Paper 4778-2020

Let's See Further than One Observation: Applying Hash Objects for Typical Clinical Programming Tasks

Valeriia Oreshko, IQVIA

ABSTRACT

It is commonly known that SAS® executes the data step for each observation iteratively. A common method that is used to compare the current record to data from previous records is to sort using first/last automatic variables and the retain statement. This method is often time-consuming, requires the creation of additional data steps and variables, additional sorting and is not always intuitive or easy to update. This paper will demonstrate an alternative approach to typical clinical programming through hash objects: defining baseline, detecting of repeated adverse events, flagging of all records from an interval and fuzzy merging. Hash objects allow for the loading of a whole dataset into memory and for the analyzing of combinations of observations. This paper will explain techniques on how to navigate through the hash table and provide a comparison of execution speed between code containing hash objects and traditional code.

INTRODUCTION

Hash object is a dynamic data structure. It can be summarized as an array that can be accessed by a key and uses methods to navigate it. The paper will show how hash object programming can be applied alongside with concepts of SAS®, that are more usual for clinical programmers. A detailed explanation of storing and retrieving data in a hash object is provided for each example, rather than providing a theoretical overview of the hash tools

EXAMPLE 1 – DETECTION OF REPEAT ADVERSE EVENTS

The first example demonstrates a method to flag repeat adverse event terms for a subject. Our source dataset contains three variables: SUBJ, VISN, TERM. We create the hash object from our source dataset to navigate through it in search of the repeat AE term. It is essential to rename variables in the hash object EVENTS (#1) to differentiate and compare the values in hash with those handled in a data step.

A hash object consists of keys and data items. During the declaration of the hash object at least one key must be defined to make a link with the data step. SUBJ and VISN were selected as keys (#2), because we want the data to be saved in hash in ascending order by SUBJ, VISN and there are no duplicate adverse events on one visit. The argument tag ORDERED specifies that SAS returns data in ascending order by SUBJ and VISN. The definition of data items is optional. Data items associated with key items will be retrieved from hash and stored in the dataset REPEATED. In DEFINEDATA() method we specify that all variables from source dataset should be placed in hash – in our case SUBJ, VISN and TERM (#3).

Furthermore, we declare and instantiate a hash iterator (hiter) object POINTER associated with object EVENTS (#4). This hiter enables retrieval of data from EVENTS in forward or reverse key order.

When we instantiated a new hash object with the DECLARE statement, we defined its characteristics (key, data), however SAS does not yet recognize these characteristics as variables. So, our next step is to use standard SAS statements (for ex., LENGTH) to add to

PDV variables with the same names as the object’s key and data. Otherwise we’ll receive an error. The CALL MISSING statement is used to remove log notes regarding uninitialized variables, i. e. those defined for the hash object (#5).

After setting the source dataset, the POINTER goes to the record in the EVENTS hash object that precedes the record currently read from dataset observation (#6). DO WHILE loop (#7) checks that the return code (assigned to RC variable) is equal 0 – meaning successful execution and that the desired record exists. Inside the loop we navigate through all preceding records (relative to current observation) of the hash object EVENTS and mark an adverse event as repeated if the same TERM presents in part of EVENTS that corresponds to the current SUBJ value.

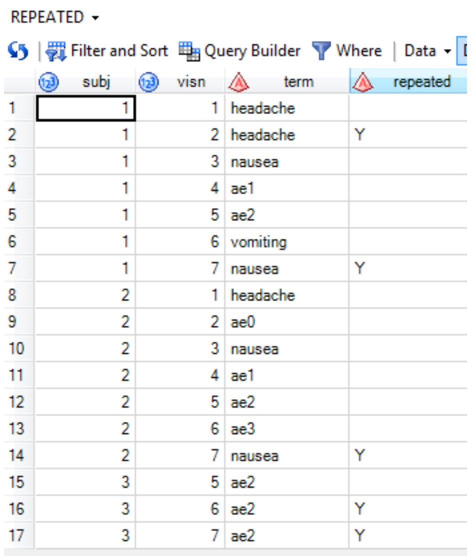
The code for the detection of repeat adverse events is provided below:

```
data repeated(keep= subj visn term repeated);
   if _N_ = 1 then do;
        declare hash events(dataset:'adv_ev(rename=(term=hash_term
#1                                  visn=hash_visn subj=hash_subj))',
                                    ordered: 'a');
         events.definekey('hash_subj', 'hash_visn');   #2
         events.definedata(all:"yes");   #3
         events.definedone();
      declare hiter pointer('events');   #4
   end;
   length hash_term $20 hash_subj 8.;
   call missing(hash_term, hash_subj);   #5
   set adv_ev;

   do i = 1 to _N_-1;
         rc = pointer.next();   #6
   end;

   do while(rc = 0);
         if subj = hash_subj and hash_term = term then repeated="Y";   #7
         rc = pointer.prev();
   end;
run;
```

In Display 1 the result of execution of code above is presented.



	subj	visn	term	repeated
1	1	1	headache	
2	1	2	headache	Y
3	1	3	nausea	
4	1	4	ae1	
5	1	5	ae2	
6	1	6	vomiting	
7	1	7	nausea	Y
8	2	1	headache	
9	2	2	ae0	
10	2	3	nausea	
11	2	4	ae1	
12	2	5	ae2	
13	2	6	ae3	
14	2	7	nausea	Y
15	3	5	ae2	
16	3	6	ae2	Y
17	3	7	ae2	Y

Display 1. Output dataset for example 1

DUPLICATES IN HASH OBJECTS

The default structure of a hash object only allows a unique set of data items per key value. If data loaded into a hash object contains duplicate key items, only the record with the first occurrence of key items will be stored in the hash object. Suppose, we need to remove repeated adverse events from prior ADV_EV dataset and only keep the occurrence of an adverse event with the latest VISN. The argument tags of the hash object declare statement have options to handle duplicates in dataset. It can be used as an analogue of NODUPKEY, NODUPREC options in proc sort.

At the beginning of the data step, the variables needed for the hash object are loaded into the PDV with the help of alternative method (#1 - instead of writing LENGTH statement). The condition of the "IF" statement is always false. Although, SAS will not execute the "SET", it will be compiled. During compilation all variables from the dataset named on the SET statement are added to the PDV.

In the standard DECLARE statement we add argument tag DUPLICATE: 'r', that tells SAS to store the last duplicate key record (#2).

The OUTPUT method (#3) loads the content from the hash object EVENTS to the dataset UNIQUE_AE, that can be used further in code. The sample of described code is provided below:

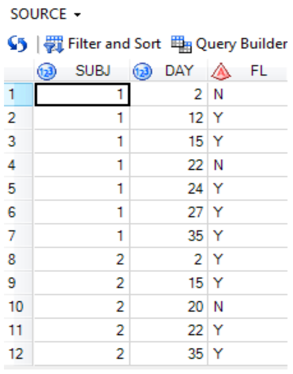
```
data _null_;
    if _N_= 0 then set adv_ev;          #1
    if _n_ = 1 then do;
          declare hash events(dataset: 'adv_ev', duplicate: 'r',
                              ordered: 'a');
          events.definekey('subj', 'term');          #2
          events.definedata(all: 'yes');
          events.definedone();
    end;
    rc = events.output(dataset: 'unique_AE');          #3
run;
```

EXAMPLE 2 – FLAGGING OF ALL RECORDS FROM AN INTERVAL

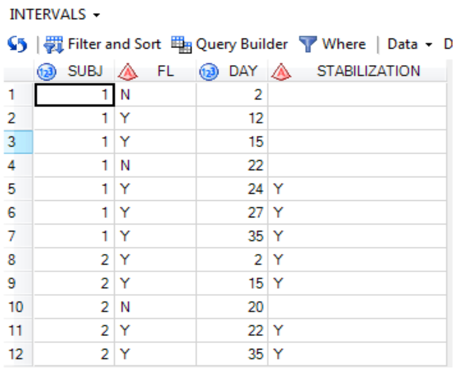
Suppose, we want to flag all the results that meet a specific condition continuously throughout a defined period – stabilization period. To decide if the current record from dataset should be included in the stabilization period, we need to look ahead to subsequent records ensuring they are within the interval. Hash object enables us to view the limits of stabilization period from the record that is currently handled in data step.

In this example we are looking for intervals where FL = “Y” during at least 10 days (Display 2).

The first block (#1) is responsible for setting the SOURCE dataset and, as seen in the prior example, for creating hash variables in the PDV. Further (#2), we create the hash object SRC and placing within it all data from the SOURCE dataset, specifying HASHSUBJ and HASHDAY as key items. The hash iterator linked to SRC object is also created at this step. After execution of the third block (#3) iterator HITER reaches the record in SRC object next to the observation currently read in data step.



	SUBJ	DAY	FL
1	1	2	N
2	1	12	Y
3	1	15	Y
4	1	22	N
5	1	24	Y
6	1	27	Y
7	1	35	Y
8	2	2	Y
9	2	15	Y
10	2	20	N
11	2	22	Y
12	2	35	Y



	SUBJ	FL	DAY	STABILIZATION
1	1	N	2	
2	1	Y	12	
3	1	Y	15	
4	1	N	22	
5	1	Y	24	Y
6	1	Y	27	Y
7	1	Y	35	Y
8	2	Y	2	Y
9	2	Y	15	Y
10	2	N	20	
11	2	Y	22	Y
12	2	Y	35	Y

Display 2. Input and output datasets for example 2

Let’s consider closely what happens for each observation of SOURCE dataset in the code step #4 taking _N_ = 5 as an example. It is essential to understand that during loop execution the record from the dataset will be the same and only hash records are searched.

Iteration 1 of “WHILE” LOOP (blue route in Figure 1):

- At the beginning TRACER is on the record with _N_ = 6 in hash.
- The condition to execute the body of the loop is true: subj=hashsubj =1, hashfl = “Y”, rc1 = 0, as record with _N_=6 (next to _N_=5 from dataset) exists in hash.
- Hashday - day = 27 – 24 = 3 < 10 - condition of “IF” statement is FALSE and tracer goes to the next record in hash (_N_=7).

Iteration 2 of “WHILE” LOOP (green route in Figure 3):

- TRACER is on the record with _N_ = 7 in hash.
- The condition to execute the body of the loop is true: subj=hashsubj =1, hashfl = “Y”, rc1 = 0, as record with _N_=7 exists in hash.
- Hashday - day = 35 – 24 = 11 > 10 - condition of “IF” statement is TRUE. The variable STABILIZATION in dataset is mapped with “Y” and retained until FL = “N” appears in dataset. “WHILE” loop ends for record with _N_ = 5 from dataset.

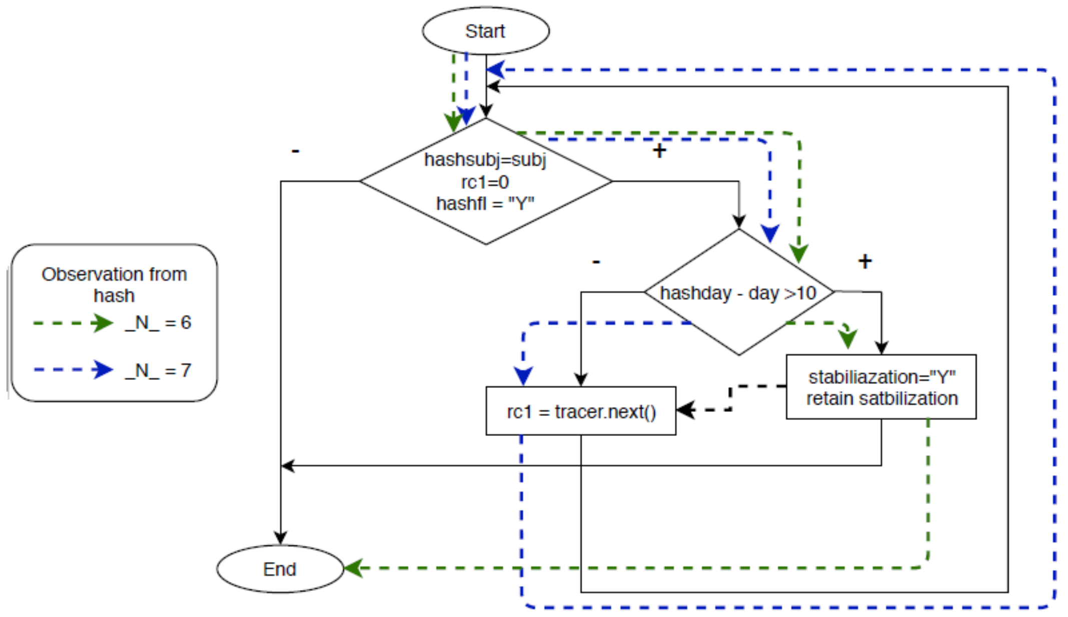


Figure 1. Scheme of work WHILE LOOP #4 when record with _N_= 5 is read from dataset SOURCE

The value “Y” for STABILIZATION will be retained in dataset until in data step the record with FL = “N” or new subject appears. Then we’ll begin to look for the new interval of stabilization once record with fl = “Y’ is read from the dataset. Here is the code for defining the needed interval:

```
data intervals(keep=subj fl day stabilization);
    set source;
    by subj day;
    length hashfl $1  hashday hashsubj  8. ;      #1
    call missing(hashfl, hashday, hashsubj);
    if _n_=1 then do;
          declare hash src (dataset: 'source(rename=(fl = hashfl day =
          hashday   subj  = hashsubj))',ordered: 'a');
          src.definekey('hashsubj', 'hashday');
#2        src.definedata(all:"yes");
          src.definedone();

          declare hiter tracer("src");
    end;
    rc1 = tracer.first();
    do i = 1 to _n_;  rc1 = tracer.next(); end;           #3
    if fl = "Y" and first.subj=0 then do;
          do while (subj = hashsubj and hashfl = "Y" and rc1 = 0);
                if hashday - day > 10 then do;
                      stabilization = "Y";
                      retain stabilization;
#4                    leave;
       end;
                rc1 = tracer.next();
          end;
    end;
    else call missing(stabilization);
run;
```

EXAMPLE 3 – MAPPING OF BASELINE FLAG

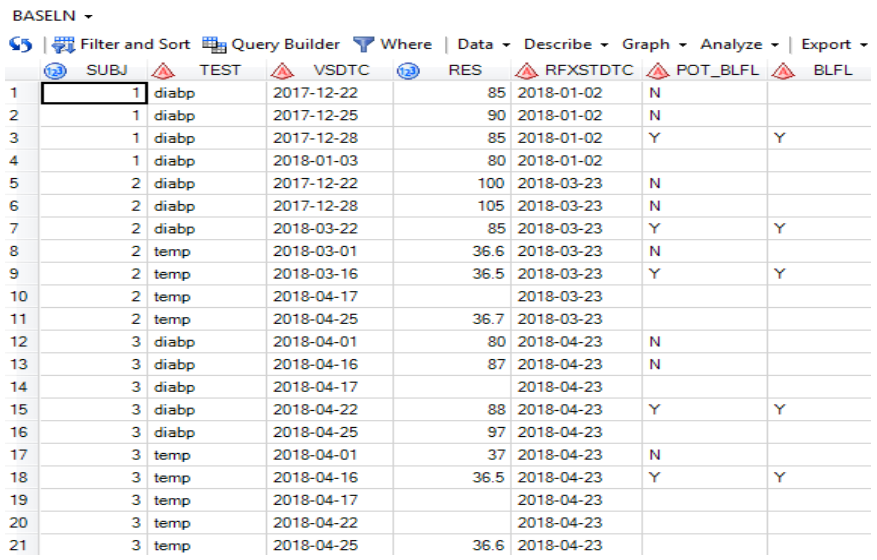
The example code below presents the method of detecting a baseline record for the finding dataset and outputting a flag back into the finding dataset using hash objects. To implement this example, we'll need 2 hash objects: the first one - with data about subjects' start treatment date, the second one containing data about tests performed.

After declaration of hash objects and iterator (#1), we will firstly look up the corresponding RFXSTDTC in the previously created hash table STTRT with the help of FIND method. This method connects handled dataset with hash table by the key variable (SUBJ) and uploads the variables specified in DEFINEDATA method (RFXSTDTC in our case). This step is the analogue of merge statement to get date of start treatment from demographic data.

When SAS reads a new record from a set dataset, our hash iterator TRACER is placed on the first observation of hash table that contains tests' results (#2). Using DO loop (#3) we are moving iterator to the next (relative to our current read from VS) observation. Then we check, if current record from VS can potentially be baseline one, (i.e., it has non-missing result and date is prior to start of treatment) and map all such records with POT_BL flag (#4). The goal of further executed WHILE loop (#5) is to define whether there is any potential baseline record after current one. If yes, the current record is not a true baseline, because there is later one but at the same time earlier that start of treatment. Note, that our data in hash table was sorted while declaring hash object through argument tag SORTED, that is why in WHILE loop the next element will always have the bigger VSDTC that current record has. In Display 3 you can see the resulting dataset with a mapped baseline flag.

The code for baseline mapping is presented below:

```
data baseln(drop= hash: rc: i);
     length rfxstdtc hashtest hashvsdtc $10 hashsubj hashres 8. ;
     call missing(rfxstdtc, hashtest, hashvsdtc, hashsubj, hashres);
     if _n_ = 1 then do;
    declare hash sttrt(dataset: 'dm');
           sttrt.definekey('subj'); sttrt.definedata('rfxstdtc');
           sttrt.definedone();
    declare hash tests(dataset: 'vs(rename=(subj=hashsubj test=hashtest
#1  vsdtc=hashvsdtc res=hashres))', ordered: 'a');
           tests.definekey('hashsubj','hashtest','hashvsdtc');
           tests.definedata(all:"yes");
           declare hiter tracer ("tests");
           tests.definedone();
     end;
     set vs;
     if sttrt.find() = 0;*DM merging;
     rc2 = tracer.first();                #2
     do i = 1 to _n_; rc2 = tracer.next(); end;             #3
     if vsdtc < rfxstdtc and not missing(res) then do; pot_bl = "Y";  #4
        do while(rc2 eq 0 and subj eq hashsubj and test eq hashtest);
           if hashvsdtc<rfxstdtc and nmiss(hashres)=0 then
           do;
                 pot_bl="N";
#5               leave;
           end;
             rc2 = tracer.next();
        end;
        if pot_bl = "Y" then blfl = "Y;
    end;
  run;
```



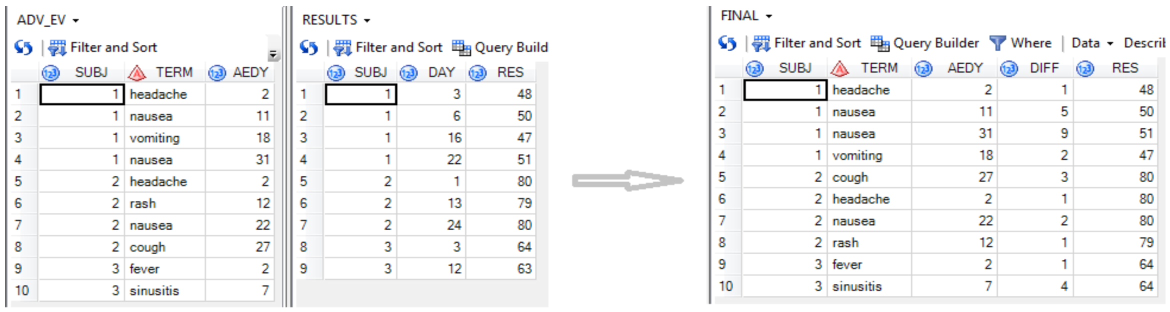
	SUBJ	TEST	VSDTC	RES	RFXSTDTC	POT_BLFL	BLFL
1	1	diabp	2017-12-22	85	2018-01-02	N	
2	1	diabp	2017-12-25	90	2018-01-02	N	
3	1	diabp	2017-12-28	85	2018-01-02	Y	Y
4	1	diabp	2018-01-03	80	2018-01-02		
5	2	diabp	2017-12-22	100	2018-03-23	N	
6	2	diabp	2017-12-28	105	2018-03-23	N	
7	2	diabp	2018-03-22	85	2018-03-23	Y	Y
8	2	temp	2018-03-01	36.6	2018-03-23	N	
9	2	temp	2018-03-16	36.5	2018-03-23	Y	Y
10	2	temp	2018-04-17		2018-03-23		
11	2	temp	2018-04-25	36.7	2018-03-23		
12	3	diabp	2018-04-01	80	2018-04-23	N	
13	3	diabp	2018-04-16	87	2018-04-23	N	
14	3	diabp	2018-04-17		2018-04-23		
15	3	diabp	2018-04-22	88	2018-04-23	Y	Y
16	3	diabp	2018-04-25	97	2018-04-23		
17	3	temp	2018-04-01	37	2018-04-23	N	
18	3	temp	2018-04-16	36.5	2018-04-23	Y	Y
19	3	temp	2018-04-17		2018-04-23		
20	3	temp	2018-04-22		2018-04-23		
21	3	temp	2018-04-25	36.6	2018-04-23		

Display 3. Output dataset for example 3

EXAMPLE 4 – FUZZY MERGING

As started in example 1, the default definition of a hash object excludes the duplicate records by key items. In this example we'll consider a method to place multiple sets of data per key value in hash and the advantages seen in manipulating duplicates in hash objects.

We'll use two datasets: ADV_EV (contains adverse event terms and the day of their occurrence) and RESULTS (results of one test and day when it was performed). Suppose, we want to find the result of the given test that was performed closest to (before or after) the adverse event (Display 4).



ADV_EV

	SUBJ	TERM	AEDY
1	1	headache	2
2	1	nausea	11
3	1	vomiting	18
4	1	nausea	31
5	2	headache	2
6	2	rash	12
7	2	nausea	22
8	2	cough	27
9	3	fever	2
10	3	sinusitis	7

RESULTS

	SUBJ	DAY	RES
1	1	3	48
2	1	6	50
3	1	16	47
4	1	22	51
5	2	1	80
6	2	13	79
7	2	24	80
8	3	3	64
9	3	12	63

FINAL

	SUBJ	TERM	AEDY	DIFF	RES
1	1	headache	2	1	48
2	1	nausea	11	5	50
3	1	nausea	31	9	51
4	1	vomiting	18	2	47
5	2	cough	27	3	80
6	2	headache	2	1	80
7	2	nausea	22	2	80
8	2	rash	12	1	79
9	3	fever	2	1	64
10	3	sinusitis	7	4	64

Display 4. Source and wanted datasets for example 4

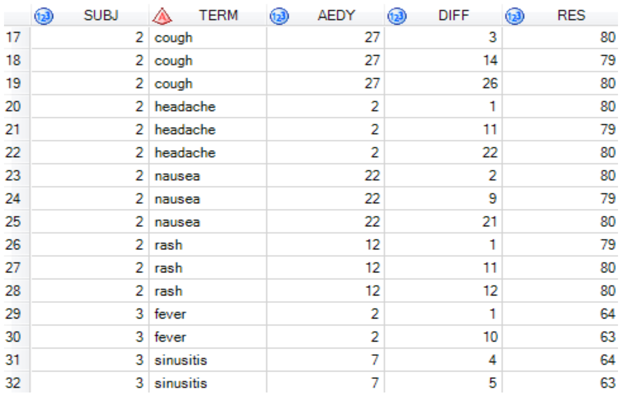
As we need the final dataset to have the same structure as ADV_EV, we'll handle ADV_EV in dataset and the RESULTS will be loaded in hash. The two source datasets have only one common key variable - SUBJ. The datasets contain more than one observation per subject, so we are forced to load multiple records per key value.

At the beginning of data step execution, we declare two hash objects. In the first RES_HASH – we place our source dataset with results. It is important to highlight that we are using argument tag "multidata" - with its help we can load several results per one value of variable SUBJ, that is defined as key item.

The second hash ALL_COMB is not filled immediately after declaration, as its DECLARE statement doesn't contain the 'DATASET:' argument tag. Further in loop #4 the data will be uploaded to ALL_COMB hash object using method ADD (#5).

In block #3 we set ADV_EV dataset with the END option to allow detection of the last observation. The method FIND() in this case retrieves the first set of data items for every subject from the hash object RES_HASH.

In #4 loop we calculate the difference between DAY and AESTDY for each adverse event and each result within one subject. In other words, we formulate a hash object that has cartesian products of adverse events*results for each subject. To navigate through RES_HASH we use the FIND_NEXT() method that is typical for objects with duplicate key items. It finds and places the next set of data items for the current key value in the data step. To see the result of loop #5 we store the content of ALL_COMB in dataset CHECK (#6 – Display 5)



	SUBJ	TERM	AEDY	DIFF	RES
17	2	cough	27	3	80
18	2	cough	27	14	79
19	2	cough	27	26	80
20	2	headache	2	1	80
21	2	headache	2	11	79
22	2	headache	2	22	80
23	2	nausea	22	2	80
24	2	nausea	22	9	79
25	2	nausea	22	21	80
26	2	rash	12	1	79
27	2	rash	12	11	80
28	2	rash	12	12	80
29	3	fever	2	1	64
30	3	fever	2	10	63
31	3	sinusitis	7	4	64
32	3	sinusitis	7	5	63

Display 5. Interim result – a piece (for subj = 2 and 3) of ALL_COMB hash object loaded in CHECK dataset

When the end of dataset ADV_EV is reached (#7), we already know all of the differences between days of result and adverse event – they are stored in CHECK dataset in DIFF variable. Now we just need to select the result with minimum DIFF for each subject, adverse event. We then create a hash object from CHECK dataset with key that specifies unique adverse event and loads only first occurrence. The dataset CHECK is sorted by DIFF because it was made from ALL_COMB hash object whose definition contains the argument tag 'ordered' and DIFF as key item. So, the observation loaded in MIN_COMB hash object and then outputted in FINAL dataset has the minimum difference between days of adverse event and result.

The last two DELETE methods (#8) remove hash objects ALL_COMB and RES_HASH during data step execution as they are no longer needed. Step #8 is not mandatory, the hash object would be deleted automatically by SAS at the end of data step, but it demonstrates that memory can be manually cleared if resources are limited. The full code for the described example is listed below:

```
data _null_;
if _n_ = 0 then set results;
    if _n_ = 1 then do;
          declare hash res_hash(dataset: 'results', multidata:"yes");
          res_hash.definekey('subj');
          res_hash.definedata(all:"yes");                            #1
          res_hash.definedone();
          declare hash all_comb(ordered: 'yes' multidata:  "yes");
          all_comb.definekey('subj', 'term', 'aestdy', 'diff');
          all_comb.definedata('subj', 'term', 'aestdy', 'diff', 'res');  #2
```

```
            all_comb.definedone();
        end;
        set adv_ev end = last;
        rc = res_hash.find();          #3

        do while (rc=0);
                diff = abs(day-aestdy);
                rc1 = all_comb.add(key: subj, key: term, key: aestdy,
   #5                              key: diff, data: subj, data: term,       #4
                                   data: aestdy, data: diff, data: res);
                rc = res_hash.find_next();
        end;
        rc3 = all_comb.output(dataset: 'check');          #6
        if last then do;
            declare hash min_comb(dataset: 'check', ordered: "yes");
            min_comb.definekey('subj', 'term' , 'aestdy');
            min_comb.definedata('subj', 'term', 'aestdy', 'diff', 'res');
            min_comb.definedone();                                          #7
            rc4 = min_comb.output(dataset: 'final');
            all_comb.delete();
            res_hash.delete();    #8
        end;
    run;
```

SPEED OF EXECUTION

Execution time was measured for the examples provided in paper and compared with the execution times of standard SAS code that perform the same task. The results are provided in Table 1.

N obs in dataset placed in hash	Example 1		Example 2		Example 3		Example 4	
	hash	no hash	hash	no hash	hash	no hash	hash	no hash (SQL)
1 000	0.01	0.03	0.04	0.08	0.12	0.02	0.15	0.03
10 000	0.04	0.05	0.06	0.19	0.15	0.05	0.19	0.06
100 000	0.05	0.27	0.09	0.48	0.22	0.34	0.21	0.41
1 000 000	0.08	2.38	1.12	4.15	0.25	2.99	0.22	3.79
5 000 000	1.35	26.08	4.21	44.31	6.31	19.81	6.28	40.81

Table 1. Comparison of execution time: hash vs standard code

Two major benefits of hashing are performing a table lookup using two unsorted datasets and as seen in table 1 a low and almost consistent runtime when scaled. When matching two tables by a variable it is better to place the larger of the two into hash. For an appropriately implemented hash data structure, the cost of lookups is independent of the number of elements stored in the table.

FAQ WHILE WORKING WITH HASH OBJECTS

1. Which dataset should be uploaded to hash and which should be set? – Consider the form of your final dataset, you should set the dataset that has similar structure to your final one. A dataset in hash is like a stable database.

2. When do we need to rename variables? - When the same dataset is uploaded to hash as is set in the data step.
3. How long does a hash object persist? – It exists only during execution of the data step in which the hash object was created. So, if you need to use previously created hash object in other data step, you should recreate it in current data step. The content of hash object can be saved for further usage in the dataset (with help of output() method), but it does not save the hash object.
4. How much memory does a hash object occupy? – A rough estimation can be made by multiplying the number of observations by the observation length. For a more precise estimation the attribute use ITEM_SIZE.

CONCLUSION

In clinical programming there are many situations (sorting, merging, complex manipulation with multiple records) where using hash tables to store data in memory can be a useful technique. The limit of memory can be an obstacle while using a hash object, as the amount of data that can be loaded into a hash object is the amount of memory available to the SAS session. It is desirable to clear memory manually using appropriate methods while working with hash objects. At first glance, the hash approach can be perceived as sophisticated one, so it often more reasonable to use this approach in stable pieces of code such as standard macros, that require less day to day maintenance.

REFERENCES

Burlew, Michele M. 2012. SAS® Hash Object Programming Made Easy. Cary, NC: SAS Institute Inc.

P.Dorfman, L.Shajenko. Data Step Programming Using the Hash Objects, NESUG 2004, Baltimore, MD 2004.

CONTACT INFORMATION

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

Valeriia Oreshko
IQVIA
valeriia.oreshko@quintiles.com