Paper BB-83

Hash Objects for Everyone

Jack Hall, OptumInsight

ABSTRACT

The introduction of Hash Objects into the SAS® toolbag gives programmers a powerful way to improve performance, especially when JOINing a large data set with a small one. This presentation will focus on the basics of creating and using a simple hash object, using an example from the Healthcare Insurance sector.

INTRODUCTION

Hash objects have many more capabilities than will be covered in this presentation. A number of papers have been presented at SESUG and other SAS Users Group meetings including SAS Global Forums. What I have found, reading these papers and attending some of these presentations, is that there was always something in the sample coding that was just not explained well enough to make me feel comfortable with hash objects.

So I decided to do some research and create a basic presentation and sample program that will let every SAS programmer understand, and feel comfortable with, hash objects.

First the basics, which have been covered in many other papers, and are presented here for the sake of completeness.

In the simplest usage (this is all we will cover here), a hash object allows a SAS programmer to easily set up and use a lookup table in memory. As with most things in SAS, there is more than one way to implement a lookup table. PROC FORMAT, the DATA step MERGE statement, and the JOIN command in PROC SQL can all be used for this purpose. The main advantage of the hash object, as opposed to a MERGE or a JOIN, is that the hash object resides in memory, whereas MERGE and JOIN operate on SAS data sets that are stored on disk. This can make the hash object approach significantly faster.

Here are some basic things to know about hash objects:

- Hash objects are created, live and die within a single DATA step. After the DATA step is done, the hash object is no more.
- Hash objects can be thought of as a temporary SAS table residing in memory.
- A hash object has one or more “KEY” fields, along with other fields.
- Hash objects are not currently implemented in PROC SQL.
- The commands for accessing hash objects are different from what you might be used to in standard DATA step programming, but they are not complex at all.

THE PROBLEM

The example I will use comes from a real life project that I was assigned last year. Another programmer who was familiar with hash objects had started the project and was reassigned before he could finish it. So I inherited a half-completed program that used hash objects. I had two choices – I could rewrite the code to replace the hash objects with JOINs, or I could keep the hash objects and use this as a learning experience, which is what I did.

My company has a database of insurance claims, and most of my work involves writing SAS programs to read through these claims to identify any that match a specific pattern, which could imply fraud, abuse or errors in medical coding. One such database has over 50 million claim records. For the project I will use here, we also had two relatively small tables, giving the maximum allowable dosage for injectable drugs, identified either by a generic ‘j-code’ or a more specific ‘ndc code’. This “ndc code” is the “National Drug Code”, identifying the drug and the vendor.

DESIGNING THE PROGRAM

The approach was this. First we would do a full scan of the claims table, extracting all claim records for injected drugs, within a certain time period. In some cases we had only the j-code, while in other cases we had both the j-code and the ndc code. We also had the number of units administered. Once we had this file created, we needed to compare the actual dosage with the maximum allowable dosage, to identify claims that were over the limit.

As stated above, there are many ways to accomplish this in SAS. We could read the two small tables into SAS arrays and code the lookup logic ourselves. Or we could use PROC FORMAT create a format file to match each j-code with a maximum dosage number, and another format file for ndc codes. You can probably think of other ways.

With the DATA – MERGE approach, we would use the j-code table and the ndc code table (both of which originally came to me as EXCEL files, by the way) to create two SAS data sets. We would then merge these with the main claims file, one at a time, after sorting, so that the claims file would then have variables on each row for ACTUAL DOSAGE, JCODE MAX and NDC MAX (some of these could be missing values). At this point we could create logic to use these variables to identify the aberrant claims and create a report. *SAS code using this approach can be found in APPENDIX A.*

In the case of PROC SQL – JOIN, the two lookup tables would be JOINed to the claims file with LEFT JOIN commands in one step, (no sorting required), resulting in the same output SAS data set as in the DATA – MERGE example. The main drawback with both of these techniques, as compared with using hash objects, is that we have to read everything from all three tables from disk storage, which is much slower than reading from memory. *SAS code using this approach can be found in APPENDIX B*.

THE SAS CODE TO CREATE THE DATA SETS

Here is the code to create the input data sets for this simple example.

** (In our real application, this dataset could have TENS OF MILLIONS of observations);*

```
DATA claims;
  input   @1  claim_id $10.
          @12 units      4.
          @17 paid       5.
          @22 jcode     $5.
          @28 ndc      $14. ;
CARDS;
A123456789   25  360 11111 11111ABCDEFGHI
B234567890  125  125 22222 22222BCDEFGHIJ
C345678901 2500 1280 33333 33333CDEFGHIJK
D456789012    5  675 44444 44444DEFGHIJKL
E567890123 1500  200 55555 55555EFGHIJKLM
;
```

** In our real application, these two data sets have only a few THOUSAND observations;*

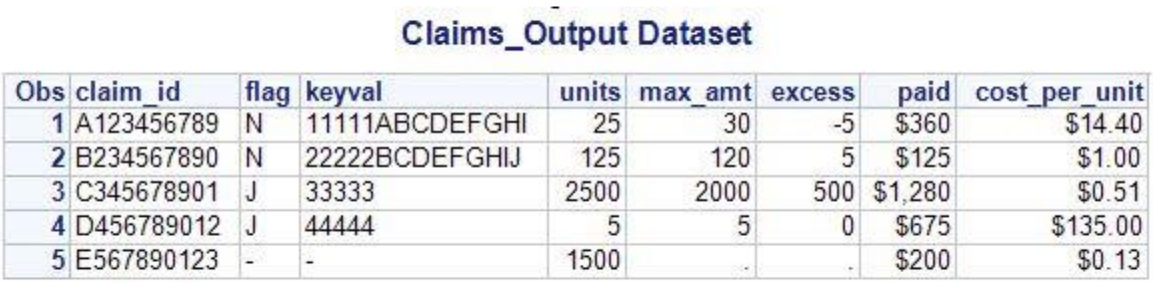
```
data ndc_map;
  input @1  ndc       $14.
        @16 max_units_ndc    4. ;
cards;
11111ABCDEFGHI   30
22222BCDEFGHIJ  120
22222ZYXWVUTSR  175
33333YXWVUTSRQ 2500
44444CDEFGHIJK   10
;

data jcode_map;
  input @1 jcode           $5.
        @7 max_units_jcode  4. ;
cards;
11111   20
22222  100
33333 2000
44444    5
;
```

THE MAIN SAS CODE, VERSION 1, USING HASH OBJECTS

```
Data Claims_Output (keep = claim_id flag keyval
         units max_amt excess paid cost_per_unit);
  retain claim_id flag keyval units max_amt excess
         paid cost_per_unit;
  length flag $ 1 keyval $ 14 max_amt 4;
  format paid dollar6. cost_per_unit dollar10.2;

/*******************************************************************************
 This next step does NOT read any data from the two lookup tables,
 but it DOES read and set up the HEADER information (formats, etc.)
 *******************************************************************************/

     if 0 then set ndc_map jcode_map;

/*************************************************************************************
  We want to create the two HASH tables only one time, since they will be
  retained in memory for the duration of this DATA step. Two ways to do this are:
  1) Use "_n_ = 1" as below, to create them only the first time through, or
  2) Wrap the rest of the DATA step within a "DO UNTIL ALLDONE" loop.
 *************************************************************************************/

     if _n_ = 1 then do;
       declare hash ndc_lookup (dataset: "ndc_map");
        * (this creates the hash object 'ndc_lookup'
           from the SAS data set 'ndc_map');

          ndc_lookup.definekey  ("ndc");
        * (this defines the field 'ndc' as the lookup key);

          ndc_lookup.definedata ("max_units_ndc");
        * (this is the field to be returned. You may have more
           than one field defined.);

          ndc_lookup.definedone();
        * (ends the declare statement);

          call missing(max_units_ndc);
        * (initializes the return field with missing values);

       declare hash jcode_lookup (dataset: "jcode_map");
         jcode_lookup.definekey  ("jcode");
         jcode_lookup.definedata ("max_units_jcode");
         jcode_lookup.definedone();
         call missing(max_units_jcode);
     end;
```

```
/* Get an observation from the Claims dataset, and set the "max_units" values that
   may be returned from the hash objects to missing values. */

  set claims;

  call missing(max_units_ndc);
  call missing(max_units_jcode);

/****************************************************************************************************************
 If the current value of the variable "ndc" IS FOUND in the ndc_look hash object,
 the "return code" will be 0, and we can then use the variable "max_units_ndc"
 ****************************************************************************************************************/

  if ndc_lookup.find() = 0 then do;
     excess  = units - max_units_ndc;
     max_amt = max_units_ndc;
     flag    = 'N';
     keyval  = ndc;
  end;

/*************************************************************************************************************************
 If the current value of the variable "ndc" IS NOT FOUND in the ndc_look hash object,
 we will look in the jcode_look hash object. If we find a match to the current value
 of the variable "jcode", we can then use the variable "max_units_jcode"
 ************************************************************************************************************************/

  else if jcode_lookup.find() = 0 then do;
     excess  = units - max_units_jcode;
     max_amt = max_units_jcode;
     flag    = 'J';
     keyval  = jcode;
  end;

* If we don't find a match for either ndc or jcode, set the calculated variables to missing.;

  else do;
     call missing(excess);
     call missing(max_amt);
     flag   = '-';
     keyval = '-';
  end;

  cost_per_unit = paid/units;
*  if excess > 0 then output;  * (remove '*' to output only 'bad' claims);
run;
```

THE OUTPUT FROM THIS PROGRAM



Claims_Output Dataset

Obs	claim_id	flag	keyval	units	max_amt	excess	paid	cost_per_unit
1	A123456789	N	11111ABCDEFGHI	25	30	-5	$360	$14.40
2	B234567890	N	22222BCDEFGHIJ	125	120	5	$125	$1.00
3	C345678901	J	33333	2500	2000	500	$1,280	$0.51
4	D456789012	J	44444	5	5	0	$675	$135.00
5	E567890123	-	-	1500	.	.	$200	$0.13

In this example, the claims file contains five records, but in the real application it might have several million records. The two lookup tables would have a couple thousand records each, which means they are small files relative to the claims file. And this is the pattern to look for when considering using hash objects – a large file using one or more smaller lookup files.

CONCLUSION

In this paper, we have looked at the basics of working with hash objects, specifically for the purpose of implementing lookup tables. Hash objects are of particular benefit in speeding up the execution of programs in which a very large data set uses one or more relatively small data sets as lookup tables.

Becoming comfortable with hash objects requires the programmer to develop a basic understanding of how objects and methods work, and also how SAS operates behind the scenes. The former is mostly about learning some new syntax, such as ’if ndc_lookup.find() = 0 …’, and what happens when it is executed. Fortunately, the basic methods are very simple to understand and use, and there is good SAS documentation about the more advanced methods. Also, papers from SAS Global Forums and regional SAS users groups contain good explanations of these.

Understanding how SAS operates behind the scenes is helpful in order to fully understand statements like ‘if 0 then set ndc_map;’. Since ‘if 0’ can never be true, this seems like a statement that will do nothing. And yet, because of the way SAS operates behind the scenes, the statement does cause SAS to get and remember the variable attributes of the data set ‘ndc_map’. Basic SAS courses usually include some discussion of the “Program Data Vector”, and how and when elements are loaded into it. Understanding this process will make you feel more comfortable and confident in all of your SAS programming,

REFERENCES

As I researched this topic, trying to understand what was necessary to create and use hash objects and what was optional, I came across several papers from past SESUG and SAS GLOBAL conferences which helped me put it all together. I recommend these for anyone just getting started using hash objects. I know there are many other papers out there, some of which I have read and some that I have not, so if you are interested, searching the proceedings for 'hash objects' will lead you to much good information.

Lafler, Kirk Paul. "An Introduction to SAS Hash Programming Techniques". Proceedings from the 2011 Southeast SAS User Group Meeting.

Dorfman, Paul M. and Eberhardt, Peter. "Two Guys on Hash". Proceedings from the 2010 Southeast SAS User Group Meeting.

ACKNOWLEDGEMENTS

CONTACT INFORMATION

Your comments and questions are welcome. Please feel free to contact the author at:

Jack Hall, Software Engineer
OptumInsight
(612) 632-6593
john.hall@optum.com
bigjake29662@charter.net

APPENDIX A

For comparison, here is some corresponding code using DATA / MERGE.

```
Proc SORT DATA=Claims;
  by jcode claim_id;

Proc SORT DATA=jcode_map;
  by jcode;

Data Claims_Output_1;
  Merge Claims (IN=IN1)  jcode_map (IN=INJ);
  by jcode;
  length flag $ 1 keyval $ 14 max_amt 4;
  If IN1;
  if INJ then do;
       excess  = units - max_units_jcode;
       max_amt = max_units_jcode;
       flag    = 'J';
       keyval  = jcode;
  end;
run;

Proc SORT DATA=Claims_Output_1;
  by ndc;

Proc SORT DATA=ndc_map;
  by ndc;

* NDC numbers overwrite JCODE numbers if they are available;

Data Claims_Output (keep = claim_id flag keyval
        units max_amt excess paid cost_per_unit);
  format paid dollar6. cost_per_unit dollar10.2;
  retain claim_id flag keyval units max_amt
         excess paid cost_per_unit;
  merge Claims_Output_1 (IN=IN1) ndc_map (IN=INN);
  by ndc;
  if IN1;
  if INN then do;
       excess  = units - max_units_ndc;
       max_amt = max_units_ndc;
       flag    = 'N';
       keyval  = ndc;
  end;
  cost_per_unit = paid/units;
*  if excess > 0 then output;
run;
```

APPENDIX B

Finally, here is some corresponding code using PROC SQL / JOIN.

```
PROC SQL NOPRINT;
  Create Table Claims_Output_1 as
  select a.*,
         b.max_units_ndc,
         c.max_units_jcode
  from      Claims    a
  left join ndc_map   b
    on a.ndc = b.ndc
  left join jcode_map c
    on a.jcode = c.jcode;
Quit;

data Claims_Output (keep = claim_id flag keyval
        units max_amt excess paid cost_per_unit);
  retain claim_id flag keyval units max_amt excess
        paid cost_per_unit;

  format paid dollar6. cost_per_unit dollar10.2;
set Claims_Output_1;

  if max_units_ndc > 0 then do;
       excess  = units - max_units_ndc;
       max_amt = max_units_ndc;
       flag    = 'N';
       keyval  = ndc;
  end;
  else if max_units_jcode > 0 then do;
       excess  = units - max_units_jcode;
       max_amt = max_units_jcode;
       flag    = 'J';
       keyval  = jcode;
  end;

  else do;
       call missing(excess);
       call missing(max_amt);
       flag    = '-';
       keyval  = '-';
  end;
  cost_per_unit = paid/units;
* if excess > 0 then output;
run;
```