Paper CT06

Blink and You Will Miss It !

Daniel Lindenbaum, Phastar, London, United Kingdom

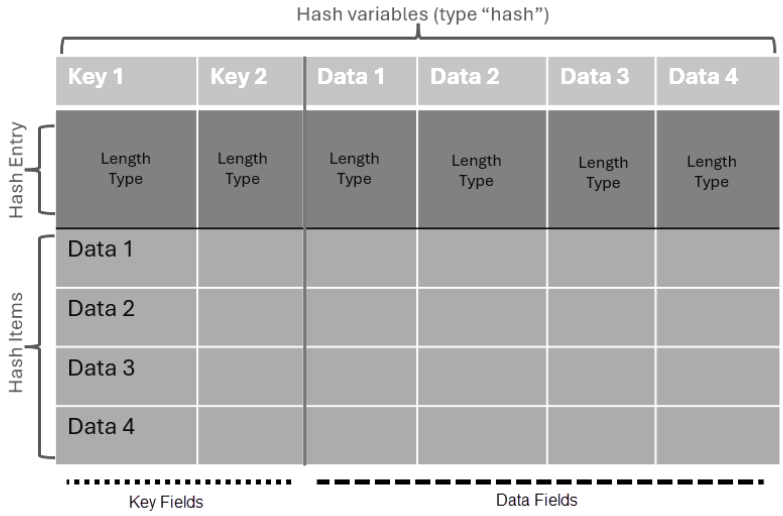
ABSTRACT

Hash table programming opens doors to under-explored and powerful areas of SAS® but many programmers avoid it due to quirky syntax and perceived difficulties. For those brave enough to roll up their sleeves and accept the challenge, there are great rewards to be uncovered. Even at an elementary level, hash objects offer novel and concise solutions to many daily programming tasks. Data operations are carried out in memory and minimize disk I/O. Memory operates in the nanosecond region as opposed to the milliseconds required for disk access – one thousand times faster. Sort, Merge (without pre-sorting), Lookup and Modify datasets serve as an introduction to the possibilities that are available when adopting this efficient element of the SAS language.

INTRODUCTION

Programmers face a wide range of challenges regarding the processing and summary of clinical data. Data is often complex, and it can be necessary to access multiple data sources, along with subsets, re-sorts and joins: to produce a specified presentation of data. The simplification and reduction of code should always be on the agenda for overall efficiency gains, not only in terms of streamline processing but also time saved as a result of reduced QC duration, easier re-work and the updating inherited programs. Although hash table programming can appear daunting to the uninitiated (and harnessing the full capabilities of hashing can indeed become complex), if time is taken to grasp the fundamentals, then a door is opened to an additional tool for the programmer's repertoire. Even an understanding of the basics offers a major expansion of code choice and opportunity. The objective of this paper is to open the door to the hash object, with a focus on elementary hash programming.

ANATOMY OF A HASH OBJECT



The *hash entry* is the descriptor portion of the hash object; containing information about the individual fields.
Any data that is contained within the hash object are stored as *hash items*. All variables belonging to the hash object are of type "*hash*" as opposed to the more typical character/numeric fields that datastep programmers will be used to. The key fields can be scalar only; the permitted content being character or numeric. The data fields can be scalar [1] or non-scalar [2].

[1] Scalar data is of either character or numeric type. There are no matrix or vector data structures such as arrays.

[2] Non-scalar data is of either character, numeric, hash (a memory array) or hash iterator type data and can contain structures with multiple data points.

CONCEPTS, BEHAVIOURS & POTENTIAL PITFALLS

SAS ® hash tables are programming objects that reside solely in computer memory. This makes them rapid vehicles for data access and operations. Data is read just once from disk and then loaded into memory. It is important to bear in mind there is a limitation on the memory available to SAS and there must be adequate resource to fit the active hash table. Consideration for other server users should also be in mind. The code that controls these objects is concise and easy to follow once the basic techniques are mastered but there is an associated learning curve for the newcomer to this side of SAS. The main tools for data reads, writes, comparisons, lookups and other manoevers are carried out by syntax items known as "methods". There are many methods available and this paper will only skim the surface. The aim being to equip the fledgling hash programmer for performing basic data related tasks and preparing the ground for taking on the more sophisticated capabilities of the system. SAS objects only exist for the duration of the datastep to which they belong and in order to to pass data from step to step via the hash route will require the use of intermediary datasets. Methods can generate return codes, if requested, the values of these being a zero for a successful operation and greater than zero for a negative outcome. These returned values can be used as a means to further control and drive program code. The concept of key variables and data variables will be introduced; key variables can be repeated in the data portion of the hash object, making the values more available for data operations and reporting. It should be noted that the hash object requires a mirror variable of the same name in the associated dataset, to operate without error. The hash object is unable to automatically pull variable descriptor information from the associated dataset counterpart; code must be employed that forces the two data structures to communicate. The SAS compiler needs to 'see' the attribute information.

METHODS

Method	Function
define_key	Define the names of the key fields to be used in the hash object
define_data	Define the names of the data fields to be used in the hash object
define_done	Complete the hash object definition process
add	Insert data values to the hash object data fields
check	Evaluate whether a key field value exists in the hash object
find	As per the check method, also returns the corresponding hash object data values to the PDV
output	Send the hash object contents to a dataset
clear	Delete all the values from a hash object, the object itself is not deleted
first	Retrieve the data from the first record in the hash object and place the values in the PDV
last	Retrieve the data from the last record in the hash object and place the values in the PDV
next	Retrieve data from the subsequent record (if it exists) in the hash object and send the values to the PDV
prev	Retrieve the data from the previous record (if it exists) in the hash object and place the values in the PDV

SKELETON SYNTAX

The following code will define and create an instance of the hash object. It will not load any data values.

```
data _null_;                                              (1)
  length key1 8 key2 $10 data1 data2 $25;                 (2)
  call missing(key1, key2, data1, data2);                 (3)
  declare hash hob();                                     (4)
    hob.definekey('key1', 'key2');                        (5)
    hob.definedata('key1', 'key2', 'data1', 'data2');     (6)
  hob.definedone();                                       (7)
run;
```

(1) No output dataset required.
(2) Notify the PDV of variable attributes (broadcast the attributes to the compiler)
(3) Initialize PDV values to missing, we need "mirror" variables in the PDV and the hash object.
(4) Declare the named hash object named "hob" and create an instance of the hash object.
(5) Supply the names of key fields for the hash object
(6) Name the data fields for the hash object.
(7) End the definition phase for the hash object.

THE ADD METHOD

This is one of the routes for the addition of data values to the hash object, one record at a time from the PDV. The PDV variable will be used as source for the 'same named' variable in the hash object.

```
data _null_;
  length key1 8 key2 $10 data1 data2 $25;
  declare hash hob();
    hob.definekey('key1', 'key2');
    hob.definedata('key1', 'key2', 'data1', 'data2');
  hob.definedone();
  key1=value1;      (1)
  key2='value2';
  data1='value3';
  data2='value4';
  rc=hob.add();     (2)
run;
```

(1) Assign values to the PDV variables.
(2) Add the current PDV variable values to the named hash object and generate a return code. The ADD method takes no arguments within the parentheses.

The value of the PDV variable key1 will be added to the hash object variable key1, the PDV value for key2 will be added to the hash object variable key2, and so on. The return code (rc) will take the value of zero if the *add()* operation is successful, if the operation fails then a return code greater than zero will be observed.

CHECK & FIND METHODS

Both of these methods are used to identify the occurrence of a key variable value in a hash object. The value will be taken from the PDV variable with an equivalent name and the hash object trees will be searched for a match. CHECK and FIND have differing actions on identification of a key match: CHECK will simply give a return code of zero for a match and greater than zero for a non-match whereas FIND will give a return code of zero plus a fetch of the values from the hash object for the key associated data fields.

```
%macro chkfnd(method=);
  data _null_;
    declare hash hob();
      hob.definekey('key1');
      hob.definedata('key1','data1');
    hob.definedone();
    key1=1000;                          (1)
    data1='Alpha';
    rc1=hob.add();                      (2)
    key1=2000;                          (1)
    data1='Beta';
    rc1=hob.add();                      (2)
    do z=1000, 2000, 3000;              (3)
      key1=z;
      call missing(data1);              (4)
      rc2=hob.&method.();               (5)
      put _all_;
    end;
  run;
%mend chkfnd;

%chkfnd(method=check)                   (a)
%chkfnd(method=find)                    (b)
```

```
(1)  Assign PDV values to key1 and data1 fields
(2)  Add a row from the PDV values to the hash object
(3)  Loop through values to assign to the key field in the PDV
(4)  Initialise the PDV value for the data field to missing to avoid retaining issues
(5a) check the key values held in the hash object for the key value currently held
     in the PDV
(5b) Check the key values held in the hash object for the key value currently held
     in the PDV and return the associated hash object data variable values
```

EXCHANGING DATA WITH A FILE

Data can be imported to a hash object directly from a dataset; hash object content can also be written to an output dataset.

```
data sourcea;
 input cshort:$1. nshort:8. cfull:$10.;
  cards;
  a  1 apple
  p 16 pear
  n 15 nectarine
  b  2 banana
  ;
run;

data _null_;
  if 0=1 then set sourcea;                    (1)
  declare hash hob(dataset:'sourcea');        (2)
    hob.definekey('cshort','nshort');         (3)
    hob.definedata('cshort','nshort','cfull');
  hob.definedone();
  put _all_;
  hob.output(dataset: 'sourcea2');            (4)
  stop;
run;
```

```
(1) Read the descriptor information from the source dataset to update the PDV
    with variable header information that will be used to match with the hash
    object variables.
(2) Define the hash object along with the named dataset to be used as source to
    populate the hash object variables.
(3) Define the keys and data variables for the hash object; these variables must
    also be available in the source dataset.
(4) Define an output dataset using the output method. The contents of the hash
    object will be delivered to the named dataset.
```

SORTING & DUPLICATE KEY PROCESSING

Hash objects can be sorted according to the specified key fields. The order can be either ascending (“a” or “y”), descending (d). The tag that is used for ordering is “ordered:”. The “multidata:” tag facilitates the inclusion of records with duplicate keys. The tag that is used for more directed handling of duplicate keys is “duplicate:”.

```
data sourceb;
  input idvar:8. Name:$10. Age:8. Country:$2.;
  cards;
  163  Sarah 32  UK
  142  Bob   24  SP
   26  Kate  20  FR
  161  Chris 40  IT
   26  Kath  36  NO
  161  Colin 44  SW
    ;
run;
```

```
%macro dsort(keepdup=, sequence=, outds=);
  data _null_;
    if 0 then set sourceb;
    dcl hash hob(dataset:"dsource", ordered:"&sequence", multidata:"&keepdup"); (1)
        hob.definekey("idvar");
        hob.definedata("idvar","name","age","country");
      hob.definedone();
    hob.output(dataset:"&outds");                                              (2)
    stop;
  run;
%mend dsort;

%dsort(keepdup=y, sequence=a, outds=dsource_ya)       (3)
%dsort(keepdup=y, sequence=d, outds=dsource_yd)       (4)
%dsort(keepdup=n, sequence=a, outds=dsource_na)       (5)
%dsort(keepdup=n, sequence=d, outds=dsource_nd)       (6)

(1) Declare the hash object along with the data source, desired sort sequence and the
    tag for permitting/discarding duplicate keys within the object.
(2) Define the output data destination.
(3) Request an ascending sort, keeping duplicate keys.
(4) Request a descending sort, keeping duplicate keys.
(5) Request an ascending sort, removing duplicate keys.
(6) Request a descending sort, removing duplicate keys.
```

If duplicates are to be dropped then the first encountered observation within the key value will be kept, using the "multidata" tag value of "n".

```
%macro ddup(keepdup=, sequence=, dup=, outds=);
  data _null_;
    if 0 then set sourceb;
    dcl hash hob(dataset:"dsource", ordered:"&sequence", duplicate: "&dup"); (1)
        hob.definekey("idvar");
        hob.definedata("idvar","name","age","country");
      hob.definedone();
    hob.output(dataset:"&outds");
    stop;
  run;
%mend ddup;

%ddup(sequence=a, dup=add, outds=ds_add)              (2)
%ddup(sequence=a, dup=replace, outds=ds_replace)      (3)
%ddup(sequence=a, dup=error, outds=ds_error)          (4)

(1) Declare the hash object along with the data source, desired sort sequence and
    the method for handling duplicates within the object.
(2) Perform an ascending sort, keeping the first key record per group and discarding
duplicates.
(3) Perform an ascending sort, keeping the last key record per group and discarding
duplicates.
(4) Perform an ascending sort, creating an ERROR condition if a duplicate key
    is encountered.
```

When using the "duplicate" tag, there is more control. The tag value "add' will keep the first observation in the key set, the tag value "replace" will keep the last observation of the key set and the tag value "error" will trigger an error if a duplicate key is encountered.

LOOKUP

A data value contained in a dataset can be looked-up in another larger dataset, returning records if a key match is found. For the sake of server memory conservation, it makes sense place the smallest of the two datasets into the hash object and to use this for performing a look-up in the larger one. In a test lookup with a master dataset of 2.5 million data points, retrieving a single record from the unsorted data using hash objects, took ~4% of the time taken, when compared with an equivalent merge based retrieval.

```
data sourcec;
  input trtcode:$1. Trtment:$20. subjid:8. Sex:$1. age country:$3.;
  cards;
  a  active 1001 M 42 FRA        (1)
  a  active 2007 F 38 UK
  p  active 8012 M 27 SPA
  a  active 1002 M 45 GMY
  p  active 6101 F 55 ITA
  a  active 4121 F 48 POR
  ;
run;

data lkup;                       (2)
  subjid=8012;
run;

data pat8012;
  if 0 then set sourcec;
  if _n_=1 then do;
    dcl hash hob(dataset:"lkup"); (3)
      hob.definekey('subjid');    (4)
    hob.definedone();
    end;
    set pats;
    if (hob.check()=0) then do;   (5)
      msg='dataset key matches lookup hash key';
      output;                     (6)
    end;
 run;
```

Total rows: 1 Total columns: 7

	trtcode	Trtment	subjid	Sex	age	country	msg
1	p	active	8012	M	27	SPA	dataset key matches lookup hash key

(1) Populate the main source dataset.
(2) Populate the lookup dataset.
(3) Read the smaller of the two datasets into the hash object (server memory conservation).
(4) Define the hash object key that will be used for the look-up match.
(5) If the return code from the check is zero then a match has been found.
(6) Output the record if (5) is a success.

HASH ITERATOR OBJECT

An object of non-scalar type that can be defined within a hash object. It enables the records of the hash object to be processed one at a time, from 'top to bottom' or 'bottom to top. The 'next' or 'previous' record can be selected from any given point in the table.

```
data sourced;
  input trtcode:$1. testcd:$5. test:$20. subjid:8. sex:$1. vnum:8. result:8.;
  cards;
  a  bpsys Systolic 1001 M 4 130
  b  bpdia Diastolic 1006 F 3 82
  a  hrate Heart 2008 M 2 76
  a  bpsys Systolic 3012 F 3 128
  b  bpsys Diastolic 1006 F 4 78
  b  hrate Heart 7103 M 2 68
  a  bpsys Systolic 3009 F 1 142
  ;
run;

data _null_;
  if 0 then set sourced;
  declare hash hoba(dataset:'vitals', ordered:'a');
    declare hiter hita('hoba');
    hoba.definekey('subjid', 'vnum');
    hoba.definedata('subjid', 'vnum', 'trtcode', 'testcd', 'test','sex','result');
  hoba.definedone();
  call missing(of _all_);
    hita.first();
    put _all_;
    hita.next();
    put _all_;
    hita.last();
    put _all_;
    hita.prev();
    put _all_;
  stop;
run;
```

```
(1) Define the hash iterator and the hash table to which it is to be assigned.
(2) Set all PDV values to missing.
(3) Move the pointer to the first record of the associated hash object.
(4) Move the pointer to the subsequent record of the hash object.
(5) Move the pointer to the last record of the hash object.
(6) Move the pointer to the previous record of the hash object.
(7) Allow the compiler to see the descriptor information from the source dataset
    but prevent data records being read in by the datastep.
```

Ordered (ordered:'a') record sequence:

```
  a  bpsys Systolic  1001 M 4 130
  b  bpdia Diastolic 1006 F 3 82
  b  bpsys Diastolic 1006 F 4 78
  a  hrate Heart     2008 M 2 76
  a  bpsys Systolic  3009 F 1 142
  a  bpsys Systolic  3012 F 3 128
  b  hrate Heart     7103 M 2 68
```

Retrieved records:

```
NOTE: There were 7 observations read from the data set WORK.VITALS.
trtcode=a testcd=bps test=Systolic subjid=1001 sex=M vnum=4 result=130 _ERROR_=0 _N_=1
trtcode=b testcd=bpd test=Diastolic subjid=1006 sex=F vnum=3 result=82 _ERROR_=0 _N_=1
trtcode=b testcd=hra test=Heart subjid=7103 sex=M vnum=2 result=68 _ERROR_=0 _N_=1
trtcode=a testcd=bps test=Systolic subjid=3012 sex=F vnum=3 result=128 _ERROR_=0 _N_=1
```

OBJECT MODIFICATION

A dataset can be modified by using a hash object as the vehicle for updates. The new values are loaded into a hash object and the master dataset is checked for key value matches – using the FIND method. If a key match is located then the new values are loaded into the PDV from the hash object and the "coalesce" datastep function is used to modify the original PDV value with data from the hash object. The modified data is then output via the standard datastep execution.

```
data sourcee;                                    (1)
  set sashelp.class(keep=name age);
run;

data mods;                                       (2)
  infile datalines truncover dlm=',' dsd;
  input name:$8. age2:8.;
  datalines;
  Carol,18
  John,17
  ;
run;

data modified;
  if _n_=1 then do;
    if 0 then set mods;
    dcl hash hob(dataset:'mods');                (3)
      hob.definekey('name');
      hob.definedata(all:'y');
    hob.definedone();
  end;
  modify modified;                               (4)
  if hob.find()=0 then do;
    age=coalesce(age2, age);                     (5)
  end;
run;

(1) Create the source dataset.
(2) Create the modified value dataset
(3) Read the modified data into a hash object.
(4) Specify the data source for modification.
(5) Update the source dataset with the values from the modification data, by key.
```

MERGE/JOIN

A merge can be achieved by loading the satellite (rightmost) dataset into the hash object and using the FIND method to locate matched keys in the master dataset (leftmost). Data is added to the PDV from the satellite data and the record is output if the keys match. If there is not a match then any non-key fields from the satellite data are assigned to missing values, in order to avoid retained value issues.

```
/*Master data*/
data sourcef;
  set sashelp.class(keep=name age);
run;

/*Satellite data*/
data mods;
  infile datalines truncover dlm=',' dsd;
  input name:$8. age2:8.;
  datalines;
  Carol,18
  John,17
  ;
run;

%macro joins(master=, sat=, datvar=, missvar=);
  data left_&master(drop=rc);
    if 0 then set &master;
    declare hash hob();
```

```
      hob.definekey('name');
      hob.definedata('name', &datvar);
    hob.definedone();
    do until(eof1);
      set &sat end=eof1;
      rc=hob.add();               (1)
    end;
    do until(eof2);
      set &master end=eof2;
      rc=hob.find();              (2)
      if rc=0 then output;
      else do;
        call missing(&missvar);   (3)
        output;
      end;
    end;
    stop;
  run;
%mend joins;

%joins( master=sourcef, sat=mods, datvar=%str('age2'), missvar=%str(age2) );

(1) Add the data from the satellite dataset to the hash object.
(2) Read in the data from the master dataset and pull the values from the hash
    object (age2 variable) if the PDV key value exists in the hash object.
(2) If the matching key value is not found in the hash object then set the value of
    age2 to missing, to prevent retained values from the satellite dataset.
```

CONCLUSION

Hash objects are a most flexible and efficient means of using the SAS language with an additional dimension, no pun intended. Whilst being a little alien in nature to those from a traditional data step and procedure driven background, it represents a potential expansion in coding repertoire and technique, that is there for the taking. If the time is taken to learn and practice with the hash object syntax, then it will become second nature, to have it as a “go-to” strategy. Programs can become simpler, code more compact and data operations more flexible. Take the plunge and imerse yourself in this topic for a few weeks and you will master the basic techniques. The full extent of what is possible via hash object programming goes way beyond the scope of this paper. The intention is to pique interest and provide an elementary introduction.

ACKNOWLEDGMENTS

I am most grateful to Paul Dorfman for his expert insights into the world of hash object programming and for answering all my questions. Also to both Paul Dorfman and Don Henderson for their book, giving such a detailed and accessible account of the hash object and it’s uses.

RECOMMENDED READING

Data Management Solutions Using SAS Hash Table Operations: A Business Intelligence Case Study, Paul Dorfman & Don Henderson, SAS Institute, 2018.

CONTACT INFORMATION

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

Author Name: Daniel Lindenbaum
Company: Phastar
Address: UK HQ, 2D Bollo Lane, Chiswick, London, W4 5LE.
Work Phone: +44 (0) 743 555 4654
Email: daniel.lindenbaum@phastar.com
Website: https://phastar.com/