PaperA03-2007

Table Lookups...You Want Performance?

Rob Rohrbough, Rohrbough Systems Design, Inc.

Presented to the *Midwest SAS® Users Group*
Monday, October 29, 2007
Paper Number A3

Abstract

Over the years SAS Software has provided many ways of accessing data related to a table or file. They range from match merges, to SQL joins and the use of indices, to formats (yes, formats), to the hash object. Each serves its own purpose in different contexts. This paper will give you a review of many of them with examples and syntax, and it will discuss their advantages and disadvantages. We will look at performance considerations and discuss the new Hash Object introduced to the Data Step in SAS 9.1.3.

Introduction

One of the most common issues in dealing with data is associating values in a relatively short list with each record in a large dataset. We typically call this "table lookup". On the surface, the problem looks easy – especially for users familiar with the ubiquitous SQL language. However, when we try to optimize table lookups for a sizable database the performance isn't always what we desire. SAS gives us a myriad of solutions to choose from. As with all performance issues, your "mileage" may vary. So, we attempt here to illustrate several techniques and compare their performance in a totally artificial environment – with apologies and encouragement to pick one or two to try in your own environment.

This paper covers the following techniques:

1. Join with PROC SQL – a left join that will retain target records having no lookup match.
2. Data Step Merge – of the target and lookup files.
3. Array Lookup – with temporary data step arrays and a "DOW" loop.
4. Format Lookup – with data step initialization using "flex code".
5. Format Lookup – with PROC FORMAT initialization.
6. Manual Hash
7. Lookup with the SAS9 Hash Object

All seven techniques solve the same problem with the same data. In the discussion below we will have brief exposure to the items in quotes above.

The Problem and Solutions

We have a list of hospital visits, in which patients do not have their city or state listed. They do have their zip codes and we have a file of zip codes with city names. We would like to summarize by city and/or state. However, we want to account for those patients where there is no matching zip code. The visit history file (50MB) has 63,022 records and the zip code lookup file (2MB) has 29,470 records.

We want to compare performance and choose the best alternative for this situation. You may have situations for which the best solution is not the one shown to be most efficient here. For instance, you may have data sets whose sizes differ from this example. A much smaller lookup data set may perform better with the techniques that bring the lookup into memory. As we will see here, some solutions may perform much differently than we would expect on our own intuition.

Join with PROC SQL

The simplest is sometimes the best. SAS Institute continues to work on optimization of PROC SQL. In many cases it can give performance comparable to or exceeding a data step. Here is a SAS log snippet (from SAS v 8.2) that looks up city and state based on zip for our visit file:

```
29726  * Join with proc SQL;
29727  proc sql;
29728    create table TabLook.bothSQL as
29729      select distinct a.*, b.City, b.ST
29730      from TabLook.Visit a left join TabLook.Zip b
29731      on a.PTzip = b.zip
29732      ;
NOTE: Table TABLOOK.BOTHSQL created, with 63022 rows and 129 columns.

29733  quit;
NOTE: PROCEDURE SQL used:
      real time           35.64 seconds
      cpu time            4.99 seconds
```

In this example we create a table that is an outer join of the visit file and the zip code lookup file. We specify a “left join” since we do not want to lose any visit records in which the zip code is not valid. Also, we do not account for duplicates. The list of zip codes we used does contain a few duplicates as we shall see later. Notice that our visits did not hit any of them as evidenced by the same number of rows (63022) in our result dataset, bothSQL, as in our source, Visit, dataset.

Data Step Merge

Faster in most cases – not always dramatic as this (also with v8.2):

```
29740  * Data Step Merge;
29741  proc sort data=TabLook.Visit out=TabLook.VisitSort;
29742    by PtZip;

NOTE: There were 63022 observations read from the data set TABLOOK.VISIT.
NOTE: The data set TABLOOK.VISITSORT has 63022 observations and 127 variables.
NOTE: PROCEDURE SORT used:
      real time           6.89 seconds
      cpu time            1.15 seconds


29743  proc sort data=TabLook.Zip out=TabLook.ZipSort;
29744    by zip;

NOTE: There were 29470 observations read from the data set TABLOOK.ZIP.
NOTE: The data set TABLOOK.ZIPSORT has 29470 observations and 8 variables.
NOTE: PROCEDURE SORT used:
      real time           0.29 seconds
      cpu time            0.09 seconds


29745  data TabLook.BothMerge;
29746    merge TabLook.VisitSort (in=a)
29747          TabLook.ZipSort  (in=b rename=zip=PtZip keep=zip City ST);
29748    by PtZip;
29749    if a;
29750  run;

NOTE: There were 63022 observations read from the data set TABLOOK.VISITSORT.
NOTE: There were 29470 observations read from the data set TABLOOK.ZIPSORT.
NOTE: The data set TABLOOK.BOTHMERGE has 63022 observations and 129 variables.
NOTE: DATA statement used:
      real time           1.53 seconds
      cpu time            0.35 seconds
```

Note that this example still does not account for duplicates.

Array Lookup

If you have a short lookup, this technique may be efficient. However, if your lookup table is significant as it is here, this can be extraordinarily slow (also shown with 8.2):

```
29755  data TabLook.BothArray (keep=from--mrreas10 City ST);
29756    array Zips(30000) $5 ;
29757    array Cities(30000) $17;
29758    array States(30000) $2;
29759    retain z i (0 0);
29760    do until (lasta);
29761      set TabLook.zip end=lasta;
```

```
29762      z + 1;
29763      Zips(z) = Zip;
29764      Cities(z) = City;
29765      States(z) = ST;
29766    end;  * until;
29767
29768    do until (lastb);
29769      set TabLook.Visit end=lastb;
29770      City = '';
29771      ST   = '';
29772      do i = 1 to z;
29773        if PtZip = Zips(i) then do;
29774          City = Cities(i);
29775          ST   = States(i);
29776          leave;  * Choose first if duplicates;
29777        end;  * if;
29778      end;  * do i;
29779      output;
29780    end;  * do until;
29781    stop;
29782  run;

NOTE: There were 29470 observations read from the data set TABLOOK.ZIP.
NOTE: There were 63022 observations read from the data set TABLOOK.VISIT.
NOTE: The data set TABLOOK.BOTHARRAY has 63022 observations and 129 variables.
NOTE: DATA statement used:
      real time           5:38.85
      cpu time            5:35.93
```

Note that this example, almost by default, takes the first of any duplicate zip code. Also, we chose an array size of 30,000 because we knew that the zip code table contained only 29,470 values. If we did not know how many zip code values there were or we were going to repeat this lookup with potentially different numbers of zip values, we could use the NOBS= option in a prior data step to place the exact count in a macro variable that we could then use to precisely define the array sizes every time.

As for the lack of efficiency, there are search techniques, e.g., binary search, that we could use to speed the array lookup DO loop. However, with SAS features such FORMATs and the Hash Object, there is very little reason to hand-code such a search technique.

Format Lookup - PROC FORMAT flex code

This technique is one of the author's personal favorites (pre-SAS9). It is a little more code, but can be packaged in an include file that can be called once. It actually generates code ("flex code") that creates a format that can be used in many subsequent data steps:

```
48   * Format Lookup - PROC FORMAT flex code;
49   data TabLook.ZipClean
50        Tablook.ZipDups;
51     set TabLook.ZipSort;
52     by Zip;
```

```
53     if first.zip then output TabLook.ZipClean;
54     else              output Tablook.ZipDups;
55   run;

NOTE: There were 29470 observations read from the data set TABLOOK.ZIPSORT.
NOTE: The data set TABLOOK.ZIPCLEAN has 29467 observations and 8 variables.
NOTE: The data set TABLOOK.ZIPDUPS has 3 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.05 seconds


56   data _NULL_;       * Create translation format;
57     length start  $7;
58     length label $30;
59     set TabLook.ZipClean end=last;
60     file "D:\RSD\Tools\Tutorial\TableLookups\DataV9flxFmt\zipfmt.flx";
61     if _N_ = 1 then put @3 'proc format;' /
62                        @5 'value $zips (min=5)';
63     start =  quote(trim(zip));
64     label = put(City, $char17.) || put(ST, $char2.);
65     label = quote(trim(label));
66     put @7 start ' = ' label;
67     if last then put @7 "OTHER = '                    '" /
68                     @7 ';' / 'quit;';
69   run;

NOTE: The file "D:\RSD\Tools\Tutorial\TableLookups\DataV9flxFmt\zipfmt.flx" is:
      File Name=D:\RSD\Tools\Tutorial\TableLookups\DataV9flxFmt\zipfmt.flx,
      RECFM=V,LRECL=256

NOTE: 29472 records were written to the file
      "D:\RSD\Tools\Tutorial\TableLookups\DataV9flxFmt\zipfmt.flx".
      The minimum record length was 5.
      The maximum record length was 38.
NOTE: There were 29467 observations read from the data set TABLOOK.ZIPCLEAN.
NOTE: DATA statement used (Total process time):
      real time           0.82 seconds
      cpu time            0.08 seconds


70   %include "D:\RSD\Tools\Tutorial\TableLookups\DataV9flxFmt\zipfmt.flx";
NOTE: Format $ZIPS has been output.

NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.62 seconds
      cpu time            0.50 seconds


29543  data TabLook.BothFmtFlx;
29544    set TabLook.Visit;
29545    City = substr(put(ptZip, $zips.), 1, 17);
29546    ST   = substr(put(ptZip, $zips.), 18, 2);
29547  run;

NOTE: There were 63022 observations read from the data set TABLOOK.VISIT.
NOTE: The data set TABLOOK.BOTHFMTFLX has 63022 observations and 129 variables.
```

```
NOTE: DATA statement used (Total process time):
      real time           28.71 seconds
      cpu time            0.59 seconds
```

Here we specified the first of any duplicate zip codes to load into our format. We could have used "if last.zip" to choose the last of any duplicate. In practicality you may wish to have a more discriminating algorithm.

We used a hard-coded path in the Windows environment to store the generated code. We could have declared a temporary file with a "FILENAME SASCODE TEMP;" statement and then used the SASCODE file ref for the temporary code. This has the advantage of portability among platforms, allocating the file in system temporary storage. Also, it automatically discards the generated code. An advantage of explicitly declaring a path for the current platform is that you can examine the generated code for debugging.

Notice the use of the QUOTE function to anticipate any embedded quotes in the city name that could interfere with the quoting used in the flex code. This example was executed on a virtual machine separate from the benchmarks appearing later. The benchmarks give a more accurate assessment of the relative speeds of the techniques.

Format Lookup - PROC FORMAT with a control data set

If you are intimidated by generating code programmatically, SAS provides a convenient way to generate a format for table lookup directly from an existing data set (example SAS v8.2):

```
29501  data TabLook.ctrl;
29502    set TabLook.ZipClean (rename=(zip=start)) end=last;
29503    keep start label fmtname type hlo;
29504    retain fmtname '$zips' type 'c';
29505    label = put(City, $char17.) || put(ST, $char2.);
29506    output;
29507    if last then do;
29508      hlo = 'O';
29509      label = ' ';
29510      output;
29511    end;  * if;
29512  run;

NOTE: There were 29467 observations read from the data set TABLOOK.ZIPCLEAN.
NOTE: The data set TABLOOK.CTRL has 29468 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.05 seconds


29513  proc format cntlin=TabLook.ctrl;
NOTE: Format $ZIPS is already on the library.
NOTE: Format $ZIPS has been output.
29514  run;
```

```
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.18 seconds
      cpu time            0.18 seconds

NOTE: There were 29468 observations read from the data set TABLOOK.CTRL.

29515  data TabLook.BothFmtCtrl;
29516    set TabLook.Visit;
29517    City = substr(put(ptZip, $zips.), 1, 17);
29518    ST   = substr(put(ptZip, $zips.), 18, 2);
29519  run;

NOTE: There were 63022 observations read from the data set TABLOOK.VISIT.
NOTE: The data set TABLOOK.BOTHFMTCTRL has 63022 observations and 129 variables.
NOTE: DATA statement used (Total process time):
      real time           14.77 seconds
      cpu time            0.69 seconds
```

Since the inception of the CNTLIN= option in PROC FORMAT, SAS Institute has recommended this approach over the Flex-code approach. It eliminates the extra step of having to generate "flex" code.

Using the OUTPUT statement when producing the control data set is critical. If omitted, you will lose the last value to your "catch-all" (`hlo = 'O'`). Note that the code for this example does not match the benchmarks later in the paper due to correcting this issue.

SAS uses a binary search for its format lookups – starting at the middle and repeating the process if no match in the half revealed to contain the match based on the comparison results. However, if you know the likely occurrence of the lookup values and believe that ordering them in likelihood order would be even faster than a binary search, you can specify the NOTSORTED option on your PROC FORMAT statement. E.g., in our running example, if you know the zip code 12345 occurs 90% of the time in the visit table, you could force a hit the first time in 90% of the lookups by placing 12345 as the first format range.

Manual Hash

This is a variation on the array solution – with a much faster result. It is not an advanced lookup scheme, like a binary search, but an even faster method – a direct lookup (hash) using the value of the lookup variable (key) as an array or memory offset (also SAS v8.2 here):

```
59305  data TabLook.BothManHash (keep=from--mrreas10 City ST);
59306    array Cities(100000) $17 _temporary_;
59307    array States(100000)  $2 _temporary_;
59308    retain i (0);
59309    do until (lasta);
59310      set TabLook.ZipClean end=lasta;
59311      i = input(zip, 5.0);
59312      Cities(i) = City;
```

```
59313      States(i) = ST;
59314    end;  * until;
59315
59316    do until (lastb);
59317      set TabLook.Visit end=lastb;
59318      if ptZip > '00000' then do;
59319        i = input(ptZip, 5.0);
59320        City = Cities(i);
59321        ST   = States(i);
59322      end; else do;
59323        City = ' ';
59324        ST   = ' ';
59325      end;  * if;
59326      output;
59327    end;  * do until;
59328    stop;
59329  run;

NOTE: There were 29467 observations read from the data set TABLOOK.ZIPCLEAN.
NOTE: There were 63022 observations read from the data set TABLOOK.VISIT.
NOTE: The data set TABLOOK.BOTHMANHASH has 63022 observations and 129 variables.
NOTE: DATA statement used:
      real time           0.98 seconds
      cpu time            0.32 seconds
```

The array here is larger than for the array search technique since we have to account for every possible value of the key field, ptZip. Since we did not account for duplicates when we built our hash array, the last duplicate will be chosen. To choose the first, we would have to test the Cities and/or States arrays to see if a value had already been stored.

If we didn't know the range of values for our lookup keys – in this case zip codes, we could use a PROC MEANS MIN RANGE beforehand to find the minimum and maximum values for they keys. This actually would minimize the number of array elements needed for the array. Instead of declaring a table for all the theoretical values as we did for our example, we could define the size of the array as the range. We then would offset each value by the minimum (INPUT(zip,5.) + min). Alternatively, we could incorporate the minimum and maximum directly as the end points using macro variables built from the PROC MEANS results (e.g., ARRAY Cities (&min:&max)). Then we would not have to use an offset with the key variable when storing or accessing the values in the arrays.

My thanks to Paul Dorfman, who has published numerous papers on the topic – including hashing techniques for character variables and other cases (such as RAM constraints) when the value of the key variable cannot be used directly. More information may be found by searching the SAS-L archives at http://www.listserv.uga.edu/cgi-bin/wa?S1=sas-l for the keyword "hash" in the subject and the keyword "Dorfman" in the author.

Lookup with the SAS 9 Hash Object

But wait! Paul Dorfman no longer gives papers on manual hash objects (at least not that I recall at SUGI 30). There he gave a paper with Koen Vyverman, "Data Step Hash Objects as Programming Tools". The manual hash example above was fortunate since all zip codes (100,000 of them) fit into resident memory (RAM). What if they don't? What if the keys are sparse? Answer to both: much more sophisticated (translation: painful) coding. Want to avoid it and use hashes painlessly? Use the SAS 9 hash object:

```
29619  data TabLook.BothV9Hash (drop=rc zip) ;
29620    length Zip $5 City $17 ST $2;
29621    declare AssociativeArray hh () ;
29622    rc = hh.DefineKey ('zip') ;
29623    rc = hh.DefineData ('Zip', 'City', 'ST') ;
29624    rc = hh.DefineDone () ;
29625    do until (eof1) ;
29626      set TabLook.ZipClean (keep=Zip City ST) end=eof1 ;
29627      rc = hh.add() ;
29628    end ;
29629    do until (eof2) ;
29630      set TabLook.Visit end=eof2 ;
29631      rc = hh.find (key: ptZip);
29632      if rc ^= 0 then do;
29633        City = ' ';
29634        ST   = ' ';
29635      end;  * if;
29636      output ;
29637    end ;
29638    stop ;
29639  run ;

NOTE: There were 29467 observations read from the data set TABLOOK.ZIPCLEAN.
NOTE: There were 63022 observations read from the data set TABLOOK.VISIT.
NOTE: The data set TABLOOK.BOTHV9HASH has 63022 observations and 129 variables.
NOTE: DATA statement used (Total process time):
      real time           2.29 seconds
      cpu time            0.54 seconds
```

Observations

If you looked at the timings in the log snippets for each technique, you might be somewhat misinformed about the performance as the first six techniques were performed with SAS 8.2, the seventh with SAS 9.1.3. Also the FORMAT examples were executed on a virtual machine and not representative of the true performance of those techniques. Here is a table comparing the timings on the same native hardware:

		Run Time (Seconds)	
Technique	**Disk**	**SAS8**	**SAS9**
SQL - outer (left) join	Same	35.64	8.73
Merge left after sorts	Same	8.71	3.37
Array lookup - sequential	Same	338.85	144.03
PROC FORMAT - flex code	Same	1.09	1.89
PROC FORMAT - control data set	Same	1.27	2.43
Manual Hash	Same	0.98	2.84
Hash Object Lookup	Same		2.29

With the exception of the sequential array lookup, the SAS 8.2 techniques get faster from top to bottom. The same is somewhat true of SAS 9.1.3; however, the differences among those that beat three seconds may not be statistically significant. I will note, too, that the first time I ran the hash object method, it completed in just over one second.

Also, it is interesting that the three slowest techniques were significantly improved in SAS 9.1.3 while the faster techniques actually slowed, apparently significantly. "Apparently" since anything under two seconds may be due to a slower setup internally in SAS to gain speed in streaming execution. At the time of this writing, the author cannot verify that. So, the lesson here is to "roll your own", i.e., consider more than one solution if performance is important to you – and benchmark it. Your mileage may vary.

Conclusion

The purpose of this paper is to give you a variety of techniques that can help you address the table lookup problem with dispatch. Keep in mind that the term "table lookup" infers a value (or values) being supplied for each record in a larger table. The larger the "small" table is in relation to the "large" table, the more the problem looks like a match merge, which probably be addressed with one of the first two techniques. Depending on how small your "small" table is, and potentially on other factors, you have an arsenal of approaches. Now, go out and decide on your personal favorite!

I

Sources

1. http://ftp.sas.com/techsup/download/sample/datastep/tables.html, "Table lookup with a small table using array processing", SAS Institute.

2. U.S. Census Bureau, 1990 Zip Code by State file, hosted by PCI Geomatics, at http://spatialnews.geocomm.com/newsletter/2000/jan/zipcodes.html.

3. SAS 9.1 Help and Documentation, Your complete Guide to Syntax, How To, Examples, Procedures, Concepts, What's New, and Tutorials, ©2002-2003, SAS Institute Inc., Cary, NC, http://support.sas.com/onlinedoc/913/docMainpage.jsp.

4. Dorfman, Vyverman, "Data Step Hash Objects as Programming Tools", Paper 236-30, SUGI 30 Proceedings, http://www2.sas.com/proceedings/sugi30/236-30.pdf.

5. Rick Aster and Rhena Seidman, *Professional SAS programming Secrets*, Windcrest®/McGraw-Hill, 1991, ("flex code").

All the examples were taken from one of the above sources or from RSD production code.

For More Information

Rob Rohrbough
Rohrbough Systems Design, Inc.
Omaha, Nebraska
Rob@RohrboughSystems.com
http://www.RohrboughSystems.com
(402) 343-1493

The examples provided in this paper are available on RSD's website, above. If you have trouble locating or downloading, please contact the author.

The Fine Print