Paper AD-102

More Hash: Some Unusual Uses of the SAS® Hash Object

Haikuo Bian, Regions Bank; Carlos Jimenez, Regions Bank;
David Maddox, Regions Bank

ABSTRACT

Since the introduction of the SAS® Hash Object in SAS® 9.0 and recent enhancements, the popularity of the methodology has been grown. The significant effects of the technique in conjunction with the large memory capacity of modern computing devices has brought new and exciting capabilities to the data step. The most often cited application of the SAS Hash Object is table lookup. This paper will highlight several unusual applications of the methodology including random sampling, "sledge-hammer matching", anagram searching, dynamic data splitting, matrix computation, and unconventional transposing.

INTRODUCTION

The SAS hash object is a high performance look-up table that resides in memory and provides an efficient mechanism for data storage and retrieval based on lookup keys. It is implemented as a DATA step component and is not part of the DATA step itself. It is, in effect, a black box device that can be used inside the DATA step. However, it does not understand the DATA step language and must be manipulated using the object-dot syntax. It is important to understand that all hash object operations are performed at the DATA step run time. One of the most popular and effective uses of the hash object is the table lookup. There are a number of excellent papers by Paul Dorfman, Peter Eberhardt, and others that thoroughly discuss table lookup via the hash object. Our purpose in this paper is to highlight a few additional uses of the hash object and how it can be used to perform certain programming tasks. The applications that we will examine are "sledge-hammer matching", anagram searching, dynamic data splitting, matrix computation, unconventional transposing, and random sampling.

"SLEDGE-HAMMER MATCHING"

The term "Sledge-hammer Matching" was coined in a SAS-L discussion. The term is informal and calling it a "Domino match" may be more appropriate.

Here is an example - if we have two columns, one represents "From", and the other is "To". They can be two-ways, such as transportation hubs, or one-way, such as family trees. The purpose is to exhaust all the possible "To"s for a specific "From". The following example represents a two-way scenario:

From	To
A	B
B	C
B	D
C	E
C	B

And the expected outcome will be like this:

A	B C D E
B	C D E
C	B D E

SOLUTION

There are three key components in the following solution.

1. For the sake of efficiency and to lessen the prospect of being trapped in an indefinite loop, a collection of distinct 'From's has been prebuilt.
2. Load every possible 'To' for a specific 'From' to a hash table distinctively for all the possible levels. For every newly added 'To', a new search loop will be initiated. Even though some keys could be searched repeatedly, performance is not an issue since the process is being performed in memory.

3. Concatenate the final outcome by looping through the hash table containing all of the distinct values of 'To'.

The code may be implemented as follows:

```
/*Prepare the sample data*/
data have;
       input from $ to $ @@;
       cards;
A C A B B C B A C A A F A G C B D A E D F G D C F A A H
;

/*Prepare the distinct "From"s*/
proc sql;
       create table have1 as
              select distinct from from have;
quit;

data want;
       if _n_=1 then
              do;
                     if 0 then
                            set have (rename=(from=_from to=_to));/*Prepare the
                                                          environment for Hash 'h' variables*/
                     declare hash h(dataset:'have (rename=(from=_from to=_to))',
                     multidata:'y'); /*Load the source table into Hash, Rename to
                                    prevent overriding*/
                     h.definekey('_from'); /*Define Hash key element as _from, aka,
                                   _from is the key for Hash 'h'*/
                     h.definedata(all:'y');/*Define all of the variables in the source
                                  table as data elements*/
                     h.definedone(); /*Hash 'h' definition finished*/
              end;

       declare hash h1(ordered:'a'); /*Declare another Hash table for manipulating
        data on the fly, 'h1' will be initiated freshly for every implicit data step
           loop*/
       h1.definekey('new'); /*'new' is the key element for Hash 'h1'*/
       h1.definedata('new');/*'new' is also the data element for Hash 'h1'*/
       h1.definedone();/*Hash 'h1' definition finished*/
       declare hiter hit('h1'); /*Declare hiter object on Hash 'h1'*/
       retain new  ' '; /*Prepare the environment for Hash 'h1' variables*/
       set have1;
       length newvar $50;

       /*  Do-loop below is to exhaust 'To's for current 'From' on the first level,
             and load them into Hash 'h1'*/
       do rc=h.find(key:from) by 0 while (rc=0);
              if _to ne from then /*Only load non-from onto Hash 'h1'*/
                     h1.replace(key:_to, data:_to);
              rc=h.find_next(key:from);
       end;

       /* Do-loop above is to exhaust 'To's for current 'From' on the first level, and
          load them onto Hash 'h1'*/
       /*Outer do-loop is to exhaust 'To's for current 'From' on all levels*/
       do rc=hit.first() by 0 while (rc=0);
              rc=h.find(key:new);

              /*Inner do-loop is to keep adding new 'To's to Hash 'h1' */
              do rc=0 by 0 while (rc=0);
                     if _to ne from and h1.find(key:_to) ne 0 then
                            do;
                                   h1.replace(key:_to, data:_to); /*To add new 'To's to
```

```
                                                              Hash 'h1'*/
                                    rc=hit.first(); /*To reset searching point back to
                                           the top of the hiter 'hit'*/
                                    go to outer;
                             end;
                      else rc=h.find_next(key:new); /*Moving on to find next 'To'*/
               end;

               /*Inner do-loop is to keep adding new 'To's to Hash 'h1' */
               rc=hit.next();/*Moving down to next avaible 'From'*/
        outer:
        end;

        /*Outer do-loop is to exhaust 'To's for current 'From' on all levels*/
        /*The do-loop below is to catenate all of the 'To's in Hash 'h1'*/
        do rc=hit.first() by 0 while (rc=0);
               newvar=catx(' ',newvar,new);
               rc=hit.next();
        end;

        /*The do-loop above is to concatenate all of the 'To's in Hash 'h1'*/
        keep from newvar; /* Final output only has two variables: 'from' and 'newvar'*/
run;
```

DYNAMIC DATA SPLITTING

There are many ways to subset a SAS table conditionally. Both PROC SQL and the DATA Step (without the hash object) can provide methods to turn a big table into smaller ones. However, both approaches require that programmers at least know the number of output tables, even to prename them before starting the subsetting process. Here the hash object will provide a convenient solution with an above par efficiency.
See the following table. For the sake of simplicity, only one variable is included; however, the basic concept is not limited the number of variables.

Date
01/01/04
02/01/04
03/01/04
04/01/05
05/01/05
07/01/05
08/01/05
08/01/06
09/01/07
10/01/07
11/01/07
12/01/09
01/01/10

The table has been sorted by date. The requirement is to subset it based on continuity of “Month”. Different subsets are generated if there is a gap > 1 month. So for the example above, the final outcome will include 6 tables, with “01/01/04 02/01/04 03/01/04” being in the first table, and “12/01/09 01/01/10” in the 6th one. You have no idea how many such groups existing in the table, and what are they, unless additional data processing is done to collect this information and to store it in either macro variables or a separated table in the downstream process. This traditional

approach will not only have the overhead of requiring preprocessing the data, but also increases the complexity of the programming.

SOLUTION

The idea behind using the hash object in this application is to load the data to a hash table while monitoring the gap. When a gap is identified, the data in the hash table is output, and the contents of hash table is deleted. The same process is repeated and dynamic subsets are produced until the end of the input table is reached.

```
data have;
       input date mmddyy10.;
       format date date9.;
       cards;
01/01/04
02/01/04
03/01/04
04/01/05
05/01/05
07/01/05
08/01/05
08/01/06
09/01/07
10/01/07
11/01/07
12/01/09
01/01/10
;

data _null_;
       /*Below is to define a Hash 'h' to hold subset data*/
       if _n_=1 then
              do;
                     declare hash h(ordered:'a');/*'a' to preserve the accending
                                                   order*/
                     h.definekey('date');
                     h.definedata('date');
                     h.definedone();
              end;

       /*Above is to define a Hash 'h' to hold subset data*/
       set have end=last;

       /*Check the existance of gap (>1 month)*/
       if intck('month',lag(date),date) >1 then
              do;
                     /*if gap exists, group number increaded by 1*/
                     n+1;

                     /*if gap exists, output the subset with group number attached*/
                     rc=h.output(dataset:cats('split',n));

                     /*After the output, delete all the data in Hash 'h', preparing a
                       clean slate for the next subset*/
                     rc=h.clear();
              end;

       /*To output the last subset*/
       if last then
              do;
                     rc=h.replace();
                     rc=h.output(dataset:cats('split',n+1));
              end;
/*To load data onto Hash 'h'*/
       rc=h.replace();
```

```
run;
```

ANAGRAMS WITHIN WORDS

We know that an anagram is a type of word play, in which the letters of a word are rearranged to generate a new word, using all the original letters exactly once, for example, “cat” to “act”. This puzzle challenge is a variant from the classic anagram. It can be first found from the following source:
http://programmingpraxis.com/2014/02/21/anagrams-within-words/

Given two words, determine if the first word, or any anagram of it, appears in consecutive characters of the second word. For instance, “cat” appears as an anagram in the first three letters of “actor”, but “car” does not appear as an anagram in “actor” even though all the letters of “car” appear in “actor”. And if an anagram is present, output the results.

SOLUTION

There could be many ways to determine if an anagram match exists. One way of doing it is to break up the ‘word’ into ‘letters’ and compare them after sorting. Programmers will be able to envision that arrays seems to be a good mechanism for the job. As it is not that difficult to break a ‘word’ into an array with length of $1, and more importantly, Call Sortc() would be useful for the sorting. However, to make the code as robust as possible, it would be necessary to use the dictionary metadata tables for length information to predetermine the dimension of Array() before the matching process. Then, after the sorting, a reconcatenation may also be needed before the comparison. To summarize, if using Array(), it would be necessary to (1) Use the dictionary tables to get the length of ‘word’, and use it as the Array dimension (2) Breaking up ‘word’ in to ‘letters’ , and load them into Array (3) Sorting the Array() elements (4) Concatenating Array() elements after sorting (5)Comparing. Using the hash object,only steps (2) and (5) would be required. The size of a hash object is dynamic and is only limited by computer memory. The hash object provides a built in sorting mechanism, and it is easy to compare different hash objects using the equals() method.

```
/*Sample data: determine if 'container' contains the anagram of 'word'*/
data anagram;
       input ( word container ) ( : $100. );
       cards;
cat     actor
dinner  thundering
cab     actor
num     immunoglobulin
;
run;

data want;
       /*Do-loop below is to setup 2 Hash objects, 'h1' and 'h2', for downstream
         comparison*/
       if _n_=1 then
              do;
                     declare hash h1(ordered:'y', multidata:'y');
                     h1.definekey('letter');
                     h1.definedone();
                     declare hash h2(ordered:'y', multidata:'y');
                     h2.definekey('letter');
                     h2.definedone();
              end;

       /*Do-loop above is to setup 2 Hash objects, 'h1' and 'h2', for downstream
          comparison*/
       set anagram;

       /*To reset Hash 'h1' for each record*/
       _rc=h1.clear();

       /*Do-loop below is to load 'word' onto Hash 'h1', letter by letter, in
         accending order*/
       do _i=1 to lengthn(word);
              letter=substr(word,_i,1);
```

```
            _rc=h1.add();
      end;

      /*Do-loop above is to load 'word' onto Hash 'h1', letter by letter, in
           accending order*/

      /*Do-loop below is to block-load 'container' onto Hash 'h2', letter by letter,
        in accending order.
      For one time, only load the exact number of letters (a block) as 'word' has,
      then shifting one letter to the next block*/
      do _j=0 to lengthn(container)-lengthn(word);
            /*To reset Hash 'h2' between blocks*/
            _rc=h2.clear();

            /*Do-loop below is to extract blocks from 'container'*/
            do _i=1 to lengthn(word);
                  letter=substr(container,_j+_i,1);
                  _rc=h2.add();
            end;

            /*Do-loop above is to extract blocks from 'container'*/
            /*To determine the equality between 'h1' and 'h2'*/

            _rc=h1.equals(hash:
            'h2', result: _eq);
            /*If the equality is confirmed, then output and move on to the next
              record*/
            if _eq then
                  do;
                        output;
                        return;
                  end;
      end;

      /*Do-loop above is to block-load 'container' onto Hash 'h2', letter by letter,
        in accending order.
      For one time, only load the exact number of letters (a block) as 'word' has,
      then shifting one letter to the next block*/
      drop _: letter;
run;
```

MATRIX COMPUTATION

One of the most often encountered problems in a data management is the lookup, and SAS offers a rich arsenal of tools to tackle this issue. One can glimpse a non-exhausted list from this paper: Merging Data Eight Different Ways. However, lookups are often designed to match variable values in different tables. What if the match needed to be between variable values in one table and variable NAMES in another?

Let's say you have a Source table with variables containing binary values 1 or 0, something like this:

Value1	Value2	Value3	Value4
1	0	0	0
0	1	1	0
0	0	1	1
1	0	1	0
1	1	1	1

And another Lookup table that contains lookup values:

Value	Return
1	5
2	18
3	2
4	1

In the Lookup table, variable "Value" dictates the variable names in the Source table and (value = 1) reflects Value1, the first variable in Source table, (value=2) points to Value2, the second variable in Source table and so forth. Therefore, when A=1, then Return=5; when B=1, then Return=18, etc. Now you are asked to obtain the summary of all Values across the same row. Something like this (the commented column is for illustration only and is not ncluded in the final output):

Value1	Value2	Value3	Value4	Total
1	0	0	0	5 /*(5)*/
0	1	1	0	20 /*(18+2)*/
0	0	1	1	3 /*(2+1)*/
1	0	1	0	7 /*(5+2)*/
1	1	1	1	26 /*(5+18+2+1)*/

The same results can also be achieved using PROC FORMAT, except a format table based on the table above would need to be pre-built if there are many variables, and the customized format has to be applied to all of the variables.

SOLUTION

The key is to validate the link between variable positions in the Source table and the values in Lookup table. Here an array index is used to identify variable positions, and then the same index is used to load the Lookup table into a hash object.

```
/*Sample data: Source table*/
data source;
   input value1 value2 value3 value4;
   cards;
1     0 0     0
0     1 1     0
0     0 1     1
1     0 1     0
1     1 1     1
;;;;
   run;

/*Sample date: Lookup table*/
data lookup;
   input value return;
   cards;
1     5
2     18
3     2
4     1
```

```
;;;;

data want;
/*Loading Lookup table into Hash object h*/
   if _n_=1 then do;
      declare hash h(dataset:'lookup');
      h.definekey('value');
      h.definedata('return');
      h.definedone();
      call missing (value,return);
    end;
    set source;
    array _v value1-value4;
      do over _v; /*Here to use do-over on implicit array index*/
        if _v=1 then do;
          rc=h.find(key:_i_); /*_i_ is the automatic variable for implicit array
                                index*/
          total=sum(total,return);
         end;
      end;
        drop value return rc;
run;
```

UNCONVENTIONAL TRANSPOSING

In order to implement some statistical procedures or present the report in required formats, data needs to be delivered in a certain way. Suppose we have results of X number of races run by Y number of runners. Consider that the incoming data is something like this:

Finished Rank	Race1	Race2	Race3
1	David	Ethan	David
2	Adam	David	Bob
3	Ethan	Adam	Adam
4	Chris	Bob	Chris
5	Bob	Chris	Ethan

In the above data, there are 3 races run by 5 runners. It is not in an ideal format where you can easily evaluate the overall performance for individual runners. If we can convert the data into following form, then the computation of runners' rank becomes possible:

Runner Name	Rank_Race1	Rank_Race2	Rank_Race3
Adam	2	3	3
Bob	5	4	2
Chris	4	5	4
David	1	2	1
Ethan	3	1	5

SOLUTION

In following code, we will assemble the whole output inside the hash object. Using runner names as the hash key, updating the data elements whenever a key is 'find', then upload it into the hash table. When finished, output the whole hash table into a SAS table.

```
/*Incoming raw data*/
data input;
       input rank (race1-race3) ($);
       cards;
1 David Ethan David
2 Adam David Bob
3 Ethan Adam Adam
```

```
4 Chris Bob Chris
5 Bob Chris Ethan
;
run;

data _null_;
      if _n_ eq 1 then
            do; /*This is to define needed Hash object for output table*/
                  declare hash ha(ordered:'y');
                  ha.definekey('Runner_Name');

      ha.definedata('Runner_Name','Rank_Race1','Rank_Race2','Rank_Race3');
                  ha.definedone();
                  call missing(of Rank_Race1 - Rank_Race3);
            end;

      set input end=last;
      array r{3} race1-race3; /*Setup an array to go through all the runners race by
                                    race*/
      array o{3} Rank_Race1 - Rank_Race3; /*Setup an array to sync with the race
                                          array r(*)*/

      do i=1 to dim(r);
            if not missing(r{i}) then /*Cope with possible missing values*/
                  do;
                        Runner_Name=r{i}; /*Assing runner names to the Hash key*/
                        rc=ha.find();/*Download the most current data per this Hash
                                    key (runner name)*/
                        o{i}=rank; /*Update the data elements that needs to be
                                    updated*/
                        ha.replace();/*Upload the freshly updated data into the
                                    Hash object*/
                        call missing(of Rank_Race:); /*Reset data elements after
                                                uploading*/
                  end;
      end;

      if last then /*If reach the end of the incomming data, output the whole Hash
                  table into a SAS table*/
            ha.output(dataset:'want');
run;
```

RANDOM SAMPLING

There are many established sampling methods in SAS, including but not limited to the DATA step, PROC SQL and PROC SURVEYSELECT. What we are attempting to share here is the simplicity of using the hash object in random sampling; furthermore, its unique tangible flow would make users feel as if they were literally picking apples from a basket.

SOLUTION

The idea is to load the targeted table into a hash object. Then, depending on with or without replacement, a random sample is obtained. In the sample code, random sampling is performed based on the Uniform Distribution. The sample data set is SASHELP.CLASS, and the sample size is 10 observations. The sample code can be easily modified to do sampling based on proportions.

```
/*Random sample with replacement*/
data want_wr;
      if _n_=1 then /*Define Hash object*/
            do;
                  declare hash h(ordered:'a');
                  h.definekey('_n_'); /*Use data step implicit counter as the Hash
                                    key*/
                  h.definedata('name');/*for the purpose of demo, only only variable
                                    'name' has been included*/
```

```
                h.definedone();
        end;

    set sashelp.class (keep=name) end=last;
    _rc=h.replace(); /*Load data into Hash table*/

    if last then
        do while (1); /*This is to let go a infinite loop, which will be stopped
                        when certain condition is met*/
            _nobs=h.num_items; /*Get the total number of the observations in
                                 Hash table*/
            _rc=h.find(key:round(ranuni(1)*_nobs)); /*Random key ranges from 1
                    to the total number of the observations in the Hash
                       table*/
            _n+1; /*Count the sample size*/
            output;

            if _n ge 10 then /*when reaches targeted size, stop processing*/
                   stop;
        end;

    drop _:;
run;


/*Random sample without replacement*/
data want_wor;
    if _n_=1 then /*Define Hash object*/
        do;
            declare hash h(ordered:'a');
            h.definekey('_n_'); /*Use data step implicit counter as the Hash
                                  key*/
            h.definedata('name');/*for the purpose of demo, only only variable
                                   'name' has been included*/
            h.definedone();
        end;

    set sashelp.class (keep=name) end=last;
    _rc=h.replace(); /*Load data into Hash table*/

    if last then
        do while (1); /*This is to let go a infinite loop, which will be stopped
                        when certain condition is met*/
            _nobs=h.num_items; /*Get the total number of the observations in
                                 Hash table,
                                and not like the one in 'replacement code,
                               this number changes*/
            _key=round(ranuni(1)*_nobs);
            _rc=h.find(key:_key);

            do while (_rc ne 0); /*When there is NO corresponding keys,
                  this loop is fired up to keep generate keys randomly until
                  there is a hit*/
                   _key=round(ranuni(1)*_nobs);
                   _rc=h.find(key:_key);
            end;

            _rc=h.remove(key:_key); /*This is my favorite part: "NO
            replacement" is literally done!
            The data is removed from Hash table once it has been selected*/
            _n+1; /*Count the sample size*/
            output;

            if _n ge 10 then /*when reaches targeted size, stop processing*/
```

```
                        stop;
            end;

      drop _:;
run;
```

Conclusion

The SAS hash object offers many pleasant surprises for SAS programmers, and its potential and capacity has yet to be fully understood and explored. We hope this paper will generate some interest that will encourage more SAS users to use and experience the benefits of including the hash object in their work.

References

SAS(R) 9.3 Component Objects: Reference, SAS Institute Inc., Cary, NC, USA

Dorfman P and Eberhardt P 2011 "Two Guys on Hash."
Proceedings of the Annual Southeast SAS Users Group Conference, Alexandria, VA

Dorfman, P and Shajenko L. 2006 "Data Step Hash Objects and How to Use Them."
Proceedings of the Annual Northeast SAS Users Group Conference, Philadelphia, PA

Dorfman P and Snell G. 2003 " Hashing: Generations."
Proceedings of the Twenty-Seventh Annual SAS Users Group International Meeting, Seattle, WA

CONTACT INFORMATION

Haikuo Bian
Risk Compliance Department
Regions Bank
1900 5th Avenue North, 9th floor Birmingham, AL 35203
Phone: 205-264-4925
haikuo.bian@regions.com

Garland D. (David) Maddox, Jr
Business Loan Center – Performance Analytics and Reporting
Regions Bank
250 Riverchase Parkway East, Birmingham, AL 35244
Phone: 205-560-6339
Garland.MaddoxJr@Regions.com

Carlos Jimenez
Risk Compliance Department
Regions Bank
1900 5th Avenue North, 9th floor Birmingham, AL 35203
Phone: 205-264-5416
Carlos.m.Jimenez@regions.com