PharmaSUG 2017 - Paper TT09

Hashtag #Efficiency! An Exploration of Hash Tables and Merge Techniques

Deanna Naomi Schreiber-Gregory, Henry M Jackson Foundation, Bethesda, MD; Lakshmi Nirmala Bavirisetty, South Dakota State University, Cleveland, OH; Kaushal Chaudhary, Eli Lilly, Indianapolis, IN

ABSTRACT

Have you ever had to walk away from your computer during an analysis? Have you wondered if there is a way to increase your efficiency, save time, and be able to answer more questions? Hash tables to the rescue! This paper covers a brief introduction to the use of hash tables, their definition, benefits, concept, and theory. It also includes a review of some more applied approaches to hash table usage through code examples and applications that illustrate how using of hash tables can help improve performance time and coding efficiency. This paper will wrap up by providing a comparison of performance times between hash tables and traditional lookup and join/merge methods. An additional discussion of instances in which hash tables may not be the most effective method will also be explored. This paper is intended for any level of SAS® user who would like to learn about how hash tables can help process efficiency!

INTRODUCTION

As the size of real world data increases into terabytes and petabytes, the aggregation of data is taking a gradually larger amount of time and RAM memory to process and store. In the case that we want to merge a large amount of observations while still being conservative on memory and efficient with processing speed, we can explore the option of using hash tables. Hash tables provide us with the ability to significantly reduce the speed of processing while also decreasing our memory usage. This paper provides details on the construction of hash tables and how they can be used in SAS programming. We will explore how by using hash objects for data aggregation, we can reduce the run time processing and memory utilization compared to other procedures such as: PROC SUMMARY, PROC MEANS and PROC SQL and utilization of the MERGE statement. We will discover that in general, when we have to merge complex keys and multiple variables, the utilization of hash tables produces efficient and conservative results.

DEFINITION OF HASH TABLES

A hash table is an abstract array which is accessed by the program by using keys which allow any desired value to be used as an index. This index is called a key, and the contents of the array element at that index is called the value. In otherwords, a hash table is a data structure that stores key/value pairs and can be quickly searched by the key.

When we are dealing with duplicated keys (i.e. where two keys map to same index) then unintentional collision may occur. To resolve this collision in hash tables we would begin by pushing a colliding key onto the list, thus preserving both values without any collateral damage. We would then search for the key by hashing the index and searching the list. Lastly, we would then delete the duplicated key through the linked list. If this is the desire result, it is also worthy to note that there are several types of algorithms for dealing with collisions, such as linear probing and separate chaining.

HASH TABLES IN SAS

A hash iterator object is always associated with a hash object and a set of optional, predefined statements and procedures. The steps to creating a hash table are as follows: 1) a hash object is initiated by DECLARE statement; 2) a hash object performs any combination of the following methods and statements.

- DECLARE : The statement used to create the hash object or hash iterator object with following HASH or HITER statements. (e.g., DECLARE HASH hash_name(); and DECLARE HITER hash_iter_name();)
- DEFINEKEY() : define the single or composite key(s) that are either or complex variables for the hash object which is defined in DECLARE statement. These keys make a linkage between the DATA step and hash object.
- DEFINEDATA() : The data items are defined for this procedure by using LENGTH/ATTRIB statements in a DATA step.
- DEFINEDONE() : This function is used to complete defining the hash object i.e. the dataset which is defined with declare statement is loaded into a hash object.
- FIND() : returns 0 if the value found in the master dataset or else it returns different value.

HASH TABLE BASICS

The following tables include a comprehensive review of hash table statements and their definitions. This information is compiled from a review of the SAS®9 Hash Object Tip Sheet.

SAS Statement	Definition
declare hash obj(); **declare hash *obj*(dataset:** '*dataset_name*', **duplicate:** 'replace' | 'error', **hashexp:** *n*, **ordered:** 'a' | 'd' | 'no', **suminc:** '*count_var*');	Creates a hash object with the properties: **dataset:** loads the hash object from a data set. **duplicate:** controls how duplicate keys are handled when loading from a data set. **hashexp:** ***n*** declares 2^{n} slots for the hash object. **ordered:** specifies a key sort order when using a hash iterator or the **output** method. **suminc:** *count_var* contains the increment value for a key summary that is retrieved by the **sum** method.
rc = ***obj*.defineKey**('*key _var1* ', …, '*key_varN*'); rc = ***obj*.defineKey**(all: 'yes');	Defines a set of hash object keys given by *key_var1*…*key_varN*.
rc = ***obj*.defineData**('*data_var1*', …, '*data_varN*'); rc = ***obj*.defineData**(all: 'yes');	Defines data, given by *data_var1*…*data_varN*, to be stored in the hash object.
rc = ***obj*.defineDone**();	Indicates that key and data definitions are complete.
rc = ***obj*.add**(); rc = ***obj*.add**(key: *key_val1*, …, key: *key_valN*, data: *data_val1*, …, data: *data_valN*);	Adds the specified data associated with the given key to the hash object.
rc = **obj.find**(); rc = **obj.find**(key: key_val1, …, key: key_valN);	Determines whether the given key has been stored in the hash object. If it has, the data variables are updated and the return code is set to zero. If the key is not found, the return code is non-zero.
rc = **obj.replace**(); rc = **obj.replace**(key: key_val1 ,…, key: key_valN, data: data_val1, …, data: data_valN);	Replaces the data associated with the given key with new data as specified in data_val1…data_valN.
rc = **obj.check**(); rc = **obj.check**(key: key_val1, …, key: key_valN);	Checks whether the given key has been stored in the hash object. The data variables are not updated. Return codes are the same as for find.

rc = **obj.remove**(); rc = **obj.remove**(key: key_val1, …, key: key_valN);	Removes the data associated with the given key.
rc = **obj.clear**();	Removes all entries from a hash object without deleting the hash object.
rc = **obj.output**(dataset: 'dataset_name');	Creates dataset dataset_name which will contain the data in the hash object.
rc = **obj.sum**(sum: sum_var); rc = **obj.sum**(key: key_val1, …, key: key_valN, sum: sum_var);	Gets the key summary for the given key and stores it in the DATA Step variable sum_var. Key summaries are incremented when a key is accessed.
rc = **obj.ref**(); rc = **obj.ref**(key: key_val1 , …, key: key_valN);	Performs a find operation for the current key. If the key is not in the hash object, it will be added.
rc = **obj.equals**(hash: 'hash_obj', result: res_var);	Determines if two hash objects are equal. If they are equal, res_var is set to 1, otherwise it is set to zero.

Table 1. Hash Object – Methods

SAS Statement	Definition
i = ***obj*.num_items**;	Retrieves the number of elements in the hash object.
sz = ***obj*.item_size**;	Obtains the item size, in bytes, for an item in the hash object.
rc = ***obj*.delete**();	Deletes the hash object.

Table 2. Hash Object – Attributes

SAS Statement	Definition
declare hiter *iterobj*('*hash_obj*');	Creates a hash iterator to retrieve items from the hash object named *hash_obj*.
rc = ***iterobj*.first**();	Copies the data for the first item in the hash object into the data variables for the hash object.
rc = ***iterobj*.last**();	Copies the data for the last item in the hash object into the data variables for the hash object.
rc = ***iterobj*.next**();	Copies the data for the next item in the hash object into the data variables for the hash object. A non-zero value is returned if the next item cannot be retrieved. Use iteratively to traverse the hash object and return the data items in key order. If **first** has not been called, **next** begins with the first item.
rc = ***iterobj*.prev**();	Copies the data for the previous item in the hash object into the data variables for the hash object. A non-zero value is returned if the next item cannot be retrieved. Use iteratively to traverse the hash object and return the data items in reverse key order. If **last** has not been called, **prev** begins with the last item.

Table 3. Hash Iterator – Methods

Given that the aim of this paper is to compare the efficiency of hash tables with other procedures when accomplishing a common task, we will be comparing the processing time and memory usage of each of these procedures in three different examples.

EXAMPLE 1: MERGE TABLES USING MERGE STATEMENT, PROC SQL, AND HASH

Our first example covers table merging by comparing the `MERGE` statement, `PROC SQL`, and hash table procedure processing time and memory usage. The datasets used in this example were downloaded from the Kaggle website at https://www.kaggle.com/c/grupo-bimbo-inventory-demand/data. This data was proved as an essential aspect of a 2016 Grupo Bimbo competition during which participants were given the opportunity to forecast the demand of a product at a particular store for a given week. Of the provided datasets, which consisted of 9 weeks of sales transactions (sales and returns) in Mexico, the producto_tabla.csv and test.csv datasets were chosen for this comparison. These data sets in particular contained the product names of interest and test numbers, respectively.

USING MERGE

When we are working with large datasets with the MERGE statement, it is cumbersome as SAS requires all the datasets to be sorted or indexed before merging either with single or composite variables. The below example gives code and results for merging two datasets with a single variable: product_id.

```
proc sort data=test_data;
      by producto_id;
run;

data merge_results;
      merge test_data(in=id1) product_data(in=id2);
      by producto_id;
      if id1;
run;
```

Here is the log window for using `PROC SORT`:

```
NOTE: There were 6999252 observations read from the data set WORK.TEST_DATA.
NOTE: The data set WORK.TEST_DATA has 6999252 observations and 7 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time             14.32 seconds
      user cpu time         3.65 seconds
      system cpu time       0.76 seconds
      memory                662004.48k
NOTE: There were 2593 observations read from the data set WORK.PRODUCT_DATA.
NOTE: The data set WORK.PRODUCT_DATA has 2593 observations and 2 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time             0.00 seconds
      user cpu time         0.01 seconds
      system cpu time       0.00 seconds
      memory                1327.78k
NOTE: There were 6999252 observations read from the data set WORK.TEST_DATA.
NOTE: There were 2593 observations read from the data set WORK.PRODUCT_DATA.
NOTE: The data set WORK.MERGE_RESULTS has 6999252 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time             23.38 seconds
      user cpu time         0.90 seconds
      system cpu time       0.71 seconds
      memory                3826.65k
```

From the above SAS Log, we can see that by using `MERGE` - SAS reads all the data into memory so the processing take long time in process. The merging of two datasets containing a simple variable yielded a processing time of: 14.32 + 23.38 = ~ 37.7 seconds.

USING PROC SQL

Merging datasets using PROC SQL is toted as a more efficient method compared to the MERGE statement. However, with PROC SQL the datasets are stored into memory in order to combine datasets. Merging datasets is done through PROC SQL with a left join to merge one-to-many method.

```
Proc sql;
       create table SQL_Results as
       select *,coalesce(t.producto_id,p.producto_id) as producto_id
       from test_data t left join product_data p
       on t.producto_id=p.producto_id;
quit;
```

Here is the log window for using PROC SQL:

```
NOTE: Table WORK.SQL_RESULTS created, with 6999252 rows and 8 columns.
NOTE: PROCEDURE SQL used (Total process time):
       real time             29.39 seconds
       user cpu time         4.66 seconds
       system cpu time       1.39 seconds
       memory                554585.18k
```

From the above SAS Log, we see that PROC SQL was able to give us some more efficient results as far as processing time goes, thought we did not see an improvement in memory usage. For this example, it also did not require us to sort the datasets within the sql procedure. However, it is worthy to not that this with procedure larger datasets will prove more difficult as SAS will load the entire dataset into memory during processing.

USING HASH

There are a few additional notes to review when considering the implementation of a hash table for this example:

- For the iteration table, we will need to define the variables by either using the length statement or attrib statement. If you are looking to add any other options like format or label, then it would be a good idea to use the attrib statement in order to define the desired variables for the hash object.
- By using the declare statement we are creating hash object with the name h_product, and multidata is defined 'y' option indicates that multiple set of data items are allowed for each key value for the resultant dataset. The hash object is mainly created with two steps: 1) from the DefineKey(), Product_id is the key for the hash object which is not necessary to be unique. The following example demonstrates how to deal with unique values; 2) from the DefineData(), defining all other variables which are defined by attrib statement with all='y' option. rc indicates that if the key and the values are in the big dataset (product_data) and not in small dataset (test_data) then rc will return a value other than 0 whenever h_product.find() is processed within big dataset.
- Once the hash object is created - by using the iteration table, h_product - then we merge the dataset through test_data by using a do until loop. If rc returns a value other than 0 (i.e. the producto_id value is in test_data but not found in the hash object table), then we have to display a missing value for the resulting dataset, hash_merge, otherwise the previous values will be used and the wrong values displayed.

```
data hash_merge;
       attrib producto_id length=5 label='Product_id'
              nombreproducto length=$40. label='no_products';
       declare hash h_product(dataset:"product_data",ordered:'y',multidata:'y');
              rc=h_product.DefineKey("producto_id");
              rc=h_product.DefineData(all:'yes');
              rc=h_product.DefineDone();
       do until (eof1);
```

```
                set test_data end=eof1;
                rc=h_product.find();
                if rc ne 0 then
                        do;
                                call missing(nombreproducto);
                        end;
                output;
                drop rc;
        end;
run;
```

Here is the log window for using a hash table:

```
NOTE: There were 2593 observations read from the data set WORK.PRODUCT_DATA.
NOTE: There were 2593 observations read from the data set WORK.PRODUCT_DATA.
NOTE: There were 6999252 observations read from the data set WORK.TEST_DATA.
NOTE: The data set WORK.HASH_MERGE has 6999252 observations and 8 variables.
NOTE: DATA statement used (Total process time):
        real time               27.99 seconds
        user cpu time           1.37 seconds
        system cpu time         0.69 seconds
        memory                  5121.98k
```

From the log window results, we can plainly see that our hash table utilized significantly less time and memory to process than our previous two methods when performing the same task.

EXAMPLE 2: DATA AGGREGATION BY USING PROC SUMMARY AND HASH

Our second example covers data aggregation by comparing PROC SUMMARY and hash table processing time and memory usage. The data used in this example was downloaded from the Kaggle website at https://www.kaggle.com/c/GiveMeSomeCredit/data?. This data was provided as part of a 2011 competition in which participants were given the opportunity to build a model that financial consumers could use to make investment decisions. The dataset chosen was cs-trainng.csv which consisted of training profiles provided by the sponsored company.

USING PROC SUMMARY

```
proc summary data=training noprint;
        class numberrealestateloansorlines;
        var month_income;
        output out=agg_result(drop=_: rename=(month_income=total) )
                sum=month_income;
run;
```

The resulting log window results are given below in order to compare processing time and memory usage of the two procedures:

```
PROCEDURE SUMMARY used (Total process time):
        real time               0.01 seconds
        user cpu time           0.01 seconds
        system cpu time         0.01 seconds
        memory                  10269.65k
        OS Memory               42176.00k
```

USING HASH

Since we already know that the hash table dataset product_data has unique observations, it is not necessary to define the **`multidata`** option for hash declaration. The hash iterator object `hiter` transforms the hash object `hh_test` into a do until loop with an order specified tag to receive the data from the hash object in ascending order.

```
data hash_results;
        attrib numberrealestateloansorlines length=3 label='ssssss'
```

```
        total length=8;
    declare hash hh_test(suminc:'month_income',ordered:'y');
    declare hiter hh_iter('hh_test');
    hh_test.definekey('numberrealestateloansorlines');
    hh_test.definedone();

    do until (eof);
        set training end=eof;
        hh_test.ref();
    end;

    rc=hh_iter.first();

    do while (rc=0);
        hh_test.sum(sum:total);
        output;
        rc=hh_iter.next();
    end;

    keep numberrealestateloansorlines total;
run;
```

The resulting log window results are given below:

```
NOTE: The data set WORK.HASH_RESULTS has 28 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time            0.02 seconds
      user cpu time        0.03 seconds
      system cpu time      0.00 seconds
      memory               2744.21k
      OS Memory            33708.00k
```

In review of these results, along with their comparison to the previously derived `PROC SUMMARY` results, we see that that the processing time for these procedures ended up being quite similar, while the memory usage of `PROC SUMMARY` was significantly greater than that of hash objects.

EXAMPLE 3: MERGE TABLES FOR DUPLICATE OBSERVATIONS BY USING MERGE STATEMENT AND HASH

The duplicate key in the hash object can be handled by adding the "`multidata: 'yes'`" argument in the declare statement. An example of this method is given below using an independently created dataset of sample variables defined using `datalines`:

```
data A;
    input key var1;
    datalines;
    1 11
    1 20
    2 15
    2 24
    4 14
    5 50
    ;
run;

data B;
    input key var2;
    datalines;
    1 5
    2 6
    3 7
    4 8
    ;
run;
```

```
data C(drop=rc);
        if 0 then
                set A;
        if _n_=1 then
                do;
                        dcl hash h(dataset:'A', ordered:'A', multidata:'yes');
                        h.definekey('key');
                        h.definedata('var1');
                        h.definedone();
                end;
        set B (keep=key var2);
        rc=h.find();
        do while(rc=0);
                output;
                rc=h.find_next();
        end;
run;
```

CONCLUSION

From the different datasets, we concluded that before the version of SAS9.2, merging large datasets, data aggregation and handling duplicate observations was tricky and time consuming. With the introduction of hash tables, these processes became not only easier to implement, but also more efficient. Even though each hash object technique has its unique set of pros and cons, it does not require the sorting before merging (pros) and does not increase the complexity of a procedure due to an increase in data set size (cons). The purpose of this paper was to serve and introduction to how hash programming can make your programming more efficient when handling with large datasets.

ACKNOWLEDGEMENTS

The authors would like to thank Richann Watson and Lynn Mullins for their guidance in the preparation of a new draft of a previous iteration of this presentation. Thank you for your help!!

REFERENCES / ADDITIONAL READING

Burlew, M. M. 2012. *SAS Hash Object Programming Made Easy*. Cary, NC: SAS Insitute.

Grupo Bimbo. (2016). Grupo Bimbo Inventory Demand [test.csv & product_tabla.cv]. Retrieved from: https://www.kaggle.com/c/grupo-bimbo-inventory-demand/data.

Kaggle. (2011). Give Me Some Credit [cs-training.csv]. Retrieved from: https://www.kaggle.com/c/GiveMeSomeCredit/data?

Lafler, K. P. 2011. “An Introduction to SAS Hash Programming Techniques” *Proceedings of Southeast SAS Users Group Conference*. Alexandria, VA: SAS.

Loren, J. 2008. “How Do I Love Hash Tables? Let Me Count The Ways!” *Proceedings of SAS Global Forum 2008*. San Antonio, TX: SAS.

Muriel, E. 2007. “Hashing Performance Time with Hash Tables” *Proceedings of SAS Global Forum 2007*. Orlando, FL: SAS.

SAS Community. “Data Aggregation Using the SAS Hash Object.” https://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf/.

SAS Support. “Hash Tip Sheet.” https://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf/.

Snell, G. P. 2006. “Think FAST! Use Memory Tables (Hashing) for Faster Merging” *Proceedings of SUGI 31*. San Francisco, CA: SAS.

Warner-Freeman, J. K.. 2007. “I cut my processing time by 90% using hash tables – You can do it too!” *Proceedings of Northeast SAS Users Group Conference 2007*. Baltimore, MD: SAS.

Watson, R. & Mullins, L. 2016. “Exploring HASH Tables vs SORT/DATA Step vs PROC SQL” *Proceedings of Pharmaceutical SAS Users Group Conference 2016.* Denver, CO: SAS.

CONTACT INFORMATION

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

Lakshmi Nirmala Bavirisetty
Masters Graduate
South Dakota State University
plnimmi@gmail.com

Deanna Naomi Schreiber-Gregory
Data Analyst II / Research Associate
Henry M Jackson Foundation for the Advancement of Military Medicine
Uniformed Services University / Walter Reed Medical Center
d.n.schreibergregory@gmail.com

Kaushal Chaudhary
Senior Biostatistician
Eli Lilly
kaushal2040@gmail.com