Paper 3466-2015

E-Z Simple Hash Object Lookups

Andrew Dagis, City of Hope National Medical Center, Duarte CA

ABSTRACT

We have to pull data from several data files in creating our working databases. The simplest use of SAS® hash objects greatly reduces the time required to draw data from many sources when compared to the use of multiple proc sorts and merges.

INTRODUCTION

Retrospective medical studies are popular because, though less rigorous than well-planned prospective studies, much of the data usually have already been collected in large databases. Further, research hospitals such as City of Hope often have case-series of patients that can be found nowhere else.

The creation of working datasets may involve combining data from several large databases of varying ages and formats. Data merging programs often have to be re-run as the last project-specific data and the last missing data are collected. The time required to run and re-run these unwieldy programs becomes a nuisance that statistician-programmers would be happy to minimize.

The use of SAS hash component objects in table lookups is effective in reducing the time to draw data from large datasets. SAS 9 hashing consists of a small number of object-oriented methods, only a few of which are used in running table lookups. Hash object syntax in SAS 9 is extremely simple, utilizing 'dot notation'. Dot notation is a form-follows-function creation that also serves to indicate the use of hashing in the data step. Proc sort and data step merge are rendered obsolete for many large data combining programs.

SAS 9 HASH COMPONENT OBJECT

Hash objects in the data step of SAS 9 greatly increase the speed of data look-up, compared to sorts and merges, and usually to SQL joins. Hash objects reside entirely within RAM, and bypass the read-process-write cycle on which SAS was originally designed. Internally, bucketing is used in ways that avoid slow search algorithms. Further, direct addressing allows table lookups to run at speeds approaching O(1), that is, increasing the size of the hash object does not increase search time.

In-memory data manipulation such as hashing is not new—look at temporary arrays. The _temporary_ array is a special type of array that lies outside of the program data vector (PDV). Because of this, temporary arrays exist throughout the data step, but vanish at the end of the data step without writing their contents to the SAS dataset named at the top of the data step. These arrays bypass the data step cycle of reading one record from disk, operating on it, writing the result to disk, then repeating. The first time I saw a colleague's SAS code that had temporary arrays, I thought he was crazy for not basing his work on SAS's built-in looping feature—not very SAS like. I thought he was a SAS newbie, a refugee from C programming, having trouble wrapping his head around the way the SAS data step loops without an explicit 'do' statement. But instead, he was making his code run much faster than mine.

Hashing is another kind of in-memory technique. Hash tables, like temporary arrays, are entities that reside outside of the data step's built-in loop. They also disappear with the termination of the data step. But it is easier (in SAS 9) to code table look-ups using hash objects than to code temporary arrays. . Hash objects are keyed on variable names, not subscripts, like temporary arrays. Hash objects allow multiple keys, can mix character and numeric variables within the same hash object, and don't require prior specification of array size, outside of just listing variable names. Hash objects are flexible: they can be created, loaded, manipulated, sorted, emptied, reloaded, destroyed, and re-created, all within a data step. In more advanced usages, it is possible to create SAS datasets with multiple records from hash objects that iterate only once.

Hashing is not a new concept in computer programming, nor is it new even for SAS. SAS has been using the technique internally within Proc SQL since SAS V6.08 [1]. If you had a background in computer science, or were following Paul Dorfman's early SUGI papers on the subject [2, 3], you could code up hashing yourself--and reap the benefits of speedy lookup tables. With SAS 9, SAS has "canned" the hash object for the benefit of the rest of us. Since my primary training is in statistics from a public health perspective, my interest in hashing is limited to the hash and hash iterator SAS component objects. This facility within SAS is now stable and considered "production" as of SAS 9.1. The ability to deal with multiple records per key was added in SAS 9.2.

In the case of look-ups from large databases, when pulling in small numbers of variables per database, the speed of building a working data set using hash objects is very fast. Hashing is so fast that it puts sort-and-merge out of business. I would report my experience of this increase in SAS performance, but I want to avoid having to obtain permission from the Institutional Review Board just for that. Instead, I will share my favorite introductory reference on the topic: "I Cut My Processing Time by 90% Using Hash Tables—You Can Do It Too!" [4] is an aptly named paper that explains everything associated with simple table lookups using hashing.

The hash object, coded as a form of object –oriented programming, requires a different syntax called 'dot notation'. Fortunately, there are only a few methods to learn, especially for simple table lookup.

Hashing typically beats Proc SQL in speed. Proc SQL has an internal feature called the SQL Optimizer that directs the way Proc SQL performs table lookups (joins). Sometimes internal sorting is done, which is not a particularly fast way of manipulating data. Sometimes memory-resident techniques are performed: if one of the tables is less than 1% of the size of the other, Proc SQL may choose hashing to combine the tables. The SQL Optimizer is continually a work-in-progress [1]. Speeds in Proc SQL have been upgraded in SAS 9.3 [5], so it may be better to use Proc SQL in some cases. Benchmarking will guide you to the right choice.

USING HASH OBJECTS FOR TABLE LOOKUPS

Although a hash object exists in the data step, it resides entirely in RAM memory. Data step variables are different: typically, the records in which they reside are read one-iteration-at-a-time from the disk, operated on, and saved to an on-disk output data set. Disk reads and writes are many times slower than reads and writes to RAM. The hash object exists in the data step apart from the iterative process. This means that the hash object's variables might not be automatically refreshed to missing during data looping, because the top of the data step may not be reached. Hence the need to explicitly clear the buffer before loading each row of the hash object: 'call missing()' is good for this.

Hash object variables can be made to communicate with data step variables only if they have the same names as the hash variables. The hash object's variables must be defined before they can be loaded. This can be done with either an 'if _n_=0 then set <dataset>;' statement, or explicitly with an attrib statement. The data step's variables are defined during compilation usually with a set statement that is later executed, hence no 'if _n_=0' in front of it.

The communication between data step variables and hash variables with the same names is effected by calling hash methods and methods with parameters. SAS decided to code up its hash component object using object-oriented features. One standard feature in object-oriented programming is the 'method', which is essentially refers to the different ways a function performs depending on the context in which it is found, and on the object type of the parameters it calls. For didactic purposes, let us consider the function in the C programming language. C Functions can only return one expression, often an integer, as a return code. This returned integer is almost never the working output of the function; that is mediated through pointers to C arrays. Instead, the returned integer is typically coded by the programmer to return information about the performance of the function. In SAS hash methods, the returned expression, or return code, is a floating point number used only to indicate whether the method has worked successfully. Zero means success. Failure of a method to work successfully does not necessarily mean that the method is in an error condition; it can mean something as simple as 'a match has not been found'. The return code is useful not just in debugging code, but also for coding branch points, such as in an if-then statement.

Some methods work without being passed parameters, based on information from methods that are associated with the same hash object through dot notation. Others require parameters, as well, such as the names of key variables or data variables. The lack of concrete input and output based on variables and equals signs make the dot notation look nebulous. The way to see what the method is doing is to look at the rest of the hash object.

The structure of a method is '<return code=> hash object name.method(<'parameter1', 'parameter2', …>);'. Saving the return code is optional, depending on whether you are going to use it in making decisions or in the aforementioned debugging. Note that the hash object name comes before the dot in the dot notation. The hash object is named for purposes of maintaining organization: there may be more than one hash object in a data step. The hash object name tells the method following the dot which hash object variables to operate on.

OPERATING CHARACTERISTICS

The SAS Hash object holds one of the datasets in memory for direct addressing. The other dataset, the the base dataset, has the usual read-operate-write behavior.

Voila: you have a combine (merge/join) with:

- One less sort ahead of time, (or one less de-facto sort within Proc SQL)
- The read-in, operate, write cycle applying to just one of the two datasets,
- Memory-resident bucketing and direct-addressing techniques, resulting in
- Speeds that are orders of magnitude faster.

The table look-ups will now take an inconsequential amount of computing time and real time.

RECIPE

Hash objects are so simple to program and so effective in saving running time over sort/merge, that you should start now. Fancier stuff can come later.

The simplest hashing recipe I use is:

- Sort a base file once.
- Run a series of hash object combines (look-ups).
- Merge the series of results once, without further sorting.
- Done.

Here is an example using this recipe:

```
data BitBucket1;
   if _n_=0 then set followUp;
   if _n_=1 then do;
      DECLARE hash h(dataset: 'followUp');
      h.defineKey('IDnumber');
      h.defineData('vitalStatusFU', 'deathDateFU', 'CauseDthFu',
         'lastNameFU', 'firstNameFU');
      h.defineDone();
   end;
   call missing(IDnumber, vitalStatusFU, deathDateFu, causeDthFu,
      lastNameFu, firstNameFu);
   set coreData;
   if h.find()=0 then output;
run;
```

It is a good idea for the temporary SAS dataset followUp to have been created using a keep statement, so that unused and empty variables are not brought along, slowing down the hashing process and

needlessly inflating the size of the final working dataset.

BETTER RECIPE

You can have multiple hash-objects in one data step. This reduces all of the lookups to one step, and eliminates the final merge. Each hash-object has its own key, which may or may not be the same as the other hash-objects' keys. Each key must have a variable of the same name in the base dataset.

Notice that I have used the attrib statement, rather than the if _n_=0 then set <dataset>; statement.

```
data lib.vitalStatusData;
   attrib vitalStatusFu       length=$100
          deathDateFu         length=8         format=mmddyy10.
          causeDthFu          length=$200
          lastNameFu          length=$15
          firstNameFu         length=$15
          vitalStatusBmtFu    length=$8
          causeDthBmtFu       length=$200
          lastNameBmtFu       length=$15
          firstNameBmtFu      length=$15;
   if _n_=1 then do;
      DECLARE hash h1(dataset:'patdemfudead');
      h1.defineKey('IDnumber');
      h1.defineData('vitalStatusFu', 'deathDateFu', 'causeDthFu',
         'lastNameFu', 'firstNameFu');
      h1.defineDone();
      DECLARE hash h2(dataset:'followup');
      h2.defineKey('IDnumber');
      h2.defineData('vitalStatusBmtFu', 'deathdataeBMTFu', 'causeDthBmtFu',
         'lastNameBmtFu', 'firstNameBmtFu');
      h2.defineDone();
      DECLARE hash h3(dataset: 'regulatoryOrg');
      h3.defineKey('researchID');
      h3.defineData('deathDateRo', 'VitalStatusRo', 'causeDthRo');
      h3.defineDone();
      call missing(
         vitalStatusFu, deathDateFu, causeDthFu, lastNameFu, firstNameFu,
         vitalStatusBmtFu, deathDateBmtFu, causeDthBmtFu, lastNameBmtFu,
         firstNameBmtFu, deathDateRO, vitalStatusRo, causeDthRo);
   end;
   set coreData;
   rc1=h1.find();
   rc2=h2.find();
   rc3=h3.find();
   if ((rc1=0) or (rc2=0) or (rc3=0)) then vitalStatus=0;
   else vitalStatus=1;
```

Note that there were two lookup's using the key variable 'IDnumber', and one using'researchID'.

MULTIPLE RECORDS PER KEY

The above two recipes require that only one record be in any identifier. I like to keep my program at the level of one record per identifier, even if I have to use multiple variables in 'by' statements in the sorts, or multiple keys using hash objects.

Hash objects since SAS 9.2 have had the facility to combine multiple records per key, giving you the ability to do many-to-many merges as in Proc SQL. Since I never use many-to-many merges, I am not

going to follow up on this capability.

HOW HASHING FITS INTO MY WORK WORLD

Most of my work is in support of large retrospective studies in medical research. Retrospective studies are not definitive as are prospective studies--instead they are hypothesis-generating. They are attractive because much unique and very expensive data have already been collected and are available for use. They also provide one way of attacking problems for which prospective, randomized studies are infeasible or unethical. Prospective studies are expensive, have long data collection times, and for all the effort, they may fail, or the publication may be 'scooped' by other research centers, or the question may be rendered moot by other discoveries. Retrospective studies, on the other hand, allow more immediate reply to findings in the literature.

Because of the expense of collecting data, every effort is made not to re-collect data that might be found somewhere in the in-house databases. Retrospective studies typically need data to be drawn from several large, routinely-maintained databases. There are always study-specific data that data-collection personnel have to go to the medical record and collect. There are also cases of missing or inadequately-collected data that have to be completed.

Table look-ups are required in assembling working datasets from large databases. This is more of a cause for concern in retrospective studies than in the case of prospective studies, where data are collected onto a few small but carefully-designed study-specific tables. Sort-and-merge is adequate for constructing datasets in prospective trials, but not for table lookups in large databases.

Data collection is governed by the 80-20 rule: "Eighty percent of the effort is used to perform twenty percent of the work." Studies require 100 percent of the available data to be captured. This guarantees many re-runs of the table look-up programs that generate the study database, while the last few points trickle in.

Hash objects save time. Re-running after data modification becomes much less unwieldy. I don't have to lose track of what I am working on while waiting for an update to the data to crunch its way through the look-up tables.

CONCLUSION

Hashing has an important place in retrospective medical research. The SAS hash component object makes table look-ups very fast, with a very small investment in programming skill. Try it now--you can learn advanced applications of hashing, later.

REFERENCES

[1] Lavery, Russ. 2005. "The SQL Optimizer Project: _Method and _Tree in SAS®9.1." *Proceedings of the Thirtieth SAS Users Group Internatonal Conference*, Cary, NC: SAS Institute. Available at http://www2.sas.com/proceedings/sugi30/101-30.pdf.

[2] Dorfman, Paul. 2001. "Table Lookup by Direct Addressing." *Proceedings of the SAS Users Group International 26*, Long Beach, CA : SAS. Available at http://www2.sas.com/proceedings/sugi26/p008-26.pdf.

[3] Dorfman, Paul M. and Snell, Greg. 2003. "Hashing: Generations." *Proceedings of the SAS Users Group International 28,* Seattle, WA : SAS. Available at http://www2.sas.com/proceedings/sugi28/004-28.pdf.

[4] Warner-Freeman, Jennifer K. 2007. "I Cut My Processing Time by 90% Using Hash Tables—You Can Do It Too!" *Proceedings of the Twentieth North-East SAS Users Group Meeting*, Cary, NC: SAS Institute Inc. Available at http://www.nesug.info/Proceedings/nesug07/bb/bb16.pdf

[5] Johnson, James. 2012. "Merge vs. Join vs. Hash Objects: A comparison using 'Big' Medical Device Data." *Proceedings of PharmaSUG 2012,* Cary, NC: SAS Institute Inc. Available at http://www.pharmasug.org/proceedings/2012/TF/PharmaSUG-2012-TF08.pdf.

ACKNOWLEDGMENTS

All the statisticians at City of Hope, particularly: Rebecca Ottesen, Nora Ruel, Chris Ruel, Nicole Tsai, Joycelynne Palmer, and Jeff Longmate.

RECOMMENDED READING

- *SAS® Hash Object Programming* Made Easy
- *The Brothers Karamazov,* with all the time you'll save

CONTACT INFORMATION

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

Andrew Dagis
City of Hope National Medical Center
1500 E. Duarte Rd.
Duarte, CA 91010-3000
626-357-2658 x62354
adagis@coh.org