The SAS® Hash Object in Action

Paul Dorfman, Independent SAS Consultant, Jacksonville, FL

ABSTRACT

In SAS Version 9.1, the hash table - the very first object introduced via the DATA Step Component Interface in Version 9.0 - has finally become robust and syntactically stable. The philosophy and application style of the hash objects is quite different from any other structure ever used in the DATA step before. The most notable departure from the tradition is their run-time nature. Hash objects are instantiated, deleted, allocate memory, and get updated all at the run-time. Intuitively, it is clear that such traits should make for very inventive and flexible programming unseen in the DATA step of yore. Still better, subsequent SAS versions have added more hash methods, attributes, parameters, and functionality. This paper includes both hash propaedeutics and material intended for programmers already familiar with SAS hashigana at basic to advanced levels. A number of truly dynamic programming techniques utterly unthinkable before the advent of the canned hash objects in SAS are explored and explained using live SAS code samples. The content of the paper applies to Base SAS and any SAS product whose final result is assembled Base SAS code.

INTRODUCTION

Although *hashing* per se is, strictly speaking, merely one of direct-addressing techniques, using the word as a collective term has become quite common. Hopefully, it will be clear from the context in which sense the term is being used, however it will be used mostly in its strict meaning.

From the algorithmic standpoint, hashing is by no means a novel scheme, nor is it new in the SAS realm. In fact, a number of direct-addressing searching techniques have been successfully implemented using the Data step language and shown to be practically useful! This set of hand-coded techniques, complete with a rather painful delving into their guts, was presented at SUGI 26 and 27 [1, 2].

SAS Version 9 introduced an amazing array of new DATA step features. While most of them are rather incremental in nature, the introduction of the Data Step Component Interface (DSCI) stands out as a breakthrough. The two first objects made available via DSCI are *the hash object* and the related *hash iterator object*. These constructs can be used as a memory-resident DATA step dictionaries, fully dynamic in the sense that they can be created, used, and deleted during the DATA step execution, performing data storage and retrieval operations by a key practically in *O(1)* time. Their run-time nature allows for DATA step programming quite different from that of yore, and their *O(1)* key-search behavior results in exceptionally rapid and efficient performance.

Although this paper does contain a healthy dose of hash *propaedeutics*, its main thrust using the power of stating and solving real programming problems to demonstrate hashing techniques within and beyond basic *hashigana*, as it were.

1. HASH OBJECT PROPAEDEUTICS

What is the SAS hash object? It is a high-performance look-up table residing completely in the DATA step memory. The hash object is implemented via Data Step Component Interface (DSCI), meaning that it is not a part of the DATA step proper. Rather, you can picture it as a black-box device you can manipulate from inside the DATA step to ask it for lightning-quick data storage and retrieval services. For example, you can:

- give the hash object a key and tell it to use to ***add*** it along with associated data in the table or ***replace*** the data if the key is already there
- ***check*** whether the key is in the table, and if yes, find (retrieve) its data or remove the key with its data from the table
- ask the object to write its content to an ***output*** SAS data file independently of other DATA step actions
- make the hash object accumulate simple summary statistics for each key in the table instruct the hash object to have the table entries ***ordered*** by the key
- use *the hash iterator object* to access the ***next*** key in the table starting from the ***first*** or ***last*** key or from any given key value at all, i.e. to enumerate the entries
- do all of the above with a *hash object containing other hash objects* rather merely data stored in DATA step PDV variables.

"High-performance" means that not only all hash object operations are performed very rapidly, but that the speed of Add, Check, Find, and Remove operations (called *methods*) *does not depend on the number of keys in the table*. In other words, the time necessary to find if a given key is in the table is virtually the same for a table no matter whether it contains 100 or 1000,000 unique keys.

However, before being able to do all these wonderful things, you have to learn a few basic facts:

- 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 very simple to learn and use.
- You have to understand that all hash object operations – including the creation and deletion of the object itself – are performed at the DATA step run (execution), rather than compile time.
- However, what occurs during the compile time is very important to the hash object! You have to learn how to prepare the DATA step environment for working with the hash object, for without such preparation the object cannot even be created, let alone serve a useful purpose.

1.1. Declaring and Instantiating

Before a hash object can be used for any purpose, 2 things must happen: It has to be (1) declared and (2) instantiated. Depending on the circumstances, these things can be done one at a time or in a combination.

Example 1: Declaring a Hash and Creating Its Instance Separately

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

- This code runs error-free, hence the DATA step compiler accepts the syntax. Let us observe several essential facts pertaining to the code excerpt above:
- The hash object exists in the DATA step environment. The answer to the question "Can the hash object be used in SQL?" the answer is "no", at least for the time being.
- DECLARE is a SAS statement to declare a hash object. Alternatively, it can be abbreviated as DCL.
- HASH is a keyword. Its verbose alternative is ASSOCIATIVEARRAY. Note that if you decide to use it instead of HASH, you will have to use the same with the _NEW_ statement.
- HH is the name given to the hash object. It can be any *valid SAS name*.

Hence, using the aliases noted above, the two hash lines can be alternatively written as:

```
   declare associativearray myhash ;
   myhash = _new_ associativearray() ;
```

The declaration and instantiation of the hash object, however, can be combined in a single statement as shown below.

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.)

1.2. Keys and Data

From the user standpoint, the hash object is merely a key-indexed data file residing completely in memory. If you can picture a SAS data file, you have already pictured a SAS hash table. However, besides the fact that it is stored in memory rather than on disk, the hash table has another important distinctions from the 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

- Only the values belonging to the data portion can be retrieved and written to DATA step PDV "host" variables
- Hence, the hash object provides means to tell which columns are assigned as keys and which – as data. This is done using the *hash object methods* DefineKey and DefineData.

Let's imagine that we have 2 columns: 1 numeric variable KN and 1 character variable DC. How do we tell the hash object HH to designate variable KN as the key and variable DC – as the data portion, respectively? Example 3 below gives an answer.

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 that the *object-dot syntax* hh.DefineKey and hh.DefineData, used above to tell the object how to apportion the columns within the table, references the instance of HH created in the preceding DCL statement. Finally, DefineDone method is the instruction to finalize the creation of this object instance, in other words, to instantiate it. It is during the time of this method's execution when certain checks are performed to make sure the instance is good - in particular, that the rules of *parameter type matching* (discussed later on) are observed. Therefore, if something is awry, the SAS log error message will point to the line where the DefineDone method is coded.

If we omit or comment out the LENGTH statement, SAS will abend the step with an *execution-time* (i.e. not compile-time!) error message. This is because, firstly, all hash object statements and methods are executed at the run time. Secondly, this is because you cannot define a key or data column for the hash object if a column with the same name and data type -- called here and below a *host variable* -- is not already in the PDV. That is, *the DATA step compiler must see it first*. Of course, the LENGTH statement is not the only, and certainly not the most convenient, means to create host variables. We shall dwell on this in more detail later.

1.3. Adding Keys and Data

After keys and data column for a hash object have been defined and the object has been successfully initialized, it can be populated with key and data *values*. Due to the run-time nature of the hash object, it can be done in various ways. However, all of them use the Add or Replace hash object methods either 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 this example, the ADD method inserts the keys and data into the table one at a time. If you check the values of RC variable, you will find that they resolve to zeroes, meaning that the method has inserted the values successfully. Another method we are learning from this example is the 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. As we have noted before, only the data portion can be retrieved from a hash table. Column KN is passed to DEFINEDATA method because we want it to be written out; otherwise, only DN variable would be output. If we print the data file HH, we would see:

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

Besides being useful in its own right, OUTPUT method is an excellent diagnostic tool, as it can be employed to easily verify the content of the hash table by viewing the SAS data file to which its content was dumped.

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 our natural programming instincts 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 ;
```

1.4. Data Search and Retrieval

The real reason to have a hash table in the first place is that it allows for extremely quick searching and, if need be, data retrieval via a key. In the next example, the CHECK hash method is used to determine whether keys KN=2 and KN=5 are in the table. Then the FIND method is used to search for a key and, if the key is found, write its data portion counterpart to the corresponding PDV host variables.

Example 6: CHECK and FIND Methods

```
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 ;
```

This step writes the following lines to the SAS log:

```
KN=2 DC=  rc=0
KN=5 DC=  rc=-2147450842
KN=2 DC=data_2 rc=0
KN=5 DC=data_2 rc=-2147450842
```

What is going on here? Since KN=2 is in the table, the 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 data even when the key is found, and thus variable DC remains blank. This is because CHECK method is purposely designed to not overwrite 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 DC remains missing.

2. 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 search a memory-resident table for a key 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.

2.1. 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 “matching 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. (The "SAT" suffix is derived from the word "satellite" - a hint that this is a non-key variable.) We need to use file SMALL as a lookup table to pull S_SAT variable for each KEY having a match in file LARGE. This step shows one way to do it using the hash object:

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

```
data match (drop = rc) ;
   length key $9 s_sat 8 ;
   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 gives SAS the attributes of the key and data elements before the methods defining them could be called.
- The DECLARE hash statement names the hash table (HH).
- The DefineKey method describes the variable(s) to serve as a key into the table.
- The DefineData method is called if there is a non-key satellite information, in this case, S_SAT, to be loaded in the table.
- The DefineDone method is called to complete the initialization of the hash object, i.e. instantiate it.
- The 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 the satellite corresponding to a non-unique key will be used.*

- The 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 (keep = s_sat) ;
  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 ;
run ;
```

Here are some notable differences:

- Instead of the LENGTH statement, we can give the Define methods key and data attributes by reading the descriptor of SMALL and using the KEEP or DROP data set options to let the compiler see only the variables we need. Note that there is no need to keep KEY variable here because the compiler will have seen it anyway from the SET LARGE statement. One huge advantage of this method over the LENGTH statement comes to the fore when the hash table is defined with many key and data variables: Instead of hard coding the variable names, lengths, and types in the LENGTH statement, we have the compiler create the host variables in the PDV with all the correct attributes in one fell swoop. It saves both time and eliminates potential typos.
- DCL can be used as a shorthand for DECLARE.
- Keyword HASH can be used as an alias instead of ASSOCIATIVEARRAY. To the delight of those poor typists among us, when people speak, SAS' R&D listens!
- Instead of loading keys and satellites from SMALL one datum at a time, we can instruct the hash table constructor to load the table directly from the SAS data file SMALL by specifying the file in the object declaration via the tag parameter DATASET.
- The object parameter HASHEXP tells the table constructor to allocate 2**10=1024 AVL trees for the object. Compared with the default 2**8=256 trees, it fosters performance at the expense of a little bit more hash memory footprint. 2**20 is the max, so specifying more is pointless - SAS will cut it off at hashexp:20 anyway.
- Assigning return codes to a variable when the methods are called is not mandatory. Omitting the assignments shortens notation.

2.2. Parameter Type Matching

The LENGTH statement in the first version of the step or the attribute-extracting SET in the second one provide for what is called *parameter type matching*. When a method, such as FIND, is called, it presumes that a variable into which it can return a value matches the type and length FIND expects it to be.

It falls squarely upon the shoulders of the programmer to make sure parameter types do match. The LENGTH or SET statements above achieve the goal by giving the table constructor the names of existing Data step variables for the key (KEY, length $9) and satellite data (S_SAT, length 8).

Doing so simultaneously creates *Data step host variable* S_SAT, into which the FIND method (and others, as we will see later in the iterator section) automatically copies a value retrieved from the table in the case of a successful search.

2.3. 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, starting with 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].

2.4. Composite Keys And Multiple Satellites

In the 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.

2.5. 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 via DATASET and HASHEXP tags) 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 (    ) ;
```

3. HITER: HASH ITERATOR OBJECT

During both hash table load and look-up, the sole question we need to answer is whether the particular search key is in the table or not. The FIND and CHECK hash methods give the answer without any need for us to know what other keys may or may not be stored in the table. However, in a variety of situations we do need to know the keys and data currently resident in the table. How do we do that?

3.1 Enumerating Hash Entries

In hand-coded schemes, it is simple since we had full access to the guts of the table. However, hash object entries are not accessible as directly as array entries. To make them accessible, SAS provides the *hash iterator* object, *hiter*, which makes the hash table entries available in the form of a serial list. Let us consider a simple program that should make it all clear.

Example 9: 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 a declared hash object, in this case, "HH". After the iterator HI has been successfully *instantiated*, 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 entry with the smallest key. This is done by the method FIRST:

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

where HI is the name we have assigned to the iterator. A successful call to FIRST fetches the smallest key into the host variable K and the corresponding satellite - into the 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 > 0, and 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 be curious *why the key variable had to be also supplied to the DefineData method*? After all, each time the DO-loop iterates the iterator points to a new key and fetches a new key entry. The problem is that the host key variable K is updated only once, as a result of the HI.FIRST() or HI.LAST() method call. Calls to PREV and NEXT methods do not update the host key variable. However, a satellite hash variable does! So, if in the step above, it had not been passed to the DefineData method as an additional argument, only the key values 050 and 971 would have been printed.
The concept behind such behavior is that only data entries in the table have the legitimate right to be “projected” onto its Data step host variables, whilst the keys do not. It means that if you need the ability to retrieve a key from a hash table, you need to define it in the data portion of the table as well.

3.2. Array Sorting Via A Hash Iterator

The ability of the hash iterator object to rapidly retrieve hash table entries in order is an extremely powerful feature, which will surely find a lot of use in DATA step programming. The first iterator programming application that springs to mind immediately is using its key ordering capabilities to sort an array. The idea of how to do it is very simple:

1. Declare an *ordered* hash table keyed by a variable of the data type and length same as those of the array.
2. Declare a hash iterator.
3. Assign array items one at a time to the key and insert the key in the table.
4. Use the iterator to retrieve the keys one by one from the table and repopulate the array, now in order.

In Example 10, below, the problem is solved in a somewhat expanded form. Namely, we have two arrays. One of them, aK, is the "key" array, whose elements need to be put in order. The other, aS, is a "satellite" array, whose elements need to be permuted in sync with the corresponding elements of the key array.

Example 10: Using the Hash Iterator to Sort an Array

```
data _null_ ;
   array aK [9] (9 9 7 6 5 5 4 3 1) ;
   array aS [9] (1 2 3 4 5 6 7 8 9) ;

   dcl hash   HH (ordered:"A", multidata:"Y", hashexp:0) ;
   dcl hiter  HI ("HH") ;
   HH.defineKey  ("K") ;
   HH.defineData ("K", "S") ;
   HH.defineDone () ;

   do _n_ = lbound (aK) to hbound (aK) ;
      K = aK [_n_] ;
      S = aS [_n_] ;
      HH.add() ;
```

```
   end ;

   do _n_ = 1 by 1 while (HI.next() = 0) ;
      aK [_n_] = K ;
      aS [_n_] = S ;
   end ;

   put (aK[*]) (:1. +1) / (aS[*]) (:1. +1) ;
run ;
```

This step prints the following in the SAS log:

```
1  3  4  5  5  6  7  9  9
9  8  7  5  6  4  3  1  2
```

Let's make a few of notes:

- To handle duplicate keys, hash HH is declared with the tag parameter multidata:"Y". It means that all the keys and related data items are loaded into the table "as is", without any unduplication.
- Sorting is "stable" in the sense that for the duplicate key entries, it preserves the original relative order of the satellite array items. For example, for the duplicate key=5, the satellites in the output follow the same order as in the original input. This is equivalent to having EQUALS option on when using the SORT procedure for a similar purpose.
- HASHEXP=0 was chosen solely to make the following point: Since it means 2**0=1, i.e. a single bucket, *we have created a stand-alone AVL(Adelson-Volsky & Landis) binary tree in a Data step*, let it grow dynamically as it was being populated with keys and satellites, and then traversed it to eject the data in a predetermined key order. It means that with the advent of the hash object, there exists the ability in the DATA step to create and utilize a single binary search tree, with virtually zero programmatic effort.
- The code can be easily and in an obvious way expanded to have more than one array as a "key" and more than one array as a "satellite". Achieving the same functionality using CALL SORT* routines would require much more complex and contrived code.

4. BACK TO FUNDAMENTALS: DATA STEP COMPONENT INTERFACE (DSCI)

Now that we have gotten a taste of the new DATA step hash objects and some cool programming tricks they can be used to pull, let us consider them from a little bit more general viewpoint.
In Version 9, the hash table introduced the first *component object* accessible via a rather novel construct called the DATA Step Component Interface (DSCI). A component object is an abstract data entity consisting of two distinct characteristics: *Attributes and methods*. *Attributes* are data that the object can contain, and *methods* are operations the object can perform on its data. From the programming standpoint, an object is a kind of "black box" with certain defined properties, much like a SAS procedure. A DATA step programmer who wants an object to perform some operation on its content, does not have to program procedurally, but only needs to call whatever method is necessary for the task.

4.1. 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.

4.2. The 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. As they currently stand, the methods are as follows:

• 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.
• FIND_NEXT. After the FIND method is called, retrieve the next hash entry with the same duplicate key. If the method surfaces a non-zero return code, there are no more entries for this key.
• REF (new in 9.2). 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.

4.3. The 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 ;
```

4.4. The Operators

Currently, there is only one operator:

1._NEW_. Use it to create an instance of already declared component object. It 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. Please refer to the example below dealing with the hash of hashed (HOH) for the details of making use of the _NEW_ operator to solve a real-world programming task.

4.5. The 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 ;
```

Thus far, several component objects are accessible from a 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 (mainframe, anyone?) who have not exactly learned this kind of tongue in the kindergarten.

5. ON TO HAIRIER *HASHIGANA*

5.1 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 rapidly update its satellite?
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 ;
```

The inner Do-Until loop iterates over each BY-group with the same TRANSID value and summarizes AMT. 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. At the end of each ID BY-group, the iterator SI fetches TRANSID and SUM in the order descending by SUM, and top three retrieved entries are written to the output file. 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 new ones - defined. If the file has not run out of records, the outer Do-Until loop begins to process the next ID, and so on.

5.2. NWAY SUMMARY-Less Summarization: Pre - 9.2

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 does not need to carry all the baggage SUMMARY does. So, if the only purpose is, say, NWAY summarization, hashing 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.

5.3. Nway Summary-Less Summarization: Sas 9.2+

Note that above, the full potential of the hash method has not been achieved yet. Namely, 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 must be dumped into its host variable, have the next value added, and finally re-inserted into the table. In SAS 9.2, provision has been made to eliminate this source of inefficiency by introducing the ability of the hash object to accumulate statistics on the fly. Ray and Secosky in:

http://support.sas.com/rnd/base/datastep/dot/better-hashing-sas92.pdf

use the sample data above to show how it is done. In our notation (and with formatting adjusted to suit my personal coding preferences):

```
data hash_suminc_sum (keep = k1 k2 sum);
   dcl hash  h  (suminc:'num', hashexp:16) ;
   dcl hiter hi ('h') ;
   h.defineKey  ('k1', 'k2') ;
   h.defineDone () ;

   do until (eof);
      set input end = eof ;
      h.ref() ;
   end ;

   do rc = hi.first() by 0 while (rc = 0) ;
      h.sum (sum: sum) ;
      output ;
      rc = hi.next() ;
   end ;
   stop ;
run ;
```

Note how the hash declaration above contains parameter called SUMINC with analytical variable NUM specified as its argument. When a new record is read from INPUT, method REF(), newly-fangled in SAS 9.2, combines looking for a key in the hash table and handling aggregation depending on the search outcome. If (K1,K2) coming with the record is not found in the table, the new key pair is inserted, and the corresponding internal summary value is initialized to the value of NUM. Otherwise if the key is already in the table, no attempt to add the key is made; instead, NUM is added to the internal summary value for the existing key.

When all records from INPUT have been processed, the hash object automatically contains all unique keys from INPUT and the corresponding internal summary values, representing variable NUM separately aggregated for each (K1,K2) key pair. The DO loop that follows uses hash iterator object HI to cycle though each key present in the hash. For each key, method SUM tells to assign the internal accumulator value to host variable SUM named as an argument to parameter SUM (note the absence of quotes around the variable name) and writes the kept variables to output data set HASH_SUMINC_SUM.

The advantage of using SUMINC combined with REF and SUM methods is two-fold. First, there is no need to labor over the search-insert-accumulate logic. Second, it improves real- and cpu-time performance by about 12-15 percent.

5.4. Splitting a SAS File Dynamically via Output Method

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 variables. Thus, at any point at run-time, we can use the .OUTPUT() method to dump the contents of an entire hash table into a SAS data set, whose very name is formed using the SAS variable we need, 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_') ;
   hid.definedata ('transid', 'amt') ;
   hid.definedone ( ) ;
```

```
   do _n_ = 1 by 1 until ( last.id ) ;
      set sample ;
      by id ;
      hid.add() ;
   end ;

   hid.output (dataset: 'OUT' || put (id, best.-l)) ;
run ;
```

Variable _N_ used as the hash key makes the key for a given BY-group unique for each row, and each pair of (TRANSID, AMT) will be inserted in the table for the group. 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 how the DATA step works, but with feet not yet sufficiently wet with hashigana, the step above must look like a heresy. Indeed, how in the world is it possible to produce a SAS data set, let alone many, in a DATA _Null_ step? It is possible because, with the hash object, the output is handled completely by and inside the object when the .OUTPUT() method is called, the Data step merely serving as a shell and providing parameter values to the object constructor. This is how the step works:

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 next BY-group one record at a time and uses its variables and _N_ to populate the hash table. Thus, when the DoW-loop is finished, the hash table for this BY-group is fully populated. Now control moves to the HID.OUTPUT() method. Its DATASET: parameter is passed the current ID value, which is concatenated with the character literal OUT. The method executes writing all the variables defined within the DefineKey() method from the hash table to the file with the name corresponding to the current ID. The Data step implied loop moves program control 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. 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.

Note that completely destroying and recreating the hash object for each new BY-group is not the most efficient practice. And in SAS 9.2 and beyond, it is not necessary and should not be done. Instead, the following code should be used:

```
data _null_ ;
   if _n_ = 1 then do ;
      dcl hash hid (ordered: 'a') ;
      hid.definekey  ('_n_') ;
      hid.definedata ('transid', 'amt') ;
      hid.definedone ( ) ;
   end ;
   hid.clear() ;
   do _n_ = 1 by 1 until ( last.id ) ;
      set sample ;
      by id ;
      hid.add() ;
   end ;

   hid.output (dataset: 'OUT' || put (id, best.-l)) ;
run ;
```

The lines of code added to the Example 14 are shown above in red.

6. ON TO EVEN HAIRIER *HASHIGANA*

6.1. Peek Ahead: Hashes of Hashes

In the Q&A part after the presentation at SUGI 29, the author was asked from the audience whether it was possible to create a hash table containing references to other hash tables in its entries. Having forgotten the adage "Judge not rashly", I emphatically answered “No!” - and got it all wrong! Shortly after the conference, one of the world’s elite SAS programmers, Richard DeVenezia, demonstrated in a post to SAS-L that such construct is in fact possible, and, moreover, it can be quite useful practically. For examples of such usage on his web site, inquiring minds can point their browsers to:

http://www.devenezia.com/downloads/sas/samples/hash-6.sas

Here we will discuss the technique a little closer to *ab ovo*. Let us imagine an input file as in the example 9 above, but with the notable difference that *it is not pre-grouped by ID.* Suppose it looks as follows:

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

Now the same file-splitting problem has to be solved *in a single Data step, without sorting or grouping the file beforehand* in any way, shape, or form. Namely, we need to output a SAS data set OUT1 containing all the records with ID=1, OUT2 – having all the records with ID=2 and so on, just as we did before. How can we do that?
When the file was grouped by ID, it was easy because the hash table could be populated from a whole single BY-group, dumped out, destroyed, and re-created empty before the next BY-group commenced processing. However, in the case at hand, we cannot follow the same path since pre-grouping is disallowed.

Apparently, we should find a way to somehow keep separate hash tables for all discrete ID values, and, as we go through the file, populate each table *according to the ID value* in the current record, and finally dump the table into *corresponding output files* once end of file is reached. Two big questions on the path to the solution, are:

1. How to tell SAS to create an aggregate collection of hash tables keyed by variable ID?
2. How to address each of these tables programmatically using variable ID as a key?

Apparently, the method of declaring and instantiating a hash object all at once, i.e.

```
dcl hash hh (ordered: 'a') ;
< key/data definitions >
  hh.definedone () ;
```

which has been serving us splendidly so far, can no longer be used, for now we need to make HH *a sort of a variable* rather than *a sort of a literal* and be able to reference it accordingly. Luckily, the combined declaration above can be split into two phases:

```
dcl hash hh () ;
hh = _new_  hash (ordered: 'a') ;
```

This way, the DCL statement only declares the object, whilst the _NEW_ method creates a new instance of it every time it is executed at the run time. This, of course, means that the following excerpt:

```
dcl hash hh () ;
do i = 1 to 3 ;
   hh = _new_  hash (ordered: 'a') ;
```

```
end ;
```

creates three instances of the object HH, that is, three separate hash tables. It leads us to the question #2 above, or, in other words, to the question: *How to tell these tables apart?* The first, crude, attempt, to arrive at a plausible answer would be to try displaying the values of HH with the PUT statement:

```
26  data _null_ ;
27     dcl hash hh () ;
28     do i = 1 to 3 ;
29        hh = _new_ hash (ordered: 'a') ;
30        put hh= ;
ERROR: Invalid operation for object type.
31     end ;
32  run ;
```

Obviously, even though HH *looks* like a ordinary Data step variable, it certainly is not. This is further confirmed by an attempt to store it in an array:

```
33  data _null_ ;
34     array ahh [3] ;
35     dcl hash hh () ;
36     do i = 1 to 3 ;
37        hh = _new_ hash (ordered: 'a') ;
38        ahh [i] = hh ;
ERROR: Object of type hash cannot be converted to scalar.
39     end ;
40  run ;
```

So, we cannot store the collection of "values" HH assumes anywhere in a regular Data step structure. Then, where can we store it? In another hash table, of course! In a hash table, "data" can mean numeric or character Data step variables, *but it also can mean an object.* Getting back to the current problem, the new "hash of hashes" table, let us call it HOH, say, will contain the hashes pertaining to different ID values as its data portion. The only other component we need to render HOH fully operative is a key. Since we need to identify different hash tables by ID, this is the key variable we are looking for. Finally, in order to go through the hash table of hashes and select them for output one by one, we need to give the HOH table an iterator, which below will be called HIH. Now it is possible to put it all together:

Example 15: Splitting an *Unsorted* SAS File a Hash of Hashes and the OUTPUT Method

```
data _null_ ;
   dcl hash  hoh  (ordered: 'a') ;
   dcl hiter hih  ('hoh'       ) ;
   hoh.definekey  ('id'        ) ;
   hoh.definedata ('id', 'hh'  ) ;
   hoh.definedone () ;
   dcl hash hh () ;
   do _n_ = 1 by 1 until ( eof ) ;
      set sample end = eof ;
      if hoh.find () ne 0 then do ;
         hh = _new_ hash (ordered: 'a') ;
         hh.definekey  ('_n_') ;
         hh.definedata ('id','transid', 'amt') ;
         hh.definedone () ;
         hoh.replace () ;
      end ;
      hh.replace() ;
   end ;
   do rc = hih.next () by 0 while ( rc = 0 ) ;
      hh.output (dataset: 'out'|| put (id, best.-L))  ;
      rc = hih.next() ;
   end ;
   stop ;
run ;
```

As in the case of the sorted input, the program reports in the SAS log thus:

```
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.
```

Let us delve into the inner mechanics of HOH: How does it all work in concert?

1.The hash of hashes table HOH is declared, keyed by ID and instantiated using the usual “combined” method.
2. A hash object HH, whose future instances are intended to hold partial ID-related data from the input file, is declared, *but not yet instantiated*.
3. The next input record is read.
4. .FIND() method searches HOH table using the current ID as a key. If it does not find an HH hash object with this key, it has not been instantiated yet. Hence, it is now instantiated and stored in HOH by means of the HOH.REPLACE() method. Otherwise, an existing hash instance is copied from HOH into its ‘host variable’ HH to be reused.
5. The values from the record are inserted via HH.REPLACE() method into the hash table whose instance HH currently holds. Once again, _N_ is used as part of the composite key into HH hashes to discriminate between duplicates by ID and TRANSID.
6. If not end of file, the flow returns to line #3 and the next record is read .
7. Otherwise the iterator HIH (belonging to HOH table) is used to loop through the table one ID at a time.
8. Each time, HIH.NEXT() method extracts a new instance of HH from HOH, then the corresponding table is dumped into a data set with the name formed using the current value of ID.
9. The step is terminated.

Although the described mechanism of HOH operation may seem intimidating at once, in actuality it is not. Simply remember that instances of a hash object can be stored in a hash table (itself an object) as if they were data values, and they can be retrieved from the table when necessary to tell SAS on which one to operate currently. Another term for ‘hash’, ‘associative array’ is not at all arbitrary. Just as you can store a collection of related items of the same type in an array, you can store them in a hash - only the storage and retrieval mechanisms are different, and the hash can store a wider variety of types. Thinking through the control flow of the sample program above may be a quick way to bring the (at first, always vague) idea of how hashes of hashes operate in cohesion.

6.2. Multi-Level Hashes Of Hashes

The example above should serve as a good introduction into hash of hashes from the didactic point of view. However, it is still, well, introductory. It is possible to use this capability on a much wider scale and to a much greater benefit. Also, the depth of hash-of-hash storage can exceed that of a single level. But how can it be utilized practically? Let us consider the following real-world problem presented to SAS-L as a question by Kenneth Karan. Assume that we have a data set:

```
data a ;
   input sta: $2. gen: $1. mon: $3. Ssn: 9. ;
cards ;
NY        M       Jan    123456789
NY        M       Feb    123456789
NY        M       Mar    123456789
NY        M       Jan    987654321
NY        M       Feb    987654321
FL        M       Mar    987654321
NY        F       Jan    234567890
NY        F       Feb    234567890
FL        F       Jan    345678901
;
run ;
```

Now imagine that we have to produce a data set printed as follows:

```
Sta    Gen    Mon    _TYPE_    Count
```

```
-----------------------------------
                          0        4
              Feb         1        3
              Jan         1        4
              Mar         1        2
        F                 2        2
        M                 2        2
        F     Feb         3        1
        F     Jan         3        2
        M     Feb         3        2
        M     Jan         3        2
        M     Mar         3        2
FL                        4        2
NY                        4        3
FL            Jan         5        1
FL            Mar         5        1
NY            Feb         5        3
NY            Jan         5        3
NY            Mar         5        1
FL      F                 6        1
FL      M                 6        1
NY      F                 6        1
NY      M                 6        2
FL      F     Jan         7        1
FL      M     Mar         7        1
NY      F     Feb         7        1
NY      F     Jan         7        1
NY      M     Feb         7        2
NY      M     Jan         7        2
```

In other words, we need to produce output similar to what PROC SUMMARY would create without the NWAY option (i.e. for all possible interactions of the CLASS variable values), only the COUNT column would contain the number of distinct SSNs per each categorical keys' combination, rather than the number of observations (_FREQ_) per each combination. Of course, it can be easily accomplished by a series of sorts combined with appropriate BY-processing or by SQL explicitly listing the categorical combinations in the GROUP clause (SQL will still sort behind-the-scenes). *But is it possible to attain the goal via a single pass through the input data?* The answer is "yes"; and the Data step hash object is what affords us such a capability.

Example 16: Using Multi-Level Hash of Hashes to Create a Single-Pass Equivalent of Count Distinct Output Fitted to the PROC SUMMARY non-NWAY Template

```
%let blank_class = sta gen mon ;
%let comma_class = %sysfunc (tranwrd (&blank_class, %str( ), %str(,))) ;
%let items_class = %sysfunc (countw (&blank_class)) ;

data model ;
   array _cc_ $ 32 _cc_1-_cc_&items_class ;
   do over _cc_ ;
      _cc_ = scan ("&blank_class", _i_) ;
   end ;
run ;
proc summary data = model ;
   class &blank_class ;
   output out = types (keep = _cc_: _type_) ;
run ;
data count_distinct (keep = &blank_class _type_ count) ;
   dcl hash  hhh() ;
   hhh.definekey  ('_type_') ;
   hhh.definedata ('hh','hi') ;
   hhh.definedone () ;
   dcl hash  hh() ;
   dcl hiter hi   ;
   dcl hash  h()  ;
```

```
   do z = 0 by 0 until (z) ;
      set types end = z nobs = n ;
      array _cc_ _cc_: ;
      hh = _new_ hash (ordered: 'a') ;
      do over _cc_ ;
         if _type_ = 0 then _cc_ = '_n_' ;
         if missing (_cc_) then continue ;
         hh.definekey (_cc_) ;
         hh.definedata(_cc_) ;
      end ;
      hi = _new_ hiter ('hh') ;
      hh.definedata ('_type_','count','h') ;
      hh.definedone () ;
      hhh.add() ;
   end ;
   do z = 0 by 0 until (z) ;
      set test end = z ;
      pib_ssn = put (ssn, pib4.) ;
      do _type_ = 0 to n - 1 ;
         hhh.find() ;
         if hh.find() ne 0 then do ;
            count = 0 ;
            h = _new_ hash () ;
            h.definekey ('pib_ssn') ;
            h.definedone() ;
         end ;
         if h.check() ne 0 then do ;
            count ++ 1 ;
            h.add() ;
         end ;
         hh.replace() ;
      end ;
   end ;
   do _type_ = 0 to n - 1 ;
      call missing (&comma_class) ;
      hhh.find() ;
      do _iorc_ = hi.next() by 0 while (_iorc_ = 0) ;
         output ;
         _iorc_ = hi.next() ;
      end ;
   end ;
   stop ;
run ;
```

Let's try to reverse-engineer the inner mechanics of this code:

1. SSN is converted into its PIB4. image just to cut hash table memory usage by about half.
2. H-tables contain only 4-byte SSN images. There are as many H-tables per _TYPE_ as there are unique _TYPE_ keys; in other words, for each new value of STA, a separate instance of H-table is created, and it will contain only SSN values attributed to that key. The same is true for all other CLASS key combinations.
3. HH-tables are keyed by CLASS key combinations, each key pointing to _TYPE_, count, its own H-table and its iterator object as the data. This way, when a CLASS key of any _TYPE_ comes with the input data, the program knows which H-table, containing its own SSN values, to search.
4. When we need to output the H-table, we then know which particular iterator to use.
5. HHH-table is keyed by _TYPE_, so in this case, it contains 8 HH-tables (in turn, containing H-tables), enabling us to go from one instance of HH-table to another in a simple DO-loop.
6. In the end, _TYPE_ is used as a key to go over each instance of HHH-table and spit out the content of each HH-table to the data set COUNT_DISTINCT.
7. Note that _N_ is used as a single-value (_N_=1) key into the _TYPE_=0 table to give the table a dummy key because without a key, a hash table cannot be defined.

Readers willing to master this rather unusual technique are strongly encouraged to think thoroughly through the details of the code, using the Data step debugger if necessary.

6.3. Searching for Closest Neighbors: Setcur Method to Rescue

Let us consider a seemingly simple problem posted to SAS-L by Kevin Xu.
We have two files. File FIX contains a number of "fixed" integers, say, 3, 6, 8, 11, 12, 15, and so on, stored in variable FIX. The other file, NUM, contains either integer or fractional values stored in variable NUM, say, 2.2, 3, 4.4, 5, 6.6, 7, 8.8, 9, 11.11, 12.12, 14.9, 15.01, and so forth. For each NUM from file NUM, we need to find the closest FIX from file FIX.

At the first glance, the task is elementary: for each NUM go through all FIX values and find the difference, then select FIX, for which the difference is the smallest. However, it would work well for relatively small input. For example, assuming sample input with 1000 records in FIX and 1 million rows in NUM:

```
data fix ;
   do _n_ = 1 to 1000 ;
      fix = .1 * ceil (ranuni (1) * 1e7) ;
      output ;
   end ;
run ;

data num ;
   do num = 1 to 1e6 ;
      output ;
   end ;
run ;
```

we would have to perform 1E3*1E6=1 billion comparisons. Apparently, it is not very efficient. A more efficient approach would be to have FIX sorted. Then for each NUM, we would have to inspect only half of the FIX values on the average. However, the solution would still essentially run in N_FIX*N_NUM time. A big improvement then would be to use the binary search. While this is indeed doable - in fact, Liang Xie from SAS-L has shown adroit DATA step code to do just that – it is quite tricky to get it work correctly.

Let us recall that in a hash object, we can easily store keys in order. Suppose we have stored the FIX values 1, 3, 4, 6, 7, 9 there using FIX as a key. Now we need to find FIX closest to NUM=3. Using CHECK() method, we will find immediately that NUM=3 has an exact match in the table, therefore FIX=3 is its closest counterpart we are looking for. But what if we are looking for FIX closest to NUM=5.7, for which an exact match among FIX does not exist? Then we can insert the value of NUM in the hash table. Because it is ordered, 5.7 will end up being lodged between 4 and 6, so now the content of the table will be:

1 3 4 **5.7** 6 7 9

Now all we have to do is to find which FIX – 4 of 6 - is closer to 5.7 and of course decide on 6. Then we remove 5.7 from the table and proceed down the NUM list in the same manner. Problem solved!

Or is it? The caveat with the hash object in SAS 9.1 is that in order to get to 5.7, we will have to start the iterator at the either endpoint of the table, comparing each key to 5.7 until we bump into it. Only after that we could use PREV() and NEXT() methods to see what surrounds 5.7 in the table. And by inspecting – serially! – about half of the table on the average for each NUM in question, we lose all efficiency gained by inserting 5.7 into the table in *O(1)* time.

This is where the new iterator method SETCUR now available in SAS 9.2 comes to rescue. Much like the binary search (which what fundamentally the underlying algorithm is), it allows starting the iterator at the specified key value. Now both inserting of 5.7 into the table and locating it there occur in *O(1)* time, the corresponding DATA step code being:

Example 17: Using 9.2 SETCUR method to start the iterator at a specified key value

```
data closest_hsh (keep = num closest) ;
   if _n_ = 1 then do ;
      if 0 then set fix ;
      dcl hash  h  (dataset: "fix", ordered: "A") ;
      dcl hiter hi ("h") ;
```

```
      h.definekey  ("fix") ;
      h.definedone () ;
   end ;

   set num ;

   if h.check(key:num) = 0 then closest = num ;
   else do ;
      h.add(key:num, data:num) ;
      if hi.setcur(key:num) = 0 then if hi.prev() = 0 then _pfix = fix ;
      if hi.setcur(key:num) = 0 then if hi.next() = 0 then _nfix = fix ;
      if      nmiss (_nfix) then _inx = 1 ;
      else if nmiss (_pfix) then _inx = 2 ;
      else _inx = 1 + (num - _pfix > _nfix - num) ;

      closest = choosen (_inx, _pfix, _nfix) ;

      h.remove (key:num) ;
   end ;
run ;
```

How does the double-*O(1)* SAS 9.2 hash solution scale up? Liang Xie has suggested using PROC FASTCLUS to achieve the same goal. For the sake of comparison, the hash, FASTCLUS, and sorted array approaches, running against the 1E3*1E6 input above on a garden-variety Windows desktop, result in the following real times:

```
Sorted array 42 seconds
FASTCLUS     10 seconds
Hash          6 seconds
```

6.4. Getting Socially Fuzzy with Hashigana

The hashigana's *O(1)* run behavior naturally lends itself to problems calling for repeated searches of massive (albeit memory-resident) collections of keys. One class of such problems is related to fuzzy match (probabilistic record linkage), where repeated look-ups on "fuzzy" keys can be sometimes used to convert the problem from *O(n*m)* to *O(n+m)* run behavior. The example below should make these, admittedly rather fuzzy remarks clearer.

In January 2009, Don Henderson on SAS-L posted a problem together with a quick-and-simple SQL solution, asking for thoughts on improving its performance. In Don's own words, "The client wants to match two files using the Social Security number and wants to detect as possible matches cases where position by position 7 of the nine digits match. The problem is actually illustrated better – as it often happens - by Don's sample data and "brute force" solution:

```
data one;
 input ssn $9.;
datalines;
123456789
987654321
121212121
343434343
;
data two;
 input ssn $9.;
datalines;
123456789
987645321
121212121
232323232
;
proc sql;
 create table matches as
 select a.ssn as ssn_one,
        b.ssn as ssn_two,
        case when (a.ssn ne b.ssn) then 'Not Exact'
             else 'Exact'
```

```
        end as MatchResults
 from one a, two b
 where sum(substr(a.ssn,1,1)=substr(b.ssn,1,1),
           substr(a.ssn,2,1)=substr(b.ssn,2,1),
           substr(a.ssn,3,1)=substr(b.ssn,3,1),
           substr(a.ssn,4,1)=substr(b.ssn,4,1),
           substr(a.ssn,5,1)=substr(b.ssn,5,1),
           substr(a.ssn,6,1)=substr(b.ssn,6,1),
           substr(a.ssn,7,1)=substr(b.ssn,7,1),
           substr(a.ssn,8,1)=substr(b.ssn,8,1),
           substr(a.ssn,9,1)=substr(b.ssn,9,1)
          ) ge 7;
quit;
```

Specifically, Don wondered if regular expressions ("regexen") could be made use of to improve performance. A few good suggestions were then offered by a number of people which could cut the run time of the query above close to 50 per cent. Unfortunately, none managed to address the main issue underlying poor performance, namely, the fact that the solution's run-time scales as *O(n*m)*, where N and M are the number of records in ONE and TWO, respectively. This is because the solution compares every 7-out-of-9 digit combination from each SSN from ONE to every such positional combination for each SSN from TWO. For such two files containing, say, mere 100,000 records each, 10 billion keys (and each possible 7-from-9 digit combination within them) have to be compared, which already takes a painfully long time. But the real-world problem was much worse, since it had to deal with files filled with millions of records and executed on a mainframe with tight control over machine resources.

To break the O(n*m) scaling behavior, an approach is needed that would do away with the necessity to compare each SSN from ONE to each SSN from two. Luckily, one of the authors *(P.D.)* was privileged to have worked with probabilistic record linkage guru Sigurd Hermansen on problems of similar nature, and Sig's post in the same thread gave away a hint.

The key here is the matching pattern constraint in the specifications. To wit, there are 9!/7!/(9-7)!=36 7-digit positional patterns in a 9-digit string, and we have to match only pattern #1 of SSN1 to pattern #1 of SSN2, pattern #2 of SSN1 to pattern #2 of SSN2,..., and so on till pattern #36. *Since thus no cross-pattern matching is required, separate patterns can be matched independently*. Hence the algorithm can be devised as follows:

- DO PATTERN = 1 to 36
- Pass through each file
- Extract 7-digit pattern #N from SSN1 and SSN1
- Make it the key, having SSN1 and SSN2 as satellites
- Perform many-to-many match of result sets
- Output matches and append them to final data set
- END

At the first glance, this scheme appears pretty idiotic, because effectively it means exploding each file 36 times, then matching by (pattern number * pattern). However, it appears less so after the realization that the number of passes through each file is limited to only 36 regardless of the file size, and after the explosion has occurred, matching on each pattern becomes time-linear, provided that the selected key-search method behaves as *O(1).*

Hence, the mechanism of turning the O(m*n) problem in an O(m+n) problem stands to benefit the most from hashigana treatment. To avoid re-reading the files from disk, they are buffered on the first file passes into a "large Enough" array, which is subsequebtly re-read 35 times. The actual matching by a pattern is done by storing the pattern ID from file DSN1 and its discriminating enumerator N as a hash key and SSN1 - as a satellite, then searching the table for each respective pattern from SSN2. The discriminator is necessary because a pattern can correspond to more than one SSN1. (As we already know, in SAS 9.2, multi-hash can be used to harvest different SSN1 for the same key pattern, but since the code was tested under 9.1.3, the old simple trick of storing and harvesting hash dupes is used - which ought to be apparent from the code.) It is important that after each pass, the table is explicitly deleted, else memory overload can (and does) occur. Again, in 9.2 it is not necessary - instead .clear() method should be used. However, although .delete() is an expensive method, because it is done here only 36 times, it has practically no adverse effect.

```
%let size = 1e5 ;

data dsn1 dsn2 ;
```

```
   length ssn $ 9 ;
   do _n_ = 1 to &size ;
      ssn = put (ceil (ranuni(1) * 1e9), z9.) ; output dsn1 ;
      ssn = put (ceil (ranuni(1) * 1e9), z9.) ; output dsn2 ;
   end ;
run ;

data matches (keep = ssn: exact) ;
   array ss [2, 5000000] $ 9 _temporary_ ;
   do i = 1 to 8 ;
      do j = i + 1 to 9 ;
         dcl hash   h (hashexp: 16) ;
         h.definekey  ('id', 'n') ;
         h.definedata ('ssn1') ;
         h.definedone () ;
         do p = 1 to n1 ;
            if i = 1 and j = 2 then do ;
               set dsn1 (rename = (ssn = ssn1)) nobs = n1 ;
               ss [1, p] = ssn1 ;
            end ;
            else ssn1 = ss [1, p] ;
            id = ssn1 ;
            substr (id, i, 1) = "*" ;
            substr (id, j, 1) = "*" ;
            do n = 1 by 1 until (h.check() ne 0) ;
            end ;
            h.add() ;
         end ;
         do p = 1 to n2 ;
            if i = 1 and j = 2 then do ;
               set dsn2 (rename = (ssn = ssn2)) nobs = n2 ;
               ss [2, p] = ssn2 ;
            end ;
            else ssn2 = ss [2, p] ;
            id = ssn2 ;
            substr (id, i, 1) = "*" ;
            substr (id, j, 1) = "*" ;
            do n = 1 by 1 ;
               if h.find() ne 0 then leave ;
               if ssn1 = ssn2 then Exact = "Y" ;
               else                Exact = "N" ;
               output ;
            end ;
         end ;
         h.delete() ;
         idn ++ 1 ;
         put "Progress: pattern # " idn 2.-R " processed." ;
      end ;
   end ;
   stop ;
run ;
```

This 1e5*1e5 match runs in under 10 seconds on a garden variety AIX box and outputs 35,938 records, taking up 250M of RAM. What is critical, though, is how it scales up. Here are some log figures:

Size, obs	Real time, sec	Memory usage, Mb	Output obs
1e5*1e5	10	250	35,938
5e5*5e5	65	280	897,950
1e6*1e6	170	320	3,582,144

```
2e6*2e6   |   440          |     400          | 14,331,078
----------+----------------+------------------+-----------
5e6*5e6   |  1620          |     640          | 89,596,633
```

Even the 5m by 5m match runs under 27 minutes, which includes producing the enormous output (in the real environment likely to be infinitesimally smaller). The real time scales almost linearly as the sizes grow, "almost" probably being due to the O(log(N)), rather than O(1), nature of the AVL trees underlying SAS hash. Still, the comparison math is nowhere near the Cartesian explosion. Running 5m by 5m match using the scheme above means 36 passes over side 1, during which 36*5e6=1.8*5e7 SSNs are considered, plus, roughly speaking, 5e6*log2(5e6)=3.5e7 SSN pairs are considered searching the table. Running the same via a Cartesian explosion means 1 pass through file DSN1 and 5m passes through file DSN2, during which 2.5e13 SSN pairs are considered - the difference of good 5 orders of magnitude.

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 are mature enough to have fully 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.
• 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 production DATA step hash objects as the first dynamic DATA step structure is nothing short of a long-time breakthrough. This novel structure allows DATA step programmers to do things 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 either direction 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 such dictionaries to speed up the calculation of simple statistics instead of MEANS/SUMMARY procedures, when the latter choke on a high cardinality of their classification variables.

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

CONTACT INFORMATION

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