PhUSE 2015

Paper TS03

Getting the Most out of Keys - Using Hash Objects to Select, Subset, or Summarize

Pamela J. Giese, inVentiv Health, Durham, United Kingdom

ABSTRACT

It can be a challenge to process big datasets. For permanent datasets, indexing can provide efficient data access. But what about those huge temporary datasets created during processing? Hash components can provide a solution. In one real-life situation, the use of hash components decreased overall program performance by one hour. Hash components can also be used to select, subset, or summarize data by using key files. This paper will explore examples of using hash components on large datasets and compare metrics with more common SAS processing methods.

Introduction

Hash objects are components within SAS that provide an alternative method for processing data. Think of it as loading a dataset into a key array and then processing this "array" within another dataset. Advocates who favor use of hash objects promise increase processing speed. This paper will present an overview of hash objects and examples of using hash objects to select, subset, and summarize data.

An overview of Hash Objects

Before delving into hash objects it's good to take some time to think about how "keys" are used in SAS. Key variables are an important concept in SAS. We use key variables within sorts and within procedures to encapsulate and define hierarchy, order, and uniqueness within SAS.

```
Data newfile;
Merge myfile otherfile;
BY trialno subject visit;

Proc Freq data=newfile;
BY trialno subject visit;
Table bodyweight ;
```

Keys

Keys

To use the key variables in most procedures, the data needs to sorted by these same keys first. This usually happens with a PROC SORT or PROC SQL. It is this step which can be the most resource intensive, as SAS needs to grab extra space for the sort. For very large datasets, this can be a useful to index the dataset. This stores commonly used keys external to the regular dataset. The down side of indexed datasets is that the indexing can greatly increase the permanent storage needed, making it only practical for permanent, re-usable datasets.[1]

If you're familiar with indexed SAS datasets, it can be useful to think of hash components as a way of customizing your indices within a temporary dataset. I tend to think of hash objects as arrays of keys that work on an iterative pathway through the data. Another way to visualize how hash objects work is with the notion of "buckets". Imagine your data is a mass of "stuff" and you want to organize it. Depending on the keys of each record, the record is put into a "bucket". This is illustrated in Figure 1. The number of "buckets" is defined by the HASHEXP parameter. This is discussed in detail after Example 1.

[1] If you have access to Oracle or a similar external database/language it is possible to sort the datasets in Oracle and then continue in SAS. When configured correctly, SAS will recognize the set as sorted.

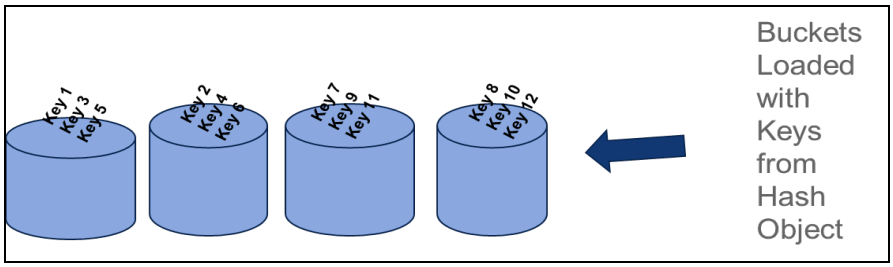

Figure 1 Example of buckets. HASHEXP:4

There are two ways to specify and work with hash objects. The first is to either specify keys within the Data step or to have a separate file with only the key values. This is done in Examples 1 and 3. The result is program code which is not conceptually different than using multiple "set" statements in a program.

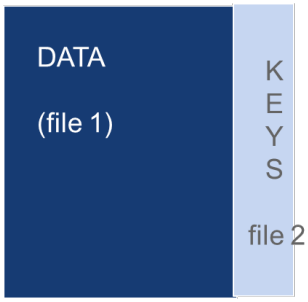

Figure 2 Hash objects as a file of data and a file of keys.

Another way that hash objects work is through the use of a hash iterator (HITER). This is demonstrated in Example 2. The hash iterator works as a pointer and moves through the data. This would allow for more complex programming where there is a need for different actions at different keys or after other constraints. It can be visualized as:

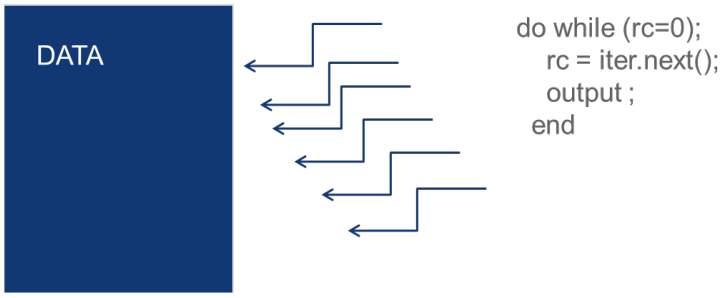

Figure 3 A hash iterator (hiter) works as a pointer to move through the dataset.

These are some of the basic concepts to prepare to work with hash objects. With most programming concepts, examples really show how code is created. Below are three examples using different tasks and showing some of the varied power of hash objects.

Examples

Example 1: Selecting from a look-up table

Problem: Our current pool has a VISIT dataset with 901128 records with keys of TRIALNO and VISIT. There is a "lookup" dataset which contains visit descriptions. This is VISDESC, again with keys TRIALNO and VISIT and consists of 1049 records. The goal is to merge these two files together to pick up the visit label VISITSP.

Let's look at the code and execution time doing this task with 1) hash objects, 2) SQL and 3) SORT then MERGE. To minimize differences, following were run in a single program:

Method 1: HASH code

```
data allvis;
   length visitsp $60;
   if _n_ = 1 then do;
```

```
        declare hash visdes(hashexp:8, Dataset: 'visdesc');
        rc=visdes.DefineData('visitsp');
        rc=visdes.definekey('trialno','visit');
        rc= visdes.definedone();
        call missing(visit, visitsp);
      END;
      set visit;
      rc = visdes.find(key:trialno, key:visit);
      if rc = 0 then do;
        output allvis;

      END;
    RUN;

NOTE: There were 1049 observations read from the data set VISDESC.
NOTE: There were 901128 observations read from the data set VISIT.
NOTE: The data set WORK.ALLVIS has 896727 observations and 22 variables.
NOTE: DATA statement used (Total process time):
    real time          7.04 seconds
    cpu time           4.09 seconds
```

Method 2: PROC SQL

```
    proc sql;
      create table sqlallvis as
      select a.*, b.visitsp from visit as a,  visdesc as b
          where a.trialno = b.trialno and a.visit = b.visit;
NOTE: Table WORK.SQLALLVIS created, with 896727 rows and 21 columns.

    QUIT;
NOTE: PROCEDURE SQL used (Total process time):
    real time          10.16 seconds
    cpu time           4.86 seconds
```

Method 3: PROC SORTS and MERGES

```
    proc sort data=visdesc out=visitdesc (keep=trialno  visit visitsp);
      by trialno visit;
    RUN;

NOTE: There were 1049 observations read from the data set VISDESC.
NOTE: The data set WORK.VISITDESC has 1049 observations and 3 variables.
NOTE: PROCEDURE SORT used (Total process time):
    real time          0.01 seconds
    cpu time           0.01 seconds


    proc sort data=visit out=visit;
      by trialno visit;
    RUN;

NOTE: There were 901128 observations read from the data set VISIT.
NOTE: The data set WORK.VISIT has 901128 observations and 20 variables.
NOTE: PROCEDURE SORT used (Total process time):
    real time          7.49 seconds
    cpu time           6.01 seconds


    data oldallvis;
      merge visitdesc(in=in1 ) visit(in=in2);
      by trialno visit;
      if in1 & in2;
    run;

NOTE: There were 1049 observations read from the data set WORK.VISITDESC.
NOTE: There were 901128 observations read from the data set WORK.VISIT.
NOTE: The data set WORK.OLDALLVIS has 896727 observations and 21 variables.
NOTE: DATA statement used (Total process time):
    real time          9.63 seconds
    cpu time           3.33 seconds
```

Proc Compares showed that all methods created the same tables (aside from variable order). So, in summary, the results look like:

Comparison of processing time in Lookup table example		
Method	Real time (seconds)	CPU time (seconds)
Hash Objects	7.04	4.03
SQL	10.16	4.86
Total Merge (including sorts)	17.13	9.35

So using hash objects for this merge was the fastest of the three methods.

Before exploring hash components in more detail, let's look at the components of our hash example. Here is our code from above:

```
data allvis;
   length visitsp $60;
   if _n_ = 1 then do;
      declare hash visdes(hashexp:8, Dataset: 'visdesc');
      rc=visdes.DefineData('visitsp');
      rc=visdes.definekey('trialno','visit');
      rc= visdes.definedone();
      call missing(visit, visitsp);
   END;
   set visit;
   rc = visdes.find(key:trialno, key:visit);
   if rc = 0 then do;
      output allvis;

   END;
RUN;
```

Hash components work with keys variables. Keys may be in separate files, as in VISDESC (Visit description look-up file) or can be specified within the Data Step. In this example the keys are in the lookup file VISDESC. They are only defined once at the first observations. The statement

```
declare hash visdes(hashexp:8, Dataset: 'visdesc');
```

declares that VISDES is a hash component, where data exists in dataset VISDESC. The HASHEXP will be explained in more detail below. Think of it as the number of "buckets" where SAS keeps the data as it is processed. The next three statements use "rc=..". rc is the return code (0,1) which indicates whether or not the step was successful. It is optional and the statements can be written without the "rc=".

`rc=visdes.DefineData('visitsp');`	Defines the list of variables to be added to the data (in this case only the variable 'visitsp' is to be kept
`rc=visdes.definekey('trialno','visit');`	Defines the key variables
`rc= visdes.definedone();`	Ends the hash definition of VISDES
`call missing(visit, visitsp);`	Allows missing values to be kept. Without this unmatched keys can cause error

Next, is the statement for SET VISIT. The statement

```
rc = visdes.find(key:trialno, key:visit);
```

Tells SAS to find the matching key in VISDES for the observation in the dataset VISIT. It there is no error (rc=0), output the record.

HASHEXP

In the previous example, the hash table itself was declared with the statement:

```
declare hash visdes(hashexp:8, Dataset: 'visdesc');
```

The value of "8" represents the HASHEXP parameter. This is a size of the internal hash table of represents 2^n where n=HASHEXP. This doesn't equate to number of records. The SAS documentation suggests to think of HASHEXP as the number of buckets for holding contents, see Figure 1. It can vary up to 20. The manual suggests experimenting with different values. So, using the previous example, here are some different performances for different HASHEXP values:

Comparison of different HASHEXP values on lookup table example		
HASHEXP	Real time (seconds)	CPU time (seconds)
8	7.04	4.03
10	7.83	3.99
12	9.48	3.41
16	11.23	4.03

20	10.24	4.14

From the above, there seems to be a point where increasing HASHEXP worsens performance. Of course all of this would vary based on individual datasets. The SAS documentation suggests that if the hash object contains 1,000,000 records, a HASHEXP of 9 or 10 would probably be optimum. This doesn't seem too far off from this example where the hash object is 901,128 records.

Which file to be used as HASH Object?

The SAS documentation seems to imply that it is better to use the larger dataset as the HASH object. Because there are multiple records in VISIT for each unique record in VISDESC, the program code needs to be modified to "walk" through the VISIT dataset for each VISDESC record. Here is the comparison of the code and timings. As performance seems to be very sensitive to server load, I reran the Look-up example able.

```
 data allvis;
   length visitsp $60;
   if _n_ = 1 then do;
      declare hash visdes(hashexp:12, Dataset: 'visdesc');
      rc=visdes.DefineData('visitsp');
      rc=visdes.definekey('trialno','visit');
      rc= visdes.definedone();
      call missing(visit, visitsp);
   END;
   set visit(keep= trialno visit patno country recno visit_d visit_m visit_y  visit_tm);
   rc = visdes.find(key:trialno, key:visit);
   if rc = 0 then do;
      output allvis;

   END;
 RUN;

NOTE: There were 1049 observations read from the data set VISDESC.
NOTE: There were 901128 observations read from the data set VISIT.
NOTE: The data set WORK.ALLVIS has 896727 observations and 11 variables.
NOTE: DATA statement used (Total process time):
    real time          5.92 seconds
    cpu time           3.50 seconds

   data allvisb;
      length country $2 visit_tm $5 patno recno visit_d visit_m visit_y ;
      if _n_ = 1 then do;
         declare hash visdes(hashexp:12, Dataset: 'visit', multidata:'Y');
         rc=visdes.DefineData('country','patno','recno','visit_d','visit_m','visit_y' ,' 'visit_tm');
         rc=visdes.definekey('trialno','visit');
         rc= visdes.definedone();
         call missing(visit, patno, country, recno, visit_d, visit_m, visit_y, visit_tm);
      END;
      set visdesc (keep= trialno visit visitsp);
      rc = visdes.find(key:trialno, key:visit);
         do while(rc = 0);

            rc = visdes.find_next();
            output allvisb;
          end;
       if rc = 0 then do;
         output allvisb;

      END;
   RUN;

NOTE: There were 901128 observations read from the data set VISIT.
NOTE: There were 1049 observations read from the data set VISDESC.
NOTE: The data set WORK.ALLVISB has 896727 observations and 11 variables.
NOTE: DATA statement used (Total process time):
    real time          5.53 seconds
    cpu time           4.68 seconds
```

Using the smaller file as the key was indeed faster, but not by much. There may be times when switching the hash

object makes sense.

Example 2: Subsetting

In the same way as you might subset using a PROC SORT with a "where=" clause to subset. You can also subset via the keys and hash object. I include this example to show it can be done. For such a simple case, no one would ever invest the coding time to use hash objects on such a simple case. Here is a simple example and comparison:

Problem: Select all records where VALUE_C >= 11.5 Original dataset WBC had 258878 observations. Resulting dataset had 22080 observations.

Method 1: Dataset

```
data datasetselect;
   set wbc;
   if value_c >=11.5;
RUN;
```

Method 2: SQL

```
proc sql;
   create table sqlselect as
        select * from wbc where value_c >= 11.5;
quit;
```

Method 3: Hash

```
data hashhi;
length country $2  trialno patno  value_c 8 ;
declare hash myhash(hashexp: 4, dataset:"work.wbc",  multidata:"yes", ordered:'yes');
myhash.DefineData ('trialno', 'country','patno', 'value_c' );
declare hiter iter('myhash');
myhash.defineKey ('value_c');
myhash.defineDone();
do while (not done);
set wbc end=done;
rc = myhash.find();
end;
done = 0
rc = iter.setcur(key: 11.5);
  do while (rc=0);
    rc = iter.next();
    output ;
  end;
run;
```

Comparison of Times to subset data		
Method	Real time (seconds)	CPU time (seconds)
Hash Objects	0.93	0.78
SQL	0.08	0.06
Data Step	0.11	0.11

Example 3: Summarizing

Analyzing concomitant medication (a medication in addition to the study drug) is one of those tasks that deceptively complex. Because the problem can often be easily verbalized – i.e. What patients were on NSAIDs in addition to the study drug at the time of a heart attack? – it seems straight-forward. However, the more an analyst dives into the specifics, the more complex the problem becomes: How are multiple drugs within the same drug class to be considered? What about interruptions? Dose changes? Are there windowing rules for the drugs? All drugs or some drugs? If we add to this the need for robust program code to handle multiple situations or subgroups, then these programs get very complex. In the course of analysis, most solutions end up creating a daily snapshot of each subject's medications of interest. In a large multi-year study with tens of thousands of patients, these temporary files can become huge. This was the problem that motivated me to investigate hash objects. Below is a snippet of from our analysis program.

Problem: Calculate the number of days that a patient is on a concomitant medication (a medication in addition to the study drug), saving the last date.

Initial Solution:

Data set contains 124,663,088 observations

```
proc sort data=patyr_drug;
    by calcno cmno subno sublabel subcode subvalue refint treatmgr
      cmny _valid trial country trunit patno obs_dt;
run;
data patyr_drug;
    set patyr_drug;
    by calcno cmno subno sublabel subcode subvalue refint treatmgr
      cmny _valid trial country trunit patno;
  retain onset_days;
  if first.patno then onset_days=1;
  else              onset_days+1;
  if last.patno;
run;
```

Real time for this chunk of code : > 1 hour 47 minutes; CPU time : > 30 minutes. The bulk of the time is in the Proc Sort.

Using Hash

Note: Still requires a PROC SORT , but NODUPKEY reduces time
(time for Proc Sort: 34 minutes Real, 15 minutes CPU)

```
****Create Key file;
proc sort data=patyr_drug out=patkeys nodupkey;
   by  calcno cmno subno sublabel subcode subvalue refint treatmgr
      cmny _valid trial county  trunit patno ;
RUN;

data onset;
declare hash myhash (hashexp:20, suminc:"count", Dataset: 'patyr_drug', duplicate: 'r');
      rc=myhash.definekey('calcno', 'cmno', 'subno' ,'sublabel', 'subcode', 'subvalue' ,'refint', 'treatmgr' ,'cmny','_valid' ,'trial', 'county'
,'trunit', 'patno' );
      rc= myhash.definedata('obs_dt');
      rc= myhash.definedone();
   count = 1;
   do while (not done);
   set patyr_drug end=done;
   rc = myhash.find();
   end;
   done=0;
do while (not done);
      set patkeys end=done;
```

Count over the keys and save as onset_days

```
      rc = myhash.sum(sum:onset_days);
      output;
   END;
   stop;
RUN;
```

Comparison of processing Times in summarization example		
Method	Real time (seconds)	CPU Time (seconds)
Hash Object: Key Sort	34:41	14:46
Hash Object: Dataset	1:16:56	25:45
Total Hash	1:51:37	40:31
Original: Full Data Sort	53:29	18:56
Original: Dataset	1:47:44	30:27
Total Original	2:41:13	49:31

CONCLUSION

This concludes a brief introduction to using Hash Objects. When dealing with big, temporary datasets, using hash objects can improve performance, mostly be reducing the need for a PROC SORT or PROC SQL. However, it is not intuitive and requires some practice and experimentation… and a fair amount of comments and documentation.

Contact Information

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

Pamela J. Giese
inVentiv Health
27 Logan St.
Langley Park
Durham / DH7 9YN
Work Phone: +44 (0) 191 373 0550
Email: pamela.giese@inventivhealth.com
Web: http://www.inventivhealthclinical.com/

About inVentiv Health

inVentiv Health is a top-tier professional services organization that accelerates the clinical and commercial success of biopharmaceutical companies worldwide. Our combined Clinical Research Organization (CRO) and Contract Commercial Organization (CCO) help clients improve their performance to deliver much-needed therapies to market. With 14,000 employees providing services to clients in 70 countries, inVentiv Health designs best practices, processes and systems to enable clients to successfully navigate an increasingly complex environment. For more information, visit inVentivHealth.com.