part Codes After the Merge';
  run;
```

Notice that this data set is very close to the one generated by the SQL step, and only really differs by the order of the observations.

```
Part Orders with Part Codes After the Merge

Obs    partcode    ordernum    partname
 1         1           1       Hubcap
 2         2          21       Wheel Cover
 3         3           4       Tire Type
 4         4           7       Rim
 5         5           7       Valve stem
 6        42          11       Wiring Harness
 7        42          11       Wiring Harness
 8        43           4       Fuse Assembly
```

The MERGE or SQL join will do the trick for most instances, however it is possible to substantially speed up the lookup process.

The following DATA step uses two SET statements to perform the merge. First an observation is read from the PARTORDR data set to establish the part code 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. As in the previous MERGE examples, logic can be included to handle observations that are in one data set and not the other.

```
* Merging without a BY or MERGE;
data withnames;
  set partordr(rename=(partcode=code));
  * The following expression is true only when
  * the current CODE is a duplicate.;
  if code=partcode then output;
  do while(code>partcode);
    * lookup the name using the code
    * from the primary data set;
    set codes(keep=partcode partname);
    if code=partcode then output;
  end;
  run;

proc print data=withnames;
title 'Part Orders and Codes Double SET Merge';
run;
```

The merged data set shows the following:

```
Part Orders and Codes Double SET Merge

Obs  code  ordernum   partcode   partname
 1     1       1          1      Hubcap
 2     2      21          2      Wheel Cover
 3     3       4          3      Tire Type
 4     4       7          4      Rim
 5     5       7          5      Valve stem
 6    42      11         42      Wiring Harness
 7    42      11         42      Wiring Harness
 8    43       4         43      Fuse Assembly
```

As in the MERGE shown earlier, the data still have to be sorted before the above DATA step can be used. 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.

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 below) 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 sorted.

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 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. They can also SLOW things down (Virgle, 1998). Be sure to read and experiment carefully before investing a lot in the use of indexes.

The following example shows the creation of indexes for the two data sets of interest in this workshop. A merge is then performed on the unsorted (but indexed) data sets using a BY statement.

```
proc datasets library=lookup;
  modify codes;
    index create partcode / unique;
  modify partordr;
    index create partcode;
run;

* Use the BY without sorting;
data indexmerge;
  merge lookup.partordr(in=inordr)
```

```
      lookup.codes(in=incodes);
  by partcode;
  if inordr & incodes;
  run;
```

USING THE KEY= OPTION

You can also lookup a value when an index exists on only the data set that contains the values to be looked up. The KEY= SET statement option identifies an index that is to be used. In the following example we want to lookup the PARTNAME in the CODES data set, which is indexed on PARTCODE.

```
data keylookup;
   set lookup.partordr; *Master data;
   set lookup.codes key=partcode/unique;
   if _iorc_>0 then partname=' ';
run;
```

A PROC PRINT of KEYLOOKUP shows:

```
Merge using the KEY= Option

Obs    partcode    ordernum    partname
 1         1           1       Hubcap
 2         2          21       Wheel Cover
 3         3           4       Tire Type
 4         5           7       Valve stem
 5         4           7       Rim
 6        49           8
 7        42          11       Wiring Harness
 8        42          11       Wiring Harness
 9        43           4       Fuse Assembly
```

An observation is read from the primary data set (PARTORDR) which contains a value of PARTCODE. This value is then used to seek out an observation in the CODES data.

The automatic variable _IORC_ is 0 when the search is successful. PARTCODE 49 is not found in the CODES data, so for this value _IORC_ is greater than 0 and a missing value is assigned to PARTNAME.. If the _IORC_ is not checked PARTCODE 49 will be given a PARTNAME from the last successful read of CODES and this is almost always not what you want.

The autocall macro %SYSRC can be used to 'decode' the values contained in _IORC_. The following example is the same as the previous one, but it takes advantage of two of over two dozen values accepted by %SYSRC.

```
data rckeylookup;
   set lookup.partordr;
   set lookup.codes key=partcode;
   select (_iorc_);
      when (%sysrc(_sok)) do;
         * lookup was successful;
         output;
      end;
      when (%sysrc(_dsenom)) do;
         * No matching partcode found;
         partname='Unknown';
         output;
      end;
      otherwise do;
         put  'Problem with lookup ' partcode=;
         stop;
      end;
   end;
run;
```

USING ARRAYS FOR KEY-INDEXING

Sometimes when 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 current versions of SAS you can build arrays that can contain millions of values (Dorfman, 2000a, 2000b).

Consider the problem of selecting unique values from a data set. In terms of our data sets we would like to make sure that the data set LOOKUP.PARTORDR has at most one observation for each value of PARTCODE. One solution would be to use PROC SORT as is done below.

```
proc sort data=lookup.partordr
                out=partordr nodupkey;
   by partcode;
```

This solution of course requires sorting. SQL with the UNIQUE function could also be used, but, as was mentioned above, this technique also may have problems.

To avoid sorting we somehow have to "remember" what part codes we have already seen. The way to do this is to use the ARRAY statement. The beauty of this technique is that the search is very quick because it only has to check one item. We accomplish this by using the part code itself as the index to the array.

```
* Selecting Unique Observations;
data unique;
  array check {10000} _temporary_;
  set lookup.partordr;
  if check{partcode}=. then do;
     output;
     check{partcode}=1;
  end;
  run;

proc print data=unique;
title 'Unique Part Orders';
run;
```

As an observation is read from the incoming data set, the numeric part code is used as the index for the ARRAY CHECK. If the array value is missing, this is the first (unique) occurrence of this part code. It is then marked as found (the value is set to 1). Notice that this step will allow a range of part codes from 1 to 10,000. Larger ranges, into the 10s of millions, are easily accommodated. The listing for the data set UNIQUE shows:

```
Unique Part Orders

Obs    partcode    ordernum
 1         1           1
 2         2          21
 3         3           4
 4         5           7
 5         4           7
 6        49           8
 7        42          11
 8        43           4
```

This process of looking up a value is exactly what we do when we merge two data sets. In the following DATA step the list of codes are read sequentially, ONCE, into an array that stores the part name using the part code as the array subscript. The second DO UNTIL then reads the data set of interest. In this loop the part name is recovered from the array and assigned to the variable PARTNAME.

```
data withnames;
  array chkname {10000} $15 _temporary_;
  do until(allnames);
     set lookup.codes end=allnames;
     chkname{partcode}=partname;
  end;
  do until(allordr);
     set lookup.partordr end=allordr;
     partname = chkname{partcode};
     output;
  end;
  stop;
  run;

proc print data=withnames;
title 'Part Orders - Double SET with Arrays';
run;
```

The listing below shows that each observation from the PARTORDR data set has been included in the resulting data set and that sorting was not necessary for EITHER data set.

```
Part Orders - Double SET with Arrays

Obs    partname          partcode    ordernum
 1     Hubcap               1           1
 2     Wheel Cover          2          21
 3     Tire Type            3           4
 4     Valve stem           5           7
 5     Rim                  4           7
 6                         49           8
 7     Wiring Harness      42          11
 8     Wiring Harness      42          11
 9     Fuse Assembly       43           4
```

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 lookup value.

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.

SUMMARY

There are a number of techniques that can be applied whenever you need to lookup a value in another table. Each of these techniques has both pros and cons and as programmers we must strive to understand the differences between these techniques. Some of the commonly applied techniques (IF-THEN-ELSE and the MERGE) have alternate methods that can be used to improve performance and may even be required under some conditions.

REFERENCES

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.

Paul Dorfman, 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.

Paul Dorfman, 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.

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

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

ACKNOWLEDGMENTS

Paul Dorfman and Bob Virgle are internationally acknowledged experts in the fields of SAS programming efficiencies and large data set techniques. Both 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*, three books, and numerous papers and posters presented at SUGI, PharmaSUG, NESUG, and WUSS. Art has been using SAS since 1976 and has served in various leadership positions in local, regional, and national user groups.

Art is a 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
P.O. Box 430
Vista, CA 92085-0430

(760) 945-0613
art@caloxy.com
www.caloxy.com

TRADEMARK INFORMATION