Paper HOW-195

Fundamentals of the The SAS® Hash Object

Paul M. Dorfman, Independent Consultant, Jacksonville, FL

ABSTRACT

Starting with the basics and progressing to some less-used features, this workshop is designed to show how the SAS hash object really works. The main emphasis will be not on amassing as many pieces of template code as possible, but rather on the fundamental things a hash object programmer must understand in order to use it in creative ways. The aim is not so much about tasting already cooked hash dishes (though there will be chances to do that, too), but about cooking properly based on the fundamental properties of the ingredients and their interactions. Cuisine is much more than just following a recipe, and the same is true for programming with the hash object! In particular, we'll learn what the DATA step compiler sees when it encounters hash object references and what it must have seen - and done - to make the hash object work when its run-time turn comes. We'll intentionally goof to learn from errors reported in the SAS log. We'll see how the variables stored in the hash object talk to their host counterparts in the Program Data Vector, and which hash methods make them affect each other and how, including those related to the hash iterator. While focusing on these underlying works, we'll learn about other utile things that together ought to form a solid basis for making the hash object one's valuable SAS programming tool.

INTRODUCTION

The SAS hash object, as its name implies, is based on a certain variety of *hashing*. Hashing is a group of table search algorithms based on the concept of *direct addressing*. They weren't new in SAS when the hash object was introduced: A number of direct-addressing searching techniques had been already successfully implemented using the DATA step language and shown to be much more efficient than theretofore existing look-up methods. For example, a set of hand-coded direct-addressing routines based on SAS arrays, replete with a rather painful delving into their guts, was presented in [1, 2]. However, for all their performance advantages, such custom-written routines have a number of drawbacks. Chiefly, their SAS language implementation is rather advanced, and adjusting them for one's particular need may be difficult for intermediate-level users.

This and other shortcomings were removed with the introduction of the SAS hash object in Version 9.0. Since the underlying code is *canned* internally (akin to a SAS procedure, for example), learning how to use it is relatively simple after certain syntactic basics have been absorbed and the fundamentals of the manner in which the hash object interacts with the DATA step variables are understood. The latter is actually the tallest hurdle on the hash object learning curve because the way it operates represents a rather sharp departure from the pre-hash DATA step programming ideology. And yet its novel nature is what makes it not merely a fast look-up table but a flexible and powerful dynamic programming tool - capable of achieving aims difficult or impossible before its advent.

PROPAEDEUTICS

In a nutshell, the SAS hash object is a *hash table* supplied with tools to control and manipulate it in the DATA step during its execution (run) time. In particular:

- The hash table can be created, loaded, emptied, and deleted.
- The table is *temporary*: Once the DATA step has stopped execution, it ceases to exist. Thus, it cannot be reused in any subsequent step. However, its content can be saved in a SAS data file or external data base.
- The table resides completely in memory (RAM, main storage, etc., depending on the OS semantics). This is *one* of the factors making its operations fast, but it's also what limits the amount of data it can contain.
- Just like any table-like object, it contains rows (*hash entries)* and columns (*hash variables)*.
- Each entry *must have at least one key column and one data column*. The key variables make up the *key portion* of the table, and the data variables - the *data portion*.
- Because the table's look-up mechanism is based on hashing, it supports such standard key-search-based table operations as ***search***, ***insert***, ***update***, ***retrieve***, and ***delete*** in *constant*, or *O(1)* time. The latter means that their

execution time does not depend on the number of entries in the table.

- It supports the ***enumeration*** operation in *O(N)* time. It means that the time required to list all the entries is directly proportional to the number of entries, *N*.
- It supports, all on its own, ***input/output (I/O)*** operations. That is, the hash object can read from and write to a SAS data file (or any other data base structure properly linked via a SAS libref). These operations are *independent* from both the file names listed in the DATA statement and from the DATA step's own I/O statements (such as SET, MERGE, OUTPUT, etc.).

The hash object is implemented as a structure *separate from the DATA step proper*. Therefore, a mechanism must exist for the DATA step (a) to recognize the commands given to the hash object from inside it as syntactically valid and (b) to communicate the content of the hash table with the DATA step variables. This mechanism is implemented via certain add-ons to the DATA step compiler (to check command syntax) and via the so-called *Data Step Component Interface (DSCI)*.

You can picture the hash object as a black-box device you can manipulate from inside the DATA step to ask it for high-performance data storage, update, retrieval, and I/O services using the object's *methods* and *operators*. You can also ask for other information (e.g. the number of entries) using its *attributes*. The DSCI will fulfill these requests for you. For example, you can:

- Give the hash object a key and tell it to use to ***add*** it to the table along with associated data. This is equivalent to the ***insert*** operation.
- Locate an entry with a given key and ***replace*** the values in its data portion with the values of the corresponding PDV variables. This is equivalent to the ***update*** operation. If the key is not in the table, ***replace*** is equivalent to ***add***.
- Merely ***check*** whether a given key is in the table. This is equivalent to the ***search*** operation.
- ***Find*** an entry with a given key and overwrite the PDV variables with the corresponding values from the data portion of the entry. This is equivalent to the ***retrieve*** operation.
- ***Remove*** an entry with a given key from the table. This is equivalent to the ***delete*** operation.
- Tell the object to ***dataset*** (i.e. *read*) a SAS data file sequentially into the hash table. This is equivalent to the ***input*** service.
- ***Output*** (i.e. *write*) the table content into a SAS data file. If it exists, it will be overwritten. If it does not, it will be (a) created with a name of your choice, (b) written to, and (c) closed - all during the DATA step execution time and without the need to list it in the DATA statement beforehand. Hence, it can be done even if the DATA statement list is _NULL_.
- Make the hash object ***accumulate*** aggregate statistics for each key using the combination of the find and replace operations.
- Make the table entries ***ordered*** according to their keys
- ***Scan*** the hash table sequentially from the first physical entry up, from the last entry down, and also in both directions starting with a given key if it's in the table. This is done using an auxiliary object associated with the table called the ***hash iterator***. Listing any entry in such a manner is tantamount to an act of data retrieval since its data portion overwrites the values of the corresponding variables in the PDV. Obviously, using a hash iterator is equivalent to the ***enumeration*** operation.
- Do all of the above with a hash object *containing other hash objects* rather than scalar data.

“High-performance” mentioned above means that its operations are performed rapidly compared to other data access and retrieval methods available in SAS. It owes its speed to three factors:

- All hash object operations (except external I/O) are performed purely in memory.
- The direct addressing nature of the hybrid hashing algorithm makes the key-search operations much faster than the algorithms based on key-comparison structures, such a single binary tree.
- Hash search behavior is superior to any other algorithm in terms of scaling. This is because while with other algorithms the time needed to find a key in the table or reject it grows with the number of the keys in the table N, hash searching makes it constant regardless of N. In other words, searching time is the same whether the table contains 100 or 1,000,000 entries.

However, before you can use the hash object to do all the wonderful things it offers, you have to learn a few things:

- The hash object does not understand the DATA step language and must be instead addressed using the so-called *object-dot syntax*. Luckily, to the extent of the hash object scope, it is quite simple to master.
- You have to understand that all hash object operations – including the creation and deletion of the object itself and memory allocation for each new entry – are performed at the DATA step run time, rather than compile time.
- You need to learn how to prepare the DATA step environment for communicating with the hash object, for without such preparation the object cannot even be defined, let alone serve a useful purpose.

DECLARING AND INSTANTIATING

Before the hash object can be used for any purpose, three things must happen:

1. It has to be declared.
2. It has to be instantiated.
3. It has be defined.

Example 1: Declaring a Hash and Creating Its Instance Separately

```
data _null_ ;
   declare hash hh ;
   hh = _new_ hash() ;
run ;
```

This code runs error-free, which means that the DATA step compiler accepts its syntax as valid. Let us observe several essential facts regarding the two lines of hash code above:

- The hash object is manipulated from within the DATA step environment. As of Version 9.4, it can also be operate from within the DS2 procedure. However, it cannot be called from the SQL procedure (notwithstanding the fact that its own hashing algorithm can be used by the SQL optimizer internally).
- The DECLARE statement declares a hash object (DECLARE can be abbreviated as just DCL).
- HASH is a keyword, thus you must code it just like that.
- HH is the name given to the hash object. It doesn't have to be HH, of course: It can be any *SAS name valid* according to the value of the VALIDVARNAME= option - including VALIDVARNAME=ANY. With the latter, you can call the object, for example, "my hash"N or even "#"N if you wish (of course, all the ensuing references must adhere to the same nomenclature).

However, the acts of declaration and instantiation can be combined in a single statement.

Example 2: Declaring a Hash and Creating Its Instance in a Single Statement

```
data _null_ ;
   dcl hash hh() ;
run ;
```

This single statement has the same effect as the two statements above. Hence the question “why then bother with the separate declaration/instantiation at all?” is not illogical. The answer is that the separate declaration/instantiation is more flexible and can be used to create more than one instance of the same declared hash object. (Since this is related to more advanced hash usage, we shall discuss it closer to the end of the paper.)

Having declared and created a hash instance does not yet mean that it is ready to be used. Another piece of code is needed to define it, and it means:

- Defining the key portion hash variables.
- Defining the data portion hash variables.
- Making sure that the *host* variables with the same names exist in the PDV (parameter type matching).
- Telling the object to complete the definition process.

KEYS AND DATA

From the standpoint of a user, the hash object is simply a data file residing completely in memory. If you can visually picture a SAS data file, you have already pictured a hash table. However, besides existing in memory rather than on disk, the hash table has another important distinction from a SAS data set, namely:

- one or more of its columns (variables) *must* be designated as the key portion of the table
- one or more columns *must* be designated as the data portion of the table

Hence, the hash object provides means to tell which columns are assigned to hold the key and which – the satellite data. This is done using the *hash object methods* DefineKey and DefineData. Imagine that we have 2 columns: one numeric variable KN, and one character variable DC. How do we tell the hash object HH to designate KN as the key into the table and DC – as the data portion of the table? One way is to hard code the variable names as *character literals* as method arguments:

Example 3: Defining Keys and Data

```
data _null_ ;
  length kn 8 dc $ 6 ;

  dcl hash hh () ;
  hh.DefineKey  ('kn') ;
  hh.DefineData ('dc') ;
  hh.DefineDone () ;
run ;
```

Note the following:

- Calling the three *methods* above requires *object-dot syntax* in the form hh.<method name>(). Though it's somewhat foreign to the DATA step of yore, the compiler recognizes it as valid.
- HH used in the calls references the hash name HH selected in the preceding DCL statement.
- hh.DefineDone() is a request to check if the object has all it needs to function and complete the definition.

Now, what is the importance of the LENGTH statement here? A simple test shows that if we comment it out, SAS will abend the step:

```
1  data _null_ ;
2  * length kn 8 dc $ 6 ;
3    dcl hash hh () ;
4    hh.DefineKey  ('kn') ;
5    hh.DefineData ('dc') ;
6    hh.DefineDone () ;
7  run ;
ERROR: Undeclared key symbol kn for hash object at line 6 column 3.
ERROR: DATA STEP Component Object failure. Aborted during the EXECUTION phase.
```

If, instead, we keep the LENGTH statement but exclude the DC variable from it, we will get the same error messages, except that now the first one will reference the data portion:

```
ERROR: Undeclared data symbol dc for hash object at line 6 column 3.
```

As any SAS programmer know, the SAS log is the best storyteller. In this case, it's telling that:

- There are no compile time errors.
- However, *at the run time*, the hash object fails the attempt at definition. As the log indicates, it happens at the line where hh.DefineDone() method is called.
- The reason why it happens is that *no host variable* named KN (or DC in the second case) is present in the PDV (or, put it another way, in the compiler symbol table).

Of course, LENGTH statement is not the only - and certainly not the most convenient - means to create a hash host variables. We shall dwell on this in more detail later.

ADDING KEYS AND DATA

After the hash object has been instantiated and the key and data portions have been defined, it can be populated with key and data values. Due to its execution-time nature, doing so may take on a variety of forms; however, all of them use ADD or REPLACE methods explicitly or behind the scenes.

Example 4: Adding Keys and Data One at a Time

```
data _null_ ;
   length KN 8 DC $ 6 ;

   dcl hash hh () ;
   hh.DefineKey  ('kn') ;
   hh.DefineData ('kn', 'dc') ;
   hh.DefineDone () ;

   kn = 2 ; dc = 'data_2' ; rc = hh.add() ;
   kn = 1 ; dc = 'data_1' ; rc = hh.add() ;
   kn = 3 ; dc = 'data_3' ; rc = hh.add() ;

   hh.output (dataset: 'hh') ;
run ;
```

In the example above:

- ADD method inserts the keys and data into the table one at a time. Checking the values of RC, you will find that they resolve to zeroes, meaning that ADD method has inserted the values successfully.
- Another method we are learning from this example, is OUTPUT method. It instructs the hash object to write its *data portion* to a SAS data file, whose name is passed to the *argument tag* DATASET (notice that the assignment requires a semicolon).
- As noted earlier, *only the data portion can be retrieved from the table*. Column KN is passed to DEFINEDATA method because we want KN to be included in the output. Accordingly, the content of data file HH will look like this:

```
KN       DC
------------
2    data_2
1    data_1
3    data_3
```

- If we didn't define KN in the data portion, the KN column would be absent. Besides being useful in its own right, OUTPUT method is an excellent diagnostic tool, as it can be used to easily see the content of the table by viewing a SAS data file.

Above, the keys and data were added one statement at a time. However, since any hash method is a run-time call, we can follow a natural programming impulse and add them in a loop.

Example 5: Adding Keys and Data in a Loop

```
data _null_ ;
   length KN 8 DC $ 6 ;

   dcl hash hh () ;
   hh.DefineKey  ('kn') ;
   hh.DefineData ('kn', 'dc') ;
   hh.DefineDone () ;

   do kn = 2, 1, 3 ;
```

```
      dc = catx ('_', 'data', kn) ;
      hh.add() ;
   end ;

   hh.output (dataset: 'hh') ;
run ;
```

SEARCH AND RETRIEVAL

Key search and data retrieval are likely the two operations the hash object is asked to do most often. Let's look at them separately:

- Many tasks - file subsetting or unduplication could be good examples - require nothing more than to quickly determine if a given key value is in the table. For this purpose, the hash object is packaged with a method called CHECK.
- On the other hand, if you need to search the table for a key and, *in the case it is found*, overwrite the PDV host variables with the values from the data portion of the corresponding hash entry, FIND method is the tool. Thus, it combines searching with moving data from the hash table to the PDV.

In this example, CHECK method is used to determine whether keys KN=2 and KN=5 are in the table. Then FIND method is used to search for a key and, if the key is found, update host variable DC with the value of its hash table counterpart.

Example 6: CHECK vs FIND

```
data _null_ ;
   length KN 8 DC $ 6 ;

   dcl hash hh () ;
   hh.DefineKey  ('kn') ;
   hh.DefineData ('kn', 'dc') ;
   hh.DefineDone () ;

   do kn = 2, 1, 3 ;
      dc = catx ('_', 'data', kn) ;
      hh.add() ;
   end ;

   call missing (of _all_) ;

   kn = 2 ; rc = hh.check() ; put (kn dc rc) (=) ;
   kn = 5 ; rc = hh.check() ; put (kn dc rc) (=) ;

   kn = 2 ; rc = hh.find() ; put (kn dc rc) (=) ;
   kn = 5 ; rc = hh.find() ; put (kn dc rc) (=) ;
run ;
--------------------------------------------------
KN=2 DC=  rc=0
KN=5 DC=  rc=-2147450842
KN=2 DC=data_2 rc=0
KN=5 DC=data_2 rc=-2147450842
```

Since KN=2 is in the table, CHECK method finds the key and returns RC=0. For KN=5, CHECK returns a non-zero RC indicating that the key is not in the table. Note that CHECK *does not retrieve* any hash data even when the key is found, and DC remains blank. This is because CHECK method is purposely designed to *not* overwrite the host variables in the PDV.

On the other hand, when FIND method finds the key, it writes the corresponding value DC=data_2 to the host variable DC. If the key is not found, FIND does not retrieve anything, and so DC remains missing.

This test demonstrates the fundamental difference between the CHECK and FIND methods. CHECK facilitates the *search* operation only. FIND facilitates the *retrieve* operation, i.e. "search plus overwrite the host variable" as a combination.

BEYOND PROPAEDEUTICS

Now we have learned enough to move on to more practical examples. Given the ability of the hash object to search for a key and retrieve data in *O(1)* time, the first application coming to mind is file matching. When two data files have to be matched by a key, the usual (and oldest) approach is to sort them by the key and MERGE. An SQL join is an alternative, but in most situations, it also involves sorting the files behind-the-scenes. However, if you have the ability to fast-search a memory table in time independent of the number of keys in the table, then one of the files can be placed in memory and searched for every key coming from the other file.

FILE MATCH

So, perhaps the best way to get a taste of this mighty addition to the DATA step family is to see how easily it can help solve the “file match problem”. Suppose we have a SAS file LARGE with $9 character key KEY, and a file SMALL with a similar key and some additional info stored in a numeric variable S_SAT. We need to use the SMALL file as a fast lookup table to pull S_SAT into LARGE for each KEY having a match in SMALL. This step shows one way how the hash (associative array) object can help solve the problem:

Example 7: File Matching (Lookup file loaded in a loop)

```
data match (drop = rc) ;
 * length key $9 s_sat 8 ; *Note: not needed here because of SET SMALL ;
   declare hash hh () ;
   hh.DefineKey  ('key'  ) ;
   hh.DefineData ('s_sat') ;
   hh.DefineDone () ;
   do until (eof1) ;
      set small end = eof1 ;
      rc = hh.add() ;
   end ;
   do until (eof2) ;
      set large end = eof2 ;
      rc = hh.find() ;
      if rc = 0 then output ;
   end ;
   stop ;
run ;
```

After all the trials and tribulations of coding hashing algorithms by hand, this simplicity looks rather stupefying. But how does this code go about its business?

- The LENGTH statement is not needed here for parameter type matching. This is because the compiler derives KEY and S_SAT for the PDV by reading the header of SMALL. Using the LENGTH statement would only create a potential for a data attribute conflict.
- DECLARE hash statement declares and instantiates the associative array (hash table) HH.
- DefineKey method describes the variable(s) to serve as a key into the table.
- DefineData method is called if there is a non-key satellite information, in this case, S_SAT, to be loaded in the table.
- DefineDone method is called to complete the initialization of the hash object.
- ADD method grabs a KEY and S_SAT from SMALL and loads both in the table. *Note that for any duplicate KEY coming from SMALL, ADD() will return a non-zero code and discard the key, so only the first instance of the satellite corresponding to a non-unique key will be used.*
- FIND method searches the hash table HH for each KEY coming from LARGE. If it is found, the return code is set to zero, and host S_SAT field is updated with its value extracted from the hash table.

If you think it is *prorsus admirabile*, then the following step does the same with even less coding:

Example 8: File Matching (Lookup file is loaded via the DATASET: parameter)

```
data match ;
   if 0 then set small ; * get key/data attributes for parameter type matching ;

   dcl hash hh    (dataset: 'work.small', hashexp: 10) ;
   hh.DefineKey  ('key'   ) ;
   hh.DefineData ('s_sat') ;
   hh.DefineDone () ;

   do until ( eof2 ) ;
      set large end = eof2 ;
      if hh.find () = 0 then output ;
   end ;
   stop ;
 * set small ; * alternative to IF 0 THEN SET SMALL atop ;
run ;
```

Here are notable differences:

- Instead of loading keys and satellites from SMALL in an explicit DO loop, we can instruct the hash table constructor to load the table directly from the SAS data file SMALL by specifying the file in the hash declaration.
- Now that SMALL is not referenced in a DO loop, we need to do something else to create the host variable S_SAT before the hash methods are called. We could use the LENGTH statement. However, it is much more convenient and less error-prone to include a non-executable SET SMALL statement somewhere in the step and let the compiler read its descriptor containing S_SAT. IF 0 THEN ensures that the SET statement reads no physical records from the file - the hash object will do it for us. The same can be alternatively (and perhaps even better) achieved by placing SET SMALL statement after the STOP statement.
- DCL can be used as a shorthand for DECLARE.
- The object parameter HASHEXP tells the table constructor to allocate 2**10=1024 hash buckets.
- Assigning return codes to a variable when the methods are called is not mandatory. Omitting the assignments shortens notation.

PARAMETER TYPE MATCHING - REVISITED

- The LENGTH statement or the attribute-extracting SET SMALL provide for what is called *parameter type matching*. Either technique lets the DATA step compiler see the variables given to the hash object as keys and data and put them in the PDV. A few notable facts are worth observing in this regard:

- This is necessary because the compiler, while scanning the step, *does not see* anything in the hash object code except for the syntax to check. Therefore, it cannot place KEY and S_SAT into the PDV merely because they are named as literals in the method calls.
- By contrast, if KEY and S_SAT are present in the step outside the hash code either by named references (such as in the LENGTH, assignment, FORMAT, sum statements, to name a few) or in the descriptor of a data set named in a file-reading statement (such as SET, MERGE, etc.), the compiler *does* see them. Hence, it places them, along with their attributes, in the PDV.
- Since this process happens during the compile time and all hash-related code is executed during the run time (i.e. after the compile phase is complete), by the time any hash code is executed, KEY and S_SET are already in the PDV and prepared to act as *host variables* for the hash object.
- As a corollary, from the standpoint of the hash object, it does not matter where in the step the compiler has seen KEY and S_SAT first, even if it happens in the very last statement in the step, such as after the STOP statement above.
- It stands to reason why the host variables must be present in the PDV before a hash data *retrieval* method, such as FIND, can be called. Indeed, if we call hh.FIND() without arguments, it can get the value of KEY to search for only from the variable already in the PDV. On the way back, if S_SAT were not in the PDV, the FIND method would not have a container into which to *retrieve* the S_SAT hash value if KEY is in the table.

It is incumbent upon the programmer to observe proper parameter type matching and make sure that all the hash variables referenced in the hash method calls are in the PDV.

HANDLING DUPLICATE KEYS

When a hash table is loaded from a data set, SAS acts as if the ADD method were used, that is, all duplicate key entries but the very first are ignored. Now, what if in the file SMALL, duplicated keys corresponded to different satellite values, and we needed to pull *the last instance* of the satellite?
With hand-coded hash schemes, duplicate-key entries can be controlled programmatically by twisting the guts of the hash code. To achieve the desired effect using the hash object, we should call the REPLACE method instead of the ADD method. But to do so, we have to revert back to the loading of the table in a loop one key entry at a time:

```
  do until ( eof1 ) ;
     set small end = eof1 ;
     hh.replace () ;
  end ;
```

However, in Version 9.2, DUPLICATE argument tag values as ‘R’ can be used to keep the *last* duplicate key entry (the tag valued as ‘E’ will report key duplicates in the SAS log as errors):

```
dcl hash hh (dataset: 'work.small', DUPLICATE: 'r', hashexp: 10) ;
```

Note that before Version 9.2, the hash object provides no mechanism of storing and/or handling duplicate keys with different satellites in the same hash table. The multi-hash introduced in Version 9.2 takes care of the problem of harvesting hash data entries with the same key. In versions prior to 9.2, which of the time of this writing are still in production use in most of major shops, this difficulty can be circumvented by discriminating the primary key by creating a secondary key from the satellite, thus making the entire composite key unique. Such an approach is aided by the ease with which hash tables can store and manipulate composite keys. Programmatic ways of storing multiple key entries and manipulating them efficiently are described in [9].

COMPOSITE KEYS AND MULTIPLE SATELLITES

In pre-V9 days, handling composite keys in a hand-coded hash table could be a breeze or a pain, depending on the type, range, and length of the component keys [1]. But in any case, the programmer needed to know the data beforehand and often demonstrate a good deal of ingenuity.
The hash object makes it all easy. The only thing we need to do in order to create a composite key is define the types and lengths of the key components and instruct the object constructor to use them in the specified subordinate sequence. For example, if we needed to create a hash table HH keyed by variables defined as

```
length k1 8 k2 $3 k3 8 ;
```

and in addition, had multiple satellites to store, such as:

```
length a $2 b 8 c $4 ;
```

we could simply code:

```
dcl hash hh () ;
hh.DefineKey  ('k1', 'k2', 'k3') ;
hh.DefineData ('a', 'b', 'c') ;
hh.DefineDone () ;
```

and the internal hashing scheme will take due care about whatever is necessary to come up with a hash bucket number where the entire composite key should fall together with its satellites.
Multiple keys and satellite data can be loaded into a hash table one element at a time by using the ADD or REPLACE methods. For example, for the table defined above, we can value the keys and satellites first and then call the ADD or REPLACE method:

```
   k1 =     1  ; k2 = 'abc' ; k3 =     3  ;
   a  =   'a1' ; b  =     2 ; c  = 'wxyz' ;
   rc = hh.replace () ;
   k1 =     2  ; k2 = 'def' ; k3 =     4  ;
   a  =   'a2' ; b  =     5 ; c  = 'klmn' ;
   rc = hh.replace () ;
```

Alternatively, these two table entries can be coded as

```
hh.replace (key:    1, key: ‘abc’, key: 3,
            data:   ‘a1’, data: 2, data: ‘wxyz’) ;
hh.replace (key:    2, key: ‘def’, key: 4,
            data:   ‘a2’, data: 5, data: ‘klmn’) ;
```

Note that more that one hash table entry cannot be loaded in the table at compile-time at once, as it can be done in the case of arrays. All entries are loaded one entry at a time at run-time.
Perhaps it is a good idea to avoid hard-coding data values in a Data step altogether, and instead always load them in a loop either from a file or, if need be, from arrays. Doing so reduces the propensity of the program to degenerate into an object Master Ian Whitlock calls “wall paper”, and helps separate code from data.

HASH OBJECT PARAMETERS AS EXPRESSIONS

The two steps above may have already given a hash-hungry reader enough to start munching overwhelming programming opportunities opened by the availability of the SAS-prepared hash food without the necessity to cook it. To add a little more spice to it, let us rewrite the step yet another time.

Example 9: File Matching (Using expressions for parameters)

```
data match ;
   set small (obs = 1) ;
   retain dsn ‘small’ x 10 kn ‘key’ dn ‘s_sat’ ;

   dcl hash hh (dataset: dsn, hashexp: x) ;
      hh.DefineKey  ( kn ) ;
       hh.DefineData ( dn ) ;
       hh.DefineDone (    ) ;

   do until ( eof2 ) ;
      set large end = eof2 ;
      if hh.find () = 0 then output ;
   end ;
   stop ;
run ;
```

As we see, the parameters passed to the constructor (such as DATASET and HASHEXP) and methods need not necessarily be hard-coded literals. They can be passed as valued Data step variables, or even as appropriate type expressions. For example, it is possible to code (if need be):

```
retain args ‘small key s_sat’ n_keys 1e6;
dcl hash hh ( dataset: substr(args,1,5)
              hashexp: log2(n_keys)
             ) ;
hh.DefineKey  ( scan(s, 2) ) ;
hh.DefineData ( scan(s,-1) ) ;
hh.DefineDone () ;
```

THE HASH ITERATOR OBJECT

With FIND method, we can retrieve the data portion hash values into their PDV host counterparts for a given search key - if it is in the table. It's somewhat analogous to retrieving the value V of an element of array A as V=A[INDEX]. However, in many practical situations we need to be able to *scan* through all or some hash table entries *sequentially* and dump their data portion values in the PDV one at a time. With the array, we would simply DO-loop from *index=lbound(A)* to *index=hbound(A)*. However, hash object entries are not accessible directly as array elements, so how can we do it?

The answer is that in order to facilitate such *enumeration*, SAS provides the special *hash iterator* object, which you declare using the keyword *HITER*. The iterator method calls make the hash table entries available in the form of a serial list. To make it clear, let us consider a simple program.

Example 10: Dumping the Contents of an Ordered Table Using the Hash Iterator

```
data sample ;
   input k sat ;
cards ;
185  01
971  02
400  03
260  04
922  05
970  06
543  07
532  08
050  09
067  10
 ;
run ;

data _null_ ;
   if 0 then set sample ;
       dcl hash  hh ( dataset: 'sample', hashexp: 8, ordered: 'A') ;
       dcl hiter hi  ( 'hh' ) ;
       hh.DefineKey  ( 'k'          ) ;
       hh.DefineData ( 'sat' , 'k' ) ;
       hh.DefineDone () ;

   do rc = hi.first () by 0 while ( rc = 0 ) ;
      put k = z3. +1 sat = z2. ;
      rc = hi.next () ;
   end ;

   do rc = hi.last () by 0 while ( rc = 0 ) ;
      put k = z3. +1 sat = z2. ;
      rc = hi.prev () ;
   end ;
   stop ;
run ;
```

We see that now the hash table is instantiated with the ORDERED parameter set to 'a', which stands for 'ascending'. When 'a' is specified, the table is automatically loaded in the ascending key order. It would be better to summarize the rest of the meaningful values for the ORDERED parameter in a set of rules:

- 'a' , 'ascending' = ascending
- 'y' = ascending
- 'd' , 'descending' = descending
- 'n' = internal hash order (i.e. no order at all, and the original key order is NOT followed)
- any other character literal different from above = same as 'n'
- parameter not coded at all = the same as 'n' by default
- character expression resolving to the same as the above literals = same effect as the literals
- numeric literal or expression = DSCI execution time object failure because of type mismatch

Note that the hash object symbol name must be passed to the iterator object as a character string, either hard-coded as above or as a character expression resolving to the symbol name of an *already declared hash object*, in this case, "HH". After the iterator HI for the table HH has been successfully declared, it can be used to fetch entries from the hash table in the key order defined by the rules given above.

To retrieve hash table entries in an ascending order, we must first point to the first physical hash entry. Since the table is ordered, it is the entry with the lowest key. This is done using method FIRST:

```
rc = hi.first() ;
```

where HI is the name we have assigned to the iterator. A successful call to FIRST fetches the first table key into host variable K and the corresponding satellite - into host variable SAT. Once this is done, each call to the NEXT method will fetch the hash entry with the next key in ascending order. When no keys are left, the NEXT method returns RC not equal to 0, and so the loop terminates. Thus, the first loop will print in the log:

```
k=050  sat=09
k=067  sat=10
k=185  sat=01
k=260  sat=04
k=400  sat=03
k=532  sat=08
k=543  sat=07
k=922  sat=05
k=970  sat=06
k=971  sat=02
```

Inversely, the second loop retrieves table entries in descending order by starting off with the call to the LAST method fetching the entry with the largest key. Each subsequent call to the method PREV extracts an entry with the next smaller key until there are no more keys to fetch, at which point PREV returns RC > 0, and the loop terminates. Therefore, the loop prints:

```
k=971  sat=02
k=970  sat=06
k=922  sat=05
k=543  sat=07
k=532  sat=08
k=400  sat=03
k=260  sat=04
k=185  sat=01
k=067  sat=10
k=050  sat=09
```

An alert reader might wonder *why does key variable K have to be also defined in the data portion*? After all, each time the DO loop iterates, the iterator fetches a new entry with a different key. The reason is the same as we've seen with FIND and OUTPUT methods. Namely, only the data portion variables are "PDV-dumpable". With respect to their communication with the PDV, the iterator methods FIRST, NEXT, LAST, and PREV act just like successful FIND: They overwrite only the PDV variables hosting their *data portion* counterparts. So, in the step above, no method calls update the host key variable. If K weren't defined in the data portion, only the last key value stored in the PDV before the first DO loop would be printed.

The concept behind such behavior is that only data portion variables "have a right" to have their values written into their PDV host counterparts, whilst the keys do not. It means that if you need to retrieve the key values from a hash table into the PDV, you need to define the *key variables in the data portion* as well.

DATA STEP COMPONENT INTERFACE (DSCI)

Now that we have a taste of the DATA step hash object, let us consider it from a more generic point of view.

In Version 9.0, the hash table introduced the first *component object* accessible via a rather novel thing called *DATA Step Component Interface (DSCI).* A component object is a data structure comprising three elements: a*ttributes, operators, and methods*. *Attributes* are linked with the data the object contains. *Methods* define the operations the object can perform on its data, and *operators* have their own purpose.

From the programming standpoint, an object is a black box with known properties, much like a SAS procedure. However, a SAS procedure, such as SORT or FORMAT, cannot be called from a DATA step at the execution time, while an object accessible through DSCI - can. A DATA step programmer who wants the hash object to perform some operation on its data does not have to program it procedurally, but call an appropriate method instead.

THE OBJECT

In our case, the object is a hash table. Generally speaking, as an abstract data entity, *a hash table is an object providing for the insertion and retrieval of its keyed data entries in O(1), i.e. constant, time*. Properly built direct-addressed tables satisfy this definition *in the strict sense*. We will see that the hash object table satisfies it *in the practical sense*. The attributes of the hash table object are keyed entries comprising its key(s) and maybe also satellites.

Before any hash table object methods can be called (operations on the hash entries performed), the object must be declared. In other words, the hash table must *be instantiated* with the DECLARE (DCL) statement, as we have seen above.

METHODS

The hash table methods are used to tell a hash object which functions to perform and how. New methods are being added in almost each new SAS release and version. Hence, the list below is by no means exhaustive.

- DefineKey. Define a set of hash keys.
- DefineData. Define a set of hash table satellites. This method call can be omitted without harmful consequences if there is no need for non-key data in the table. Although a dummy call can still be issued, it is not required.
- DefineDone. Tell SAS the definitions are done. If the DATASET argument is passed to the table's definition, load the table from the data set.
- ADD. Insert the key and satellites if the key is not yet in the table (ignore duplicate keys).
- REPLACE. If the key is not in the table, insert the key and its satellites, otherwise overwrite the satellites in the table for this key with new ones.
- REMOVE. Delete the entire entry from the table, including the key and the data.
- FIND. Search for the key. If it is found, extract the satellite(s) from the table and update the host Data step variables.
- REF. Consolidate FIND and ADD methods into a single method call. Particularly useful in summarizations where it provides for a simpler programming logic compared to calling FIND and ADD separately.
- CHECK. Search for the key. If it is found, just return RC=0, and do nothing more. Note that calling this method does not overwrite the host variables.
- OUTPUT. Dump the entire current contents of the table into a one or more SAS data set. Note that for the key(s) to be dumped, they must be defined using the DefineData method. If the table has been loaded in order, it will be dumped also in order. More information about the method will be provided later on.
- CLEAR. Remove all items from a hash table without deleting the hash object instance. In many cases it is desirable to empty a hash table without deleting the table itself - the latter takes time. For instance, in Examples 6 and 9 below, program control redefined the entire hash instance from scratch when it passes through its declaration before each BY-group. For better performance, declaration could be executed only once, and the table - cleaned up by calling the CLEAR method instead.

- EQUAL (new in 9.2). Determine if two hash objects are equal.
- SETCUR (new in 9.2). Specify a key from which iterator can start scrolling the table. It is a substantial improvement over 9.1 where the iterator could start only either from the beginning or end of the table.
- FIRST. Using an iterator, fetch the item stored in a hash table *first*. If the table is ordered, the lowest-value item will be fetched. If none of the items have been fetched yet, the call is similar to NEXT.
- LAST. Using an iterator, fetch the item stored in a hash table *last*. If the table is ordered, the highest-value item will be fetched. If none of the items have been fetched yet, the call is similar to PREV.
- NEXT. Using an iterator, fetch the item stored *right after* the item fetched in the previous call to FIRST or NEXT from the same table.
- PREV. Using an iterator, fetch the item stored *right before* the item fetched in the previous call to LAST or PREV from the same table.
- SUM (new in 9.2). If SUMINC argument tag has been used in a hash table declaration, use SUM method to retrieve summary counts for each (distinct) key stored in the table.

ATTRIBUTES

Just as a SAS data file has a descriptor containing items which can be retrieved without reading any data from the file (most notably, the number of observations), a hash object has attributes which can be retrieved without calling a single method. Currently, there are 2 attributes:

1.ITEM_SIZE. Returns a size of a hash object item in bytes.
2.NUM_ITEMS. Returns the total number of items stored in a hash object.

Note that the attributes are returned directly into a numeric SAS variable, and their syntax differs from that of methods in that parentheses are not used. For example for a hash object HH:

```
item_size = HH.item_size ;
num_items = HH.num_items ;
```

OPERATORS

Currently, there is only one operator:

1._NEW_. Use it to create an instance of already declared component object.

The _NEW_ operator is extremely useful when there is a need to create more than one instance of a hash object and have the ability to store and manage each of them separately. Example 10 below and the ensuing discussion provide a good deal of detail about the operator using a real-world task.

OBJECT-DOT SYNTAX

As we have seen, in order to call a method, we only have to specify its name preceded by the name of the object followed by a period, such as:

```
hh.DefineKey ()
hh.Find ()
hh.Replace ()
hh.First ()
hh.Output ()
```

and so on. This manner of telling SAS Data step what to do is thus naturally called the *Data Step Object Dot Syntax*. Summarily, it provides a linguistic access to a component object’s methods and attributes. Note that the object dot syntax is one of very few things the Data step compiler knows about DSCI. The compiler recognizes the syntax, and it reacts harshly if the dot syntax is present but the object to which it is apparently applied is absent. For example, an attempt to compile this step,

```
data _null_ ;
   hh.DefineKey (‘k’) ;
run ;
```

results in the following log message *from the compiler* (not from the object):

```
6  data _null_ ;
ERROR: DATA STEP Component Object failure.  Aborted during the COMPILATION phase.
7     hh.DefineKey (‘k’) ;
        ____________
        557
ERROR 557-185: Variable hh is not an object.
8  run ;
```

So far, just a few component objects are accessible from the DATA step through DSCI. However, as their number grows, we had better get used to the object dot syntax really soon, particularly those *dinosaurs* among us who have not exactly learned this kind of tongue in the SAS kindergarten.

A PEEK UNDER THE HOOD

We have just seen the tip of the hash iceberg from the outside. An inquiring mind would like to know: What is inside? Not that we really need the gory details of the underlying code, but it is instructive to know on which principles the design of the internal SAS table is based in general. A good driver is always curious what is under the hood.

Well, *in general*, hashing is hashing is hashing - which means that it is always a two-staged process: (1) hashing a key to its bucket and (2) resolving collisions within each bucket. Hand-coded hashing cannot rely on the simple straight separate chaining because of the inability to dynamically allocate memory one entry at a time, while reserving it in advance could result in unreasonable waste of memory.
Since the hash and hiter objects are coded in the underlying software, this restriction no longer exists, and so separate chaining is perhaps the most logical way to go. Its concrete implementation, however, has somewhat deviated from the classic scheme of connecting keys within each node into a link list. Instead, each new key hashing to a bucket is inserted into its own binary tree. If there were, for simplicity, only 4 buckets, the scheme might roughly look like this:

```
     0         1         2         3
+---------+---------+---------+---------+
|    |    |    |    |    |    |    |    |
|   / \   |   / \   |   / \   |   / \   |
|  /\ /\  |  /\ /\  |  /\ /\  |  /\ /\  |
```

The shrub-like objects inside the buckets are *AVL (Adelson-Volsky & Landis)* trees. AVL trees are binary trees populated by such a mechanism that on the average guarantees their *O(log(N))* search time scaling behavior regardless of the distribution of the input key values.

Let us see how, given a KEY, this structure facilitates searching:

- The internal *hash function* maps the key, whether is it simple or composite, to an integer number.
- If there is no bucket with such a number, the key is not found.
- If there is, the tree in the bucket with this number is searched for KEY using binary search.
- If KEY is not there, the process stops.
- Otherwise, further actions depend on which method has been called.

Thus, the SAS object hash table behaves as *O(log(N/HSIZE))*. While it is not *exactly O(1),* it can be considered approximately such for all practical intents and purposes, as long as N/HSIZE is not too large. The number of hash buckets allocated by the hash table constructor is controlled by the HASHEXP parameter we have used above. The number of buckets is 2**HASHEXP. So, if HASHEXP=8, HZISE=256 buckets will be allocated, or if HASHEXP=20, HSIZE=1,048,576. As of the moment, 20 is the maximum; any HASHSIZE specified over 20 is truncated to 20.

How does HASHEXP affect the speed of search? Suppose that we have N=2**24, i.e. about 16.7 million keys in the table. With HASHEXP=20, I.E. HSIZE=2**20, there will be 2**4=16 keys hashing, on the average, to one bucket. Searching for a key among 16 AVL tree keys requires about 5 comparisons. Now if we had HASHEXP=8 (the default), i.e. HSiZE=2**8, there would be 2**16 keys, on the average, hashing to one bucket tree. That would require 17 comparisons to find or reject a search key. And indeed, a simple test shows that HASHEXP=20 is about twice as fast as HASHEXP=8.

The penalty comes in the form of the amount of *base memory* required to allocate 2**20 trees versus 2**8. However, it's not too severe: On the Windows-64 platform, base memory needed for HASHEXP=20 is about 17 Mb, while for HASHEXP=8 it is about 1 Mb. Despite the 17 times difference, in today's world of gigantic cheap memories 17 Mb is of little consequence. In our experience, you can code HASHEXP=20 under any circumstances and just forget about a few megabytes of wasted memory. The real limitation on the hash object memory footprint comes from the length of its entries times their number.

DYNAMIC DATA STEP STRUCTURE

The hash table object (together with its hiter sibling) represents the *first ever dynamic Data step structure*, i.e. one capable of acquiring memory and growing at run-time. There are a number of common situations in data processing when the information needed to size a data structure becomes available only at execution time. SAS programmers usually solve such problems by pre-processing data, either i.e. passing through the data more than once, or allocating memory resources for the worst-case scenario. As more programmers become familiar with the possibilities this dynamic structure offers, they will be able to avoid resorting to many old kludges.
Remember, a hash object is instantiated during the run-time. If the table is no longer needed, it can be simply wiped out by the DELETE method:

```
rc = hh.Delete () ;
```

This will eliminate the table from memory for good, but not its iterator! As a separate object related to a hash table, it has to be deleted separately:

```
rc = hi.Delete () ;
```

If at some point of a Data step program there is a need to start building the same table from scratch again, remember that the *compiler must see only a single definition of the same table* by the same token as it must see only a single declaration of the same array (and if the rule is broken, it will issue the same error message, e.g.: "Variable hh already defined"). Also, like in the case of arrays, the full declaration (table and its iterator) must precede any table/iterator references. In other words, this will NOT compile because of the repetitive declaration:

```
20   data _null_ ;
21      length k 8 sat $11 ;
22
23      dcl hash  hh   ( hashexp: 8, ordered: 1 ) ;
24      dcl hiter hi   ( 'hh'   ) ;
25      hh.DefineKey   ( 'k'    ) ;
26      hh.DefineData  ( 'sat'  ) ;
27      hh.DefineDone  () ;
28
29      hh.Delete () ;
30
31      dcl hash   hh   (hashexp: 8, ordered: 1 ) ;
                        -
                        567
ERROR 567-185: Variable hh already defined.
32      dcl hiter   hi   ( 'hh'   ) ;
                         -
                         567
ERROR 567-185: Variable hi already defined.
```

And this will not compile because at the time of the DELETE method call, the compiler has not seen HH yet:

```
39   data _null_ ;
40      length k 8 sat $11 ;
41      link declare ;
42      rc = hh.Delete() ;
             _________
             557
             68
ERROR 557-185: Variable hh is not an object.
ERROR 68-185: The function HH.DELETE is unknown, or cannot be accessed.
43      link declare ;
44      stop ;
45      declare:
46         dcl hash  hh   ( hashexp: 8, ordered: 1 ) ;
47         dcl hiter hi   ( 'hh'   ) ;
48         hh.DefineKey   ( 'k'    ) ;
49         hh.DefineData  ( 'sat'  ) ;
50         hh.DefineDone  () ;
51      return ;
52      stop ;
53   run ;
```

However, if we do not dupe the compiler and reference the object after it has seen it, it will work as designed:

```
199  data _null_ ;
200     retain k 1 sat 'sat' ;
201     if 0 then do ;
202        declare:
203        dcl hash  hh   ( hashexp: 8, ordered: 1 ) ;
```

```
      dcl hiter hi  ( ‘hh’  ) ;
      hh.DefineKey  ( ‘k’   ) ;
      hh.DefineData ( ‘sat’ ) ;
      hh.DefineDone () ;
      return ;
   end ;
   link declare ;
      rc =  hi.First  () ;
      put k= sat= ;
      rc = hh.Delete () ;
      rc = hi.Delete () ;
   link declare ;
      rc = hh.Delete () ;
      rc = hi.Delete () ;
   stop ;
run ;
```

```
k=1 sat=sat
```

Of course, the most natural and trouble-free technique of having a table created, processed, cleared, and created from scratch again is to place the entire process in a loop. This way, the declaration is easily placed ahead of any object references, and the compiler sees the declaration just once. In a moment, we will see an example doing exactly that.

DYNAMIC DATA STEP DATA DICTIONARIES

The fact that hashing supports searching (and thus retrieval and update) in constant time makes it ideal for using a hash table as a dynamic Data step data dictionary. Suppose that during DATA step processing, we need to memorize certain key elements and their attributes on the fly, and at different points in the program, answer the following:

1. Has the current key already been used before?
2. If it is new, how to insert it in the table, along with its attribute, in such a way that the question #1 could be answered as fast as possible in the future?
3. Given a key, how to update a corresponding satellite rapidly?
4. If the key is no longer needed, how to delete it?

Examples showing how key-indexing can be used for this kind of task are given in [1]. Here we will take an opportunity to show how the hash object can help an unsuspecting programmer. Imagine that we have input data of the following arrangement:

```
data sample ;
   input id transid amt ;
   cards ;
1  11   40
1  11   26
1  12   97
1  13    5
1  13    7
1  14   22
1  14   37
1  14    1
1  15   43
1  15   81
3  11   86
3  11   85
3  11    7
3  12   30
3  12   60
3  12   59
3  12   28
```

```
3  13   98
3  13   73
3  13   23
3  14   42
3  14   56
;
run ;
```

The file is grouped by ID and TRANSID. We need to summarize AMT within each TRANSID giving SUM, and for each ID, output 3 transaction IDs with largest SUM. Simple! In other words, for the sample data set, we need to produce the following output:

```
id    transid    sum
-------------------------
 1       15      124
 1       12       97
 1       11       66
 3       13      194
 3       11      178
 3       12      177
```

Usually, this is a 2-step process, either in the foreground or behind the scenes (SQL). Since the hash object table can eject keyed data in a specified order, it can be used to solve the problem *in a single step*:

Example 11: Using the Hash Table as a Dynamic Data Step Dictionary

```
data id3max (keep = id transid sum) ;
   length transid sum 8 ;
   dcl hash  ss   (hashexp: 3, ordered: ‘a’) ;
   dcl hiter si   ( ‘ss’ ) ;
   ss.defineKey   ( ‘sum’              ) ;
   ss.defineData  ( ‘sum’, ‘transid’ ) ;
   ss.defineDone () ;
   do until (last.id) ;
      do sum = 0 by 0 until (last.transid) ;
         set sample ;
         by id transid ;
         sum ++ amt ;
      end ;
      rc = ss.replace() ;
   end ;
   rc = si.last() ;
   do cnt = 1 to 3 while ( rc = 0 ) ;
      output ;
      rc = si.prev() ;
   end ;
run ;
```

It works in the following manner:

1. The inner Do-Until loop iterates over each BY-group with the same TRANSID value and summarizes AMT.
2. The outer Do-Until loop cycles over each BY-group with the same ID value and for each repeating ID, stores TRANSID in the hash table SS keyed by SUM. Because the REPLACE method is used, in the case of a tie, the last TRANSID with the same sum value takes over.
3. At the end of each ID BY-group, the iterator SI fetches TRANSID and SUM in the order descending by SUM, and the top three entries are written to the output file.
4. Control is then passed to the top of the implied Data step loop where it encounters the table definition. It causes the old table and iterator to be dropped and recreated from scratch.
5. If the file has not run out of records, the outer Do-Until loop goes on to process the next ID.

NWAY SUMMARY-LESS SUMMARIZATION

The SUMMARY procedure is an extremely useful (and widely used) SAS tool. However, it has one notable shortcoming: It does not operate quite well when the cardinality of its categorical variables is high. The problem here is that SUMMARY tries to build a memory-resident binary tree for each combination of the categorical variables, and because SUMMARY can do so much, the tree carries a lot of baggage. The result is poor memory utilization and slow run times. The usual way of mitigating this behavior is to sort the input beforehand and use the BY statement instead of the CLASS statement. This usually allows running the job without running out of memory, but the pace is even slower - because now, SUMMARY has to reallocate its tree for each new incoming BY-group.

The hash object also holds data in memory and has no problem handling any composite key, but it need not carry all the baggage SUMMARY does. So, if the only purpose is, say, NWAY summarization, hash may do it much more economically. Let us check it out by first creating a sample file with 1 million distinct keys and 3-4 observations per key, then summarizing NUM within each group and comparing the run-time stats. For the reader’s convenience, they were inserted fro the log after the corresponding steps below where relevant:

Example 12: Summary-less Summarization

```
data input ;
   do k1 = 1e6 to 1 by -1 ;
      k2 = put (k1, z7.) ;
      do num = 1 to ceil (ranuni(1) * 6) ;
         output ;
      end ;
   end ;
run ;
NOTE: The data set WORK.INPUT has 3499159 observations and 3 variables.

proc summary data = input nway ;
   class k1 k2 ;
   var num ;
   output out = summ_sum (drop = _:) sum = sum ;
run ;
NOTE: There were 3499159 observations read from the data set WORK.INPUT.
NOTE: The data set WORK.SUMM_SUM has 1000000 observations and 3 variables.
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           24.53 seconds
      user cpu time       30.84 seconds
      system cpu time     0.93 seconds
      Memory              176723k

data _null_ ;
   if 0 then set input ;
   dcl hash hh (hashexp:16) ;
       hh.definekey  (‘k1’, ‘k2’       ) ;
       hh.definedata (‘k1’, ‘k2’, ‘sum’) ;
       hh.definedone () ;
   do until (eof) ;
      set input end = eof ;
      if hh.find () ne 0 then sum = 0 ;
      sum ++ num ;
      hh.replace () ;
   end ;
   rc = hh.output (dataset: ‘hash_sum’) ;
run ;
NOTE: The data set WORK.HASH_SUM has 1000000 observations and 3 variables.
NOTE: There were 3499159 observations read from the data set WORK.INPUT.
NOTE: DATA statement used (Total process time):
      real time           10.54 seconds
      user cpu time       9.84 seconds
      system cpu time     0.53 seconds
      Memory                                  58061k
```

Apparently, the hash object does the job more than twice as fast, at the same time utilizing 1/3 the memory. And at that, the full potential of the hash method has not been achieved yet. Note that the object cannot add NUM to SUM directly in the table, as is usually the case with arrays. Due to the very nature of the process, for each incoming key, SUM first has to be dumped into its host variable, then have the next value added, and finally reinserted into the table. There is a good indication that in the future, a means will be provided for automatically aggregating some statistics specified to the object constructor. In fact, in the beta of Version 9.2 such provisions seem to have already been made.

DYNAMIC FILE SPLITTING

SAS programmers have now been lamenting for years that the Data step does not afford the same functionality with regard to output SAS data sets it affords with respect to external files by means of the FILEVAR= option. Namely, consider an input data set similar to that we have already used for Example 6, but with five distinct ID values, by which the input is grouped:

```
data sample ;
   input id transid amt ;
   cards ;
1  11   40
1  11   26
1  12   97
2  13    5
2  13    7
2  14   22
3  14    1
4  15   43
4  15   81
5  11   86
5  11   85
;
run ;
```

Imagine that we need to output five SAS data files, amongst which the records with ID=5 belong to a SAS data set OUT1, records with ID=2 belong to OUT2, and so on. Imagine also that there is an additional requirement that each partial file is to be sorted by TRANSID AMT.

To accomplish the task in the pre-V9 software, we need to tell the Data step compiler precisely which output SAS data set names to expect by listing them all in the DATA statement. Then we have to find a way to compose conditional logic with OUTPUT statements directing each record to its own output file governed by the current value of ID. Without knowing ahead of the time the data content of the input data set, we need a few steps to attain the goal. For example:

Example 13: SAS File Split in the pre-V9-hash Era

```
proc sql noprint ;
   select distinct 'OUT' || put (id, best.-l)
   into : dslist
   separated by ' '
   from   sample
   ;
   select 'WHEN (' || put (id, best.-l) || ') OUTPUT OUT' || put (id, best.-l)
   into : whenlist
   separated by ';'
   from   sample
   ;
quit ;
proc sort data = sample ;
   by id transid amt ;
```

```
run ;
data &dslist ;
   set sample ;
   select ( id ) ;
      &whenlist ;
      otherwise ;
   end ;
run ;
```

In Version 9, not only the hash object is instantiated at the run-time, but its methods also are run-time executables. Besides, the parameters passed to the object do not have to be constants, but they can be SAS *expressions*. Thus, at any point at run-time, we can use the OUTPUT method to dump the content of an entire hash table into a SAS data set, whose very name is formed using the ID value of the current by-group, and write the file out in one fell swoop:

Example 14: SAS File Split Using the Hash OUTPUT Method

```
data _null_ ;
   dcl hash hid (ordered: 'a') ;
   hid.definekey  ('_n_')                  ; * to output the whole by-group ;
*  hid.definekey  ('transid')              ; * to kill key duplicates ;
   hid.definedata ('id', 'transid', 'amt') ;
   hid.definedone ( ) ;

   do _n_ = 1 by 1 until ( last.id ) ;
      set sample ;
      by id ;
      hid.replace() ;
   end ;
   hid.output (dataset: cats ('OUT', id)) ;
run ;
```

Having _N_ as the key ensures that all the records from each by-group are output, even if they contain duplicates by TRANSID. If such duplicates are to be deleted, we only need to drop _N_ from the DefineKey method list. This step produces the following SAS log notes:

```
NOTE: The data set WORK.OUT1 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT2 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT3 has 1 observations and 3 variables.
NOTE: The data set WORK.OUT4 has 2 observations and 3 variables.
NOTE: The data set WORK.OUT5 has 2 observations and 3 variables.
NOTE: There were 11 observations read from the data set WORK.SAMPLE.
```

To the eye of an alert reader knowing his DATA step, but not having yet gotten his feet wet with the hash object, the step above must look like a heresy. Indeed, how is it possible to output a SAS data set in a Data _Null_ step? It is possible because with the hash object the output is handled completely by hash the object when the .OUTPUT() method is called, the DATA step merely serving as a shell and providing parameter values to the object constructor.

The step works as follows:

- Before the first record in each BY-group by ID is read, program control encounters the hash table declaration. It can be executed successfully because the compiler has provided the host variables for the keys and data.
- Then the DoW-loop reads the by-group with the next ID value one record at a time.
- If _N_ is the key, each record from the by-group is loaded into HID since _N_ for each record is unique.
- If TRANSID is the key, only the last instance of AMT is kept.
- Thus, when the DoW-loop is done, the hash table for this by-group is fully populated.
- Program control moves to the HID.OUTPUT() method. The current ID value is passed to the expression, which is the argument of the DATASET: tag. Thus the method dumps the entire content of HID from the by-group just having been processed into a data set named as OUT1 if ID=1, OUT2 if ID=2, and so on.
- The DATA step implied loop moves program control back to the top of the step where it encounters the hash declaration. The old table is wiped out and a fresh empty table is instantiated and redefined. Then either the next BY-group starts, and the process is repeated, or the last record in the file has already been read, and the step

stops after the SET statement hits the empty buffer.

CONCLUSION

It has been proven through testing and practical real-life application that direct-addressing methods can be great efficiency tools if/when used wisely. Before the advent of Version 9, the only way of implementing these methods in a SAS Data step was custom-coding them by hand.

The DATA step hash object provides an access to algorithms of the same type and hence with the same high-performance potential, but it does so via a pre-canned routine. It makes it unnecessary for a programmer to know the details, for great results—on par or even better than hand-coding, depending on the situation—can be achieved just by following syntax rules and learning which methods cause the hash object to produce coveted results. Thus, along with improving computer efficiency, the Data step hash objects may also make for better programming efficiency.

Finally, it should be noted that at this moment of the Version 9 history, the hash object and its methods have matured enough to come out of the experimental stage. To the extent of our testing, they do work as documented. From the programmer's viewpoint, some aspects that might need attention are:

- Parameter type matching in the case where a table is loaded from a data set. If the data set is named in the step, the attributes of hash entries should be available from its descriptor.
- The need to define the key variable additionally as a data element in order to extract the key from the table still looks a bit awkward.
- If the name of a dataset is provided to the constructor, a SAS programmer naturally expects its attributes to be available at the compile time. Currently, the compiler cannot see inside the parentheses of the hash declaration. Perhaps the ability to parse the content would be a useful feature to be addressed in the future. The word from SAS is that in 9.2, the compiler will be able to see at least something beyond the left parenthesis of hash h(), namely to recognize SAS data set options.

Putting it all aside, the advent of the hash object as the first dynamic DATA step structure is nothing short of a breakthrough. It arms DATA step (and now DS2) programmers with tools unthinkable before. Just to name a few:

- Create, at the Data step run-time, a memory-resident table keyed by practically any composite key.
- Search for keys in, retrieve entries from, add entries into, replace entries in, remove entries from, the table - all in practically O(1), i.e. constant, time.
- Create a hash iterator object for a table allowing to access the table entries sequentially.
- Delete both the hash table and its iterator at the run-time as needed.
- Wipe out the content of a hash table, without deleting the table itself.
- Make the table internally ordered automatically by a given key, as entries are added/replaced.
- Load the table from a SAS file via the DATASET parameter without explicit looping.
- Unload the table sequentially key by key using the iterator object declared for the table.
- Scroll through a hash table up or down, including starting with a given key.
- Create, at the run-time, a single (or multiple) binary search AVL tree, whose memory is allocated/released dynamically as the tree is grown or shrunk.
- Use the hash table as a quickly searchable lookup table to facilitate file-matching process without the need to sort and merge.
- Use the internal order of the hash table to rapidly sort large arrays of any type, including temporary arrays.
- Dump the content of a hash table to a SAS data file in one fell swoop using the .OUTPUT() method.
- Create the names of the files to be dumped depending on a SAS variable on the fly - the new functionality similar to that of the FILEVAR= option on the FILE statement.
- Use this functionality to "extrude" input through hash object(s) to split it in a number of SAS files with SAS-variable-governed names - if necessary, internally ordered.
- Use a hash table to store and search references to other hash tables in order to process them conditionally.
- Use hash table objects to organize true run-time Data step dynamic dictionaries.
- Use the hash object to aggregate data in a single-pass for both additive and non-additive statistics. Not only its speed and efficiency can dwarf both SUMMARY and SQL procedures, but there exist real-life situations where it is pretty much the only practical way the job can be done.

REFERENCES

1. D.Knuth. *The Art of Computer Programming*, **3**. The Art of Computer Programming
2. T.Standish. *Data Structures, Algorithms & Software Principles in C*. 0201528800: Data Structures, Algorithms and Software Principles ...
3. P.Dorfman. *Table Lookup by Direct Addressing: Key-Indexing, Bitmapping, Hashing*. SUGI 26, Long Beach, CA, 2001. SUGI 26: Table Look-Up by Direct Addressing: Key-Indexing ...
4. P.Dorfman, G.Snell. *Hashing Rehashed*. SUGI 27, Orlando, FL, 2002. SUGI 27: Hashing Rehashed
5. P.Dorfman, G.Snell. *Hashing: Generations.* SUGI 28, Seattle, WA, 2003. SUGI 28: Hashing: Generations
6. J.Secosky. *Getting Started with the DATA step Hash Object*, SGF 1, Orlando, FL, 2007. 271-2007: Getting Started with the DATA Step Hash Object
7. P.Dorfman, K.Vyverman. *DATA Step Hash Objects as Programming Tools*. SUGI 30, Philadelphia, 2005. 236-30: DATA Step Hash Objects as Programming Tools
8. P.Dorfman, L.Shajenko. *Data Step Hash Objects and How to Use Them,* NESUG 2006, Philadelphia, PA 2006. DATA Step Hash Objects and How to Use Them
9. P.Dorfman, L.Shajenko. *Crafting Your Own Index: Why, When, How*, SUGI 25, San Francisco, CA, 2006. Crafting Your Own Index: Why, When, How
10. P.Dorfman, D.Henderson. *Data Aggregation Using the SAS Hash Object*. SAS Global Forum, Dallas, TX, 2014. http://support.sas.com/resources/papers/proceedings15/2000-2015.pdf

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
4437 Summer Walk Ct.,
Jacksonville, FL 32258
(904) 260-6509
sashole@gmail.com