WUSS Paper 108-2022

Getting Started with DATA Step Hash Objects

Joshua M. Horstman, Nested Loop Consulting

ABSTRACT

The hash object provides a powerful and efficient way to store and retrieve data from memory within the context of a DATA step. This presentation will introduce the hash object, cover its basic syntax and usage, and walk through several examples that demonstrate how it can offer new and innovative solutions to complex coding problems. This presentation is intended for SAS® users who are already proficient with basic DATA step programming.

INTRODUCTION

The DATA Step Component Object interface is a SAS language feature unlike anything else in the DATA step. It brings object-oriented programming concepts found in many other programming languages into the environment of the DATA step. One of the most widely used component objects is the hash object, which is popular for its ability to store large amounts of data in memory and access the data very efficiently. This paper introduces the reader to the hash object and its companion, the hash iterator object. After a brief overview of its syntax and usage, several examples will be presented to highlight its advantages.

USING DATA STEP COMPONENT OBJECTS

As of SAS 9.4, there are five different types of component objects available. These include the hash and hash iterator objects, the Java object, and the logger and appender objects. This paper will focus exclusively on the hash and hash iterator objects.

The hash object stores a data table in memory. It provides random access to the contents of the data table based on a specified key variable or combination of variables. Because all the processing is done in memory, reading from and writing to a data table is fast and efficient. When used with large data sets, the performance benefits of the hash object can be quite substantial.

The hash iterator object works in conjunction with an existing hash object to allow for sequential access to the contents of the hash object. Data can be retrieved by stepping through one row at a time, either in forward or reverse order.

CREATING A COMPONENT OBJECT

Before a component object can be used, it must be first declared and then instantiated. Instantiation is the process by which an instance of a particular type of component object is created.

A component object is declared using the DECLARE statement (which can be abbreviated DCL). The declare statement must include both the type of component object being created as well as the name you wish to assign to it. Since this name becomes a variable on the Program Data Vector (PDV), it must comply with the normal rules for naming DATA step variables.

Once the hash object has been declared, it must be instantiated. This is done using the _NEW_ operator in an assignment statement. The following pair of statements demonstrates the declaration and instantiation of a hash object named MYHASH. Note the empty set of parentheses at the end of the assignment statement. These can be used to provide optional constructor arguments, which will be discussed in the next section.

```
declare hash myhash;
hashname = _new_ hash();
```

As a shorthand, the two statements above can be replaced with a single DECLARE statement that both declares and instantiates the hash object. Note that when using this syntax, the optional list of arguments is included at the end of the DECLARE statement.

```
declare hash myhash();
```

HASH OBJECT CONSTRUCTORS

When a component object is created, its attributes can be controlled through arguments known as constructors. The constructors appear in parentheses at the end of the statement which instantiates the object as discussed in the previous section.

The constructor arguments available are different for each type of component object. There are several different constructors that can be used when creating a hash object. The following are a few of the more useful constructors:

- **Dataset** – Specifies the name of an existing SAS data set that will be loaded into the hash object upon instantiation
- **Ordered** – Indicates the order in which data will be returned when the contents of the hash object are written out using the OUTPUT method or accessed sequentially using a hash iterator object. Valid values are 'ascending', 'descending', 'yes' (which is the same as 'ascending') or 'no' (which leaves the data unsorted). Each of these can be abbreviated by their initial letter.
- **Multidata** – Determines whether multiple data items are allowed for each key value (or combination of values). Valid values are 'yes' and 'no'. The latter indicates that key values must be unique. Each of these can be abbreviated by their initial letter.

The following statement shows how to create a hash object called MYHASH and use constructors to load a SAS data set and specify that values will be sorted in ascending order.

```
declare hash clin (dataset:'sashelp.cars', ordered:'Y');
```

HASH OBJECT METHODS

Component objects include built-in methods that define the operations the object can perform. Methods are conceptually similar to functions. They can accept arguments (but don't have to do so) and return a value. Since a DATA step could potentially use multiple component objects, it is necessary to specify the object being acted upon by utilizing the following notation.

```
MyObject.MethodName(arguments)
```

The hash object provides a few methods for refining the object definition and several others for interacting with the data stored therein. The following methods are those that refine the object definition. They are generally called after the DECLARE statement and prior to any other operations.

- **DefineKey** – Specifies a variable or list of variables that form the primary key for the hash object
- **DefineData** – Specifies the list of data set variables that are included in the hash object (or use ALL:"YES" to indicate that all variables on the PDV should be included)
- **DefineDone** – Closes the object definition. This method should be called only after the definition is complete and you are ready to start using the object.

The following code shows how these statements would be used to complete the definition of a hash object.

```
declare hash myhash();
  myhash.definekey('subject','visit');
  myhash.definedata('subject','visit','labdt');
  myhash.definedone();
```

Once a hash object has been fully defined and the definition has been closed, additional methods can be called as needed to carry out whatever processing is required. The following are some commonly used methods. Refer to *SAS® 9.4 Component Objects: Reference, Third Edition* (2016) for a complete list.

- **Add** – Adds the data currently in the PDV (Program Data Vector) to the object
- **Delete** – Deletes the hash object
- **Find** – Retrieves information from the hash object based on the values of the key variables currently found in the PDV.
- **Output** – Writes the contents of the hash object to a SAS data set
- **Replace** – Writes the data currently in the PDF to the object based on the values of the key variables, replacing whatever corresponding data is currently stored in the hash object

EXAMPLE #1: SORTING USING A HASH TABLE

Our first example involves using a hash table to sort a data set. The following DATA step reads the data set ADVRPT.DEMOG into a hash object, sorting it as the data is loaded into the hash object. It then writes the contents of the hash object out to a data set called CLINLIST.

Because we have not specified the MULTIDATA argument when constructing this hash object, multiple rows with the same key variable values are not allowed. The result of this sorting process is the same as what would be obtained using PROC SORT with the NODUPKEY option.

```
data _null_; ❶
  if 0 then set advrpt.demog(keep=clinnum subject lname fname dob); ❷
  declare hash clin (dataset:'advrpt.demog', ordered:'Y'); ❸
    clin.definekey ('clinnum','subject'); ❹
    clin.definedata('clinnum','subject','lname','fname','dob');
    clin.definedone();
  clin.output(dataset:'clinlist');  ❺
  stop;  ❻
run;
```

❶ This DATA step exists solely to allow the use of a hash object. The DATA step itself will not create an output data set.

❷ Because 0 is always false, this SET statement never executes. However, the compiler will still add the variables found in the ADVRPT.DEMOG data set to the Program Data Vector (PDV), which is the only reason this statement is included.

❸ The DATASET constructor is invoked to cause the entire ADVRPT.DEMOG data set to be loaded into the hash object CLIN when it is created.

❹ The data will automatically be sorted by the keys specified as it is loaded into the hash object.

❺ Once the hash object has been created and loaded, the OUTPUT method is called to write the already-sorted contents of the object out to the CLINLIST data set.

❻ The STOP statement closes the implied loop created by the SET statement. Since the SET statement never actually executes, it would result in an infinite loop if the STOP were omitted.

EXAMPLE #2: USING THE FIND METHOD

Our second example demonstrates the use of the FIND method to retrieve information from the hash object. This example involves two data sets related to a clinical trial. The ADVRPT.CONMED data set contains information about the concomitant medications taken by each subject during the trial while the ADVRPT.AE data set includes adverse events experienced by each subject.

The purpose of this example is to find all drugs in the CONMED data set that a subject started taking within 5 days prior to an adverse event. This is a form of many-to-many merge since each subject could potentially have multiple drug events and multiple adverse events that meet the specified criteria. While a traditional DATA step merge is not well suited for performing a many-to-many merge, this example shows how the hash object can be used to complete the task.

In the following DATA step, the entire contents of the ADVRPT.CONMED data set are loaded into a hash object in the first DO UNTIL loop. In the second DO UNTIL loop, the adverse event records from ADVRPT.AE are read one at a time. For each AE record, the date is compared with the medication date from each of the CONMED records for the same subject that were stored in the hash object. Only those combinations of event date and medication date that meet the search criteria are written to the output data set.

```
data drugEvents(keep=subject medstdt drug aestdt aedesc sev); ❶
  declare hash meds(ordered:'Y'); ❷
    meds.definekey ('subject','counter');
    meds.definedata('subject','counter','medstdt','drug');
    meds.definedone();
  do until(allmed);
    set advrpt.conmed(keep=subject medstdt drug) end=allmed; ❸
    by subject; ❹
    if first.subject then counter=0;
    counter+1; ❺
    rc=meds.add(); ❻
  end;
  do until(allae);
    set advrpt.ae(keep=subject aedesc aestdt sev) end=allae; ❼
    counter=1;
    rc=meds.find(); ❽
    do while(rc=0);
      if (0 le aestdt - medstdt lt 5) then output drugevents; ❾
      counter+1;
      rc=meds.find(); ❿
    end;
  end;
  stop;
run;
```

❶ This DATA step creates an output data set called DRUGEVENTS. It will include one row for each combination of a concomitant medication and an adverse event that meet the search criteria.

❷ The DECLARE statement and subsequent lines define a hash object called MEDS that will be used to store the concomitant medication information read in from the CONMED data set. The key variables for this hash object are SUBJECT and a variable we will create called COUNTER.

❸ The SET statement reads records from the CONMED data set one at a time. When it has read the last record, the variable ALLMED will be true. Until then, it will be false and the DO UNTIL loop will continue iterating, once for each record in the data set.

❹ Including SUBJECT on a BY statement gives us the automatic variables FIRST.SUBJECT and LAST.SUBJECT. Since a BY statement is being used, the incoming data set must be sorted.

❺ The purpose of the variable COUNTER is to number each CONMED record sequentially within each subject. The first record for each subject will have COUNTER equal to 1.

❻ After each record is read from CONMED and a value is assigned to COUNTER, the record is added to the MEDS hash object. When this DO UNTIL loop finally terminates, the entire data set will have been read in and added to the hash object, including the COUNTER.

❼ A second SET statement is used to read adverse event records from the AE data set. Like the first SET statement, this one is inside a DO UNTIL loop that will terminate only after the last record has been read.

❽ The FIND method is called on the MEDS object. It looks for a record in the hash object having key variable values that match those currently in the Program Data Vector (PDV). At this point, the value of SUBJECT in the PDV is the value that was just read in from the first record of the AE data set and the value of COUNTER is 1. Those values are used to perform the lookup. If a corresponding row exists in the hash table, the values of the other variables, DRUG and MEDSTDT, are copied into the PDV.

❾ Once a record has been read from AE and a medication record for the same subject has been retrieved from the MEDS hash object, we compare the dates to see if the search criteria are met. If so, a record is written to the output data set DRUGEVENTS.

❿ The counter is incremented, and the FIND method is called to retrieve the next medication record for the current subject from the hash object. Once all the medications for the current subject have been read, the FIND will fail because there will not exist a row in the hash object for the current SUBJECT and COUNTER. When the FIND fails, a non-zero return code is written to the variable RC and the DO WHILE loop terminates. At that point, all medication dates for the current subject have been compared to the adverse event data for the AE record that was read. The outer DO UNTIL loop iterates, another record is read from AE (possibly for the same subject, possibly for another), COUNTER is reset to 1, and the process repeats so that the dates on each medication record can be compared to the dates on each adverse event for each corresponding subject.

EXAMPLE #3: USING THE HASH ITERATOR OBJECT

The third example is an alternative solution to the problem described in the second example. The hash iterator object is used to step sequentially through the contents of the hash object rather than the FIND method. This approach is not necessarily ideal for the task at hand, but it highlights the differences between using the hash iterator object together with the hash object and using the hash object alone.

```
data drugEvents(keep=subject medstdt drug aestdt aedesc sev);
  declare hash meds(ordered:'Y');
  declare hiter medsiter('meds'); ❶
  meds.definekey ('subj','counter'); ❷
  meds.definedata('subj','counter','medstdt','drug');
  meds.definedone();
  do until(allmed); ❸
    set advrpt.conmed(keep=subject medstdt drug) end=allmed;
    by subject;
    if first.subject then do;
      counter=0;
      subj=subject; ❹
    end;
    counter+1;
    rc=meds.add();
  end;
  do until(allae); ❺
    set advrpt.ae(keep=subject aedesc aestdt sev) end=allae;
    rc = medsiter.first();❻
    do until(rc);
      if subj=subject & 0<=aestdt-medstdt<5 then output drugevents; ❼
      if subj gt subject then leave; ❾
      rc=medsiter.next();❽
    end;
  end;
  stop;
run;
```

❶ In addition to declaring a hash object called MEDS, we also declare a hiter (hash iterator) object called MEDSITER. The name of the hash object is passed in as a constructor argument to the hash iterator object to create the linkage between the two. The methods of the MEDSITER object can now be used to move sequentially through the contents of the MEDS hash object.

❷ Notice that our key variable is called SUBJ, not SUBJECT as was the case in the previous example. The reason for this is described in point #7 below.

❸ The first DO UNTIL loop is similar to the one from the previous example and serves the same purpose, which is to load the contents of the CONMED data set into the MEDS hash object.

❹ The value of the variable SUBJECT in the CONMED data set is copied to a new variable called SUBJ which is part of the hash object.

❺ As in the previous example, the second DO UNTIL loop contains a SET statement that reads records from the AE data set one at a time and continues iterating until the entire data set has been read.

❻ Instead of using the FIND method to retrieve a specific record from the hash object as was done in the previous example, we use the FIRST method of the hash iterator object to retrieve the very first record from the hash object.

❼ Since we are reading sequentially though the entire hash object, we must check that the current row in the hash object pertains to the same subject as the record most recently read from AE. The subject number retrieved from the hash object is stored in SUBJ, while the subject number read from the AE data set is in SUBJECT. We also still need to check that the dates meet the search criteria before writing a record to the output data set, DRUGEVENTS.

❽ The NEXT method of the hash iterator object will return the next sequential row from the hash object (which may or may not pertain to the same subject).

❾ Since the hash object is sorted by subject number, we can terminate the DO UNTIL loop early to improve efficiency once we reach a point in the hash object where the subject numbers are now greater than the subject number of the current AE record. No matches will be found beyond that point.

EXAMPLE #4: MERGING USING A HASH TABLE

While there are several ways to merge data sets in SAS, hash objects provide an alternative that may be more efficient than other methods. A particular advantage to this approach is that neither data set need be sorted in advance. For large data sets, this can result in a considerable performance improvement.

The general process is to load one data set into a hash object, read the other data set one record at a time using the SET statement, and use the key values read using the SET statement to perform a lookup on the hash object. The following code merges the ADVRPT.DEMOG data set with the ADVRPT.CLINICNAMES data set using the common variable CLINNUM as the key.

```
data hashnames(keep=subject clinnum clinname lname fname);
  if 0 then set advrpt.clinicnames; ❶
  declare hash lookup(dataset:'advrpt.clinicnames');  ❷
    lookup.defineKey('clinnum'); ❸
    lookup.defineData('clinname');
    lookup.defineDone();
  do until(done); ❹
    set advrpt.demog(keep=subject clinnum lname fname) end=done;
    if lookup.find() = 0 then output hashnames; ❺
  end;
  stop;  ❻
run;
```

❶ Because 0 is always false, this SET statement never executes. This statement is included to force the compiler to add the variables found in the ADVRPT.CLINICNAMES data set to the Program Data Vector (PDV).

❷ The hash object LOOKUP is declared and the DATASET constructor argument is used to load the ADVRPT.CLINICNAMES data set into the hash object upon creation.

❸ The variable CLINNUM will serve as the primary key for the hash object LOOKUP.

❹ The DO UNTIL loop will iterate until the SET statement within the loop has read all records from the ADVRPT.DEMOG data set.

❺ For each record read from DEMOG using the SET statement, the FIND method is used to retrieve the row from the hash object that has the same value of CLINNUM.

❻ The STOP statement prevents the DATA step from entering an infinite loop. This is necessary because the first SET statement will never execute and therefore never finish reading the CLINICNAMES data set.

PERFORMANCE COMPARISON SIMULATION

As has been discussed, one of the primary motivations for using the hash object is its efficiency. Particularly when very large data sets are involved, certain tasks can be completed more quickly using hash objects than more conventional methods.

To illustrate this point, a simulation was run to compare the performance of three different approaches to merging two data sets. The three methods used are the traditional DATA step merge (which includes pre-sorting each data set), a PROC SQL join, and a hash table lookup as described previously in Example #4.

Two simulated data sets were created. Both data sets were unsorted and unindexed. The first data set contained simulated data on 500,000 insurance policies. It included a randomly-generated 10-character alphanumeric MEMBERID and 25 numeric variables containing random values. The second data set contained simulated data representing 10,000,000 insurance claims. Each record contained a MEMBERID randomly selected from the first data set as well as 25 numeric variables containing random values.

The simulation was performed using SAS 9.4 running on a Windows 10 desktop PC with 16GB memory. The following table shows the results of the simulation. Both the total CPU time and the memory usage were recorded.

Method	Total CPU Time	Memory Usage
Data Step Merge (with sorts)	10.05 sec	1.079 GB
PROC SQL Join	11.14 sec	1.070 GB
Hash Table Lookup	8.54 sec	0.159 GB

In this simulated example, using a hash object to perform the merging saved a considerable amount of both CPU time and memory compared with the other methods. Of course, actual results will vary depending on the size of the data sets being merged and the system resources available. The hash object may not be the best choice in all situations. When selecting an approach for a specific task, it is advisable to perform some benchmarking to discover which method may perform best for that situation.

CONCLUSION

The hash object offers a powerful and efficient way to store and retrieve data from memory within the context of the DATA step. While the code can be more complex than conventional approaches, the hash object offers the programmer complete control of the process of reading and writing data. This can make it possible to accomplish complex tasks in a single DATA step. The hash object can also offer efficiency improvements, both in terms of processing time and memory requirements. The hash object deserves a place in any SAS programmer's toolbox.

REFERENCES

SAS Institute Inc. 2016. *SAS® 9.4 Component Objects: Reference, Third Edition.* Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

Portions of the material for this paper were adapted from a training course "Innovative Tips and Techniques: Doing More in the DATA Step", originally developed by Art Carpenter and now presented with Art's permission from time to time by Josh Horstman. That course, in turn, is based on material from Art Carpenter's book, "Carpenter's Guide to Innovative SAS Techniques."

CONTACT INFORMATION

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

Joshua M. Horstman
Nested Loop Consulting
317-721-1009
josh@nestedloopconsulting.com