Paper 4-28

HASHING: GENERATIONS

Paul M. Dorfman, Independent Consultant, Jacksonville, FL
Gregg Snell, Data Savant Consulting, Shawnee, KS

ABSTRACT

Searching is one of the most, if not the most, important and frequently performed data processing operations. The SAS® System supports this assertion with a roster of built-in searching techniques, such as merges, joins, formats, indexes, and specific operators and functions.

In the light of such diversity, it is all the more surprising that until recently, no built-in SAS methods based on fastest searching schemes – memory-resident hash-based algorithms - had been available. This gap was partially covered by SAS users crazy enough to hand-code their own hash-based algorithms in the SAS Language.

However, Version 9 changes everything. Not only has it made direct addressing available in the form of a built-in hash table (associative array) which can be searched much faster than any other SAS lookup structure; it has also introduced the first dynamic, memory-resident Data step structure.

In this paper, we will try to get a taste of this new exciting SAS tool and envision cool things it will allow us to do. We will also see that it does not make hand-coded hashing methods instantly obsolete and compare the I (hand-coded) and II (built-in) SAS hashing generations from the standpoints of functionality, programming and computer efficiency.

INTRODUCTION

Table lookup or searching is, practically speaking, the most common data processing operation. In this respect, the closest that comes to mind is sorting, but then almost always the ultimate goal of sorting is to organize a search. SAS addresses this situation like in no other software package by providing the programmer with an incredibly rich collection of built-in searching techniques.

Purposely limiting ourselves, *for the time being*, by Version 8, we might, for example, think of:

- Conditional (IF-THEN-ELSE) logic and the case (SELECT) structure
- Search operators, such as IN, LIKE, etc.
- The MERGE statement
- SQL with all its bells and whistles
- Formats and informats
- SAS indexes
- String-searching functions and regular expressions
- Internal in-memory trees implicitly used by class-type procedures

These instruments embrace a variety of searching situations and employ a number of various lookup algorithms. Some of them are designed to operate in memory, others - on disk. However, all of them (except for the SQXJHSH available through SQL), have a number of properties in common:

1. They are based on comparing a search key to one or more keys in the table.
2. In this respect, they (and hence their efficiency) only differ in the number of key comparisons needed to discover whether a search key is among the lookup keys or not.
3. If the number of lookup keys increases N times, the number of comparisons necessary to locate or reject a search key increases, in the best case scenario, log2(N) times.

A principally different searching strategy is employed by the SQXJHSH method. With SQXJHSH, the number of key comparisons per act of searching - and thus the speed of the latter – either no longer depends on the number of lookup keys or grows with N much slower than log2(N). Looking a whit forward, such behavior is typical for *direct-addressing*, as opposed to *key-comparison*, lookup methods. In Version 8 and before, SQXJHSH has been the only SAS-supplied direct-addressing search method, and since it is an SQL-only, no such method has been available in the Data step.

This shortcoming is in part compensated by the SAS Language with enough tools to help programmers implement their own searching methods, if need be, for instance:

- SAS arrays
- Direct SAS file access via the POINT= option
- SASFILE statement allowing to pre-buffer an entire data set in memory

These tools and/or structures, together with the rest of the language, are sufficient for implementing just about any searching algorithm. Of course, techniques based on these tools are not ready-to-go routines, and they have to be custom-coded. But by the same token, they are more flexible and thus often result in routines searching much faster and using fewer resources.

HASHING SEMANTICS

In the title of the paper, the term “hashing” was used collectively to denote the whole group of memory-resident searching methods not primarily based on comparison between keys, but on direct addressing. Although strictly speaking, hashing per se is just one of direct-addressing techniques, using it as a collective term has become common. Hopefully, it will be clear from the context in which sense the term is used. Mostly, it will be used in its strict meaning.

In fact, someone has even had enough time and inspiration to implement a number of direct-addressing searching techniques in the Data step and show that they work well enough in order to be useful! This set of hand-coded direct-addressing routines, together with a rather painful delving into their guts, was presented at SUGI 26 and 27 [1, 2]. For the lack of a better description, let us call them, and anything that can be derived from them, **Generation I**.

Generation I has a number of drawbacks – which we will go on to discuss later – inevitable for almost any more or less complex, performance oriented routine coded in a very high-level language, such as the SAS data step. But by the same token, it has a number of advantages, primarily: The code is available, and so it can be tweaked, changed to accommodate different specifications, retuned, etc.

However, the most important consequence of Generation I activity was that it might have impact on the advent of **Generation II** that arrived shortly after the game has been joined by SAS. And lo and behold, in Version 9, we have a present in the form of an object called *associative array*, or - yes, you guessed right! - *hash*. This object can be used as a canned box to search data via a direct-addressing algorithm implemented internally. It is a real breakthrough in more ways than one.

HOW TO READ THIS PAPER

This paper covers a *lot* of ground and it would be extremely helpful to read the referenced papers beforehand, especially [1], [2] and [4]. A dictionary wouldn’t hurt either ;-)

DIRECT-ADDRESSING: PROPAEDEUTICS

To make the discussion more concrete, consider a common task of matching two data files by a common key. Suppose that an unsorted SAS data file SMALL contains N_SMALL records with a numeric integer variable KEY and a satellite variable S_SAT. Another unsorted file called LARGE, with N_LARGE records also has the variable KEY and a satellite L_SAT. Assume that 1) LARGE is not sorted and, for whatever reason, cannot be sorted and 2) there is enough memory to hold the entire SMALL, or at least KEY and a numeric pointer to its records.

Given these conditions: *What is the most efficient way to subset* LARGE *based on the values of* KEY *in* SMALL *to produce a file* MATCH?

SAS offers a number of ready-to-go tools based on in-memory table lookup. Just to mention a couple:

1. Compile unduplicated keys from SMALL into a format using CNTLIN= option, and search it for each KEY read from LARGE.
2. Load the keys from SMALL into a sorted array and use a hand-coded binary or interpolation search to look for each key from LARGE.

Why look for something else? The efficiency and speed of such methods are principally limited, because they are comparison-based. It is known that for an arbitrarily distributed lookup keys, no comparison-based method can do better than the binary search. The latter, to either find search key among N lookup keys or reject it, must make no less than the average of log2(N)+1 comparisons. For N=1,000,000 it costs 20 comparisons plus computations and logic. Sometimes it is expressed by saying that the *binary search runs in O(log(N))* time.

Removing key comparisons as the primary basis of searching could thus be highly beneficial. But is it possible to search for a key without comparing it with the keys in a lookup table at least once? A rather paradoxical answer to this question is "yes". It is given by a radically different searching philosophy called direct addressing. And *direct addressing* finds its pure expression in *key-indexed search*.

KEY-INDEXING

The idea is simple. Assume that all keys are 1-digit numbers from 0 to 9, and SMALL has only 9 records:

OBS	1	2	3	4	5	6	7	8	9
KEY	2	3	5	2	7	9	5	7	3
S_SAT	1	2	3	0	4	5	6	9	7

Let us create a temporary array HKEY with *one node (location, address) allocated for each possible key value*. By default, SAS will initialize all the buckets to missing values. HKEY can be thought of as the following table in memory:

H	0	1	2	3	4	5	6	7	8	9
HKEY	.	.	.	.	.	.	.	.	.	.

Now, for each key from SMALL, let us look at the array location H, *whose index is equal to the value of the KEY*, i.e. at HKEY(KEY). Since there is a separate bucket for each possible key value, we are always guaranteed to find the address H=KEY. If HKEY(KEY) is missing, let us move the satellite S_SAT to H=KEY. Repeating the procedure for the rest of the keys yields

H	0	1	2	3	4	5	6	7	8	9
HKEY	.	.	1	2	.	3	.	4	.	5

It is a *key-indexed table*, and it comprises two types of *entries: empty and occupied*. Note that *duplicate keys are deleted automatically* as the table is loaded. If SMALL has no satellites or they are of no interest, an occupied entry can be marked by moving 1 into the node.

Given a search KEY, how do we look it up? All we have to do is to examine the node *whose index is equal to KEY*. If the corresponding location is missing, the key is not in the table. If it is occupied, the search has been successful, and the node contains either 1 or the related satellite. For example, if KEY=1, the search fails since the address 1 is empty. If KEY=7, we have to look at the node 07. It is occupied; therefore, the key is found, and the node returns the satellite value HKEY(7)=4.

The utter simplicity of key-indexing translates into an equally simple DATA step implementation. Suppose, for example, that our keys are integers ranging from –4E6 to +4E6. The range thus naturally defines the bounds of the array HKEY representing the key-indexed table.

```
data match ;
   array hkey (-4000000 : 4000000) _temporary_ ;
   do until ( eof1 ) ;
      set small end = eof1 ;
      if missing (hkey(key)) then hkey(key) = s_sat ;
   end ;
   do until ( eof2 ) ;
      set large end = eof2 ;
      s_sat = hkey(key) ;
      if s_sat > . then output ;
   end ;
   stop ;
run ;
```

The first Do-until loop loads the key-indexed table from SMALL; in the second, the table is searched for each key coming from LARGE, and matches are output.

Thus, irrespective of a hit or miss, the key-indexed search works via a single array reference, without comparing the search key to any lookup keys. Of course, no lookup method can be simpler and/or run faster. (For example, see the performance comparison in [1] showing in part that key-indexing outperforms presorted MERGE by a factor of 5:1.)

But the most fundamental property of the key-indexed search principally distinguishing it from any comparison-based search is that both *"inserting" a key and searching for a key is independent of the number of lookup keys*. Another manner of expressing this fact is to say that both insertions and searches in the key-indexed scheme occur *in constant time, or as O(1)*.

The question arises: If key-indexing is the fastest insertion/searching algorithm, why can we not use it at all times and forget about everything else? The answer is: Because it is based on the assumption that the lookup keys are either integers falling in a limited range or can be inexpensively converted to such integers. Our test keys take on only as many as 8,000,001 distinct values, so sufficient array space can be allocated using about 64 MB of memory. Having 80 MB of memory, one can get away with 7-digit keys. However, to deal with 9-digit SSN, an array with 1 billion elements would be needed, which is hardly practical even with the modern cheap memories. A 16-digit credit card number as a key would make straight key-indexing a nightmare.

However, the method is so promising that it is worth expanding. To do so, we ought to find a way to keep its memory usage at bay. Both the speed of key-indexing and its limitations rest upon the following facts:

- The table is directly addressed by keys themselves.
- The entire set of possible key values is addressable.
- No comparisons between the search key and lookup keys are made.

Thus, the applicability of the pure direct addressing can be, in principle, widened in two ways:

1. *Keep the key range fully addressable*, but address bits instead of bytes.
2. Drop the restrictions that (a) no two keys shall reside in one node, and (b) no comparisons between the search key and lookup keys are made.

The first approach results in a technique called *bitmapping*. The second path leads to a more versatile hybrid searching method known as *hashing*.

BITMAPPING

If the satellite information in SMALL is of no interest, the key-indexed table only serves to indicate whether a memory node, whose index corresponds to the key value, is empty or occupied. Occupied nodes can be populated with 1, and the empty ones - initialized to 0. So, if a node whose number is the value of KEY contains 1, KEY is in the table; otherwise, it is absent. But in order to tell 0 from 1 we need a single bit. Yet our key-indexing scheme uses the full memory length of a numeric array item, i.e. whole 8 bytes to store a Boolean value. If we could make efficient use of bits in such a setting, the memory usage could be potentially reduced 64 times!

Ways of doing this and executable code are discussed in detail in [1].
Just like a key-indexed table, a bitmap is a purely direct-addressed table, and hence runs as O(1) in both insertion and searching modes. Bitmapping shines in the niche that can be defined as "no-matter-how-many-short-keys". Bitmap is a champion when we only need to rapidly find out if the record with a given key should be selected, and if memory resources are sufficient for indexing the entire key range into the bits of a temporary numeric array.

HASHING: GENERATION I

Compared to key-indexing, bitmapping merely expands the workable universe of keys about 53+ times. Hashing methods approach the problem quite differently: They eliminate the requirement of a separate slot for each possible key *and* allow some amount of comparisons between the search key and lookup keys in the table *at the final stage of searching*.

A simple example is perhaps a good way of making the idea clear. Assume that SMALL contains just ten 3-digit keys and related satellites (deliberately chosen below as a straight enumeration):

```
data small ;
   input key s_sat ;
cards ;
185  00
971  01
400  02
260  03
922  04
970  05
543  06
```

```
532  07
050  08
067  09
 ;
run ;
```

To use key-indexing on such a key, we would allocate a [0:999] table and map each key to the node whose number is the key value. However, out of 1000 table nodes, only 10 will be occupied. The rest will idle playing the role of placeholders. That is, 99 per cent of the array memory will be simply wasted! The crucial question is thus: Can we get away with a table *only somewhat larger* than the number of lookup keys, and keep the benefits of direct addressing?

To answer it, let us pick some number HSIZE greater than the number of keys N_SMALL in SMALL, for instance, 17, and allocate an array sized as HKEY(0:17). Let us agree to call the array a *hash table*, HSIZE - *hash table size*, and the ratio N_SMALL/HSIZE - *load factor*. Thus, the load factor shows the number of lookup keys relative to the total number of nodes in the hash table, i.e. the *sparsity* of the table. In our example, the load factor is 0.588, or the table is about 41 percent sparse.

Now envisage some rapidly-computing function H(KEY) taking KEY as an argument and returning an address H into HKEY, unique to each key supplied, thus mapping any lookup key to its own node:

```
KEY → H(KEY) → Unique Address in [1:HSIZE]
```

That would be a *perfect hash function*. If it existed, we would only have to plug it into the key-indexing code. Although such functions *are* possible, they are quite difficult to discover, and once one is found, it can only be used for the same set of keys: Adding just an extra key will ruin everything.

A less rigid method can be obtained if we let H(KEY) *map two or more distinct keys to the same location* in HKEY:

```
KEY → H(KEY) → Some Address in [1:HSIZE]
```

If more than one key is sent to the same address, a *collision* occurs, and we must do something to tell the keys hashing to the same address apart. In other words, we have to employ a *collision resolution policy* to distinguish between keys hashing to the same location in the process of insertion and searching.

Thus, we arrive at the *core concept behind hashing*. If the hash function H(KEY) maps only a few keys per *hash bucket* H spreading the lookup keys evenly among the buckets, we can do the following:

1. Use H(KEY) to *hash* KEY to some address H.
2. If the content of H is empty, KEY is not in the table, since no key has ever hashed to H.
3. If the address is occupied, search among the few keys that have hashed to the bucket H.

Within each address, the search can be organized in a number of ways. With a large number of buckets and only a few keys per bucket, straight sequential search is the simplest and fastest. In the framework of Generation I hand-coded hashing, it is practically the only reasonable way to go, although it could conceivably be binary search or something else. Most importantly, once the only bucket H where KEY can reside has been found, the keys in this bucket are searched via a *comparison-based algorithm*.

Thus, hashing is a typical *hybrid algorithm*: It combines direct addressing with a method based on comparisons between keys in its final key-seeking stage. The average number of keys mapping to any hash node equals N_SMALL/HSIZE, i.e. the load factor. If the hash table is not full and the keys are spread uniformly, the average number of key comparisons required to find or reject a key is *less than 1*. Also, searching for a key should be the faster, the sparser the table is. So, to make a good practical use of a hash table, we ought to:

1. Choose a proper hash function H(KEY).
2. Find an efficient way of resolving collisions.

A *good hash function* apparently must:

1. Be rapidly computable.
2. Distribute lookup keys uniformly over the table.
3. Map them in the [0:HSIZE-1] range.

Among a number of techniques conforming to these requirements, the simplest is the *division method*:

```
H = MOD (KEY, HSIZE) ;
```

The number theory tells us that if HSIZE is a *prime number* far from a power of 2, MOD tends to spread the keys uniformly. Let us see how this would work for our sample set of 10 keys. If we choose the "target" load factor as 0.625 and divide it into the number of keys, we obtain 16. The first prime number greater or equal to 16 is 17, so let us select HSIZE=17. (The actual load factor is now 10/17 = 0.588.) So, we could allocate the table as

```
ARRAY HKEY (0:17) _TEMPORARY_ ;
```

To get a hash address, KEY is divided by HSIZE=17, and the remainder H is computed. H points to the H-th slot in the table where KEY must be inserted. Repeating this operation for all test keys yields the following:

```
H
---------------
00    .   .   .
01  970   .   .
02  971   .   .
03    .   .   .
04  922   .   .
05  260 532   .
06    .   .   .
07    .   .   .
08    .   .   .
09  400   .   .
10    .   .   .
11    .   .   .
12    .   .   .
13    .   .   .
14    .   .   .
15  185   .   .
16  543 050 067
17    .   .   .
---------------
```

The keys 970, 971, 922, 400, and 185 all map to their slots in HKEY one-to-one. The keys 260 and 532 produce a *single collision* at the address 05, and the keys 543, 050, and 067 result in a *double collision* in the node 16. If this table is to be stored in memory and searched, the collisions at the locations 05 and 16 have to be *resolved*.

No matter how good a hash function is, some keys are very likely to hash to the same addresses, so we will have to resolve collisions. This is another point at which hashing radically deviates from pure direct addressing, where keys are not stored in the table itself. *With hashing, the keys themselves reside in the table*, because they may need to be compared to a search key. *Collision resolution policies* differ in the ways the colliding keys are stored, linked as being mapped to the same hash address, and traversed.

COLLISION RESOLUTION: SEPARATE CHAINING

One way of resolving collisions suggests itself naturally once we cast a rapid glance at the distribution of our 10 keys among the addresses of the hash table already shown in the previous section. Keys mapped to each occupied address form visible "chains". If an address is uncontested, the chain consists of a single key or no keys at all if it is empty. Making use of such chains to resolve collisions is thus called *separate chaining*. There are two ways the chains of keys can be utilized in terms of the SAS DATA step.

First, the keys comprising the chains could be stored *outside the table* by placing them in the occurrences of a two-dimensional array. However, this may well cause very poor memory utilization. Suppose we have 100,000 keys in SMALL and map all but one of them so that no address contains more than, 2 to 4 keys, and a single unlucky address at which a whole 10 keys collide. In this case, we will be forced to create a 2-dimensional array sized as [0:HSIZE , 0:9] to resolve the collisions. Even with the load factor 1, it would need 10 times the memory the keys themselves would use.

In Generation I, efficient separate chaining cannot by organized head-on because arrays are allocated at compile time. In other words, we cannot create a dynamic structure attached to each hash bucket H (such as a link list or tree) that would grow each time we need to add a key by the amount of memory needed to accommodate the key and its satellite(s). One reason we mention something not possible to do is that it is exactly what Generation II makes possible to achieve. Therefore, we will return to the separate chaining later in the Generation II section.

Meanwhile in Generation I, the problem is solved along the lines of the Russian proverb "Gol na vydumki khitra" ("They who lack resources get ingenious"). By changing the way of memory allocation from sequential to *linked*, we can arrive at

an extremely elegant collision resolution policy called *coalesced list chaining*, which is both very fast and reasonably memory-efficient. The technique is described at length in [1, 2]; an interested reader is more than welcome to explore its intricacies.

In this paper, we will discuss the simplest collision resolution policy called *open addressing with linear probing*. Its painstaking description can be found in [1, 2]. Below, we will take a look at it, sufficient to see how Generation I may approach the collision resolution problem *in principle*.

COLLISION RESOLUTION: LINEAR PROBING

The main idea behind open addressing is as follows. Lookup keys are stored in the hash table itself. Suppose we have KEY to be loaded. First, hash it:

```
H = MOD(KEY, HSIZE) ;
```

If H points to an empty slot, place KEY there, else we have a collision. In this case, compare KEY with the key already in H. If the keys are equal, the current key is a duplicate, so we can either discard it or consider it just another key, depending on specifications. If duplicate keys are to be eliminated, get the next key; otherwise find a different slot for KEY. Step up the table one or more times one node at a time by adding 1 to H. The maximum address to which KEY can hash is HSIZE-1. So, if H=HSIZE, we are out of range [0:HSIZE-1]. To get back there, set H=0 and continue this wrap-around cycle until having found an empty node and insert the colliding key there. In this fashion, the table is "probed" linearly using a fixed *probe decrement* C=1; hence the name.

Since we have HSIZE+1 nodes in the table, but can only address HSIZE nodes from 0 to HSIZE-1, at least 1 location in the table will always be empty, thus preventing the wrap-around process from iterating infinitely. Inserting our ten sample keys into the table in this manner results in the following:

H	HKEY[H]	HSAT[H]
00	**050**	88
01	970	55
02	971	11
03	**067**	99
04	922	44
05	260	33
06	**532**	77
07	.	.
08	.	.
09	400	22
10	.	.
11	.	.
12	.	.
13	.	.
14	.	.
15	185	00
16	543	66
17	.	.

The keys 185, 971, 400, 260, 922, 970, and 543 all hash without collisions to the locations 15, 02, 09, 05, 04, 01, and 16 respectively. They get inserted into the table together with their satellites without contention. However, the next key, 532, hashes to H=05 already occupied by 260. Incrementing H by 1 results in H+1=06. Since the node 06 is empty, we place the key 532 and its satellite at H=6. The next key, 050, hashes to H=16, but it is already taken up by 543. Incrementing H by 1 results in 17, which is beyond the hashing range [0:16]. So, we go back to the beginning of the table by setting H=00. Since this node is empty, we place 050 at H=00. The last key, 067, also claims the spotlight at H=16. Just like with 050, we go up by one address, have to wrap to H=00, which now is occupied by 050. Stepping up the table, we find an empty node at H=03, which is where KEY=067 and its satellite 99 get inserted.

Now the above of loading the table suggests the way of searching it. If a search KEY hashes to an empty node, it is not in the table, period. If the node is not empty, it may or may not be in the table, so we step up the table until bumping into an empty slot. What we do before such a slot is encountered depends on whether the duplicate keys and satellites from SMALL are to be extracted or not. If they are, go all the way to the missing slot picking up all lookup keys equal to the search key (and their satellites) along the way. If not, stop at the very first instance when a matching key is found. If we have reached the end of the cluster without encountering a matching key, it is not in the table.

From these simple examples, it should be clear how linear probing can reduce the number of probes a comparison-based search requires. In the worst case scenario for the table above, linear probing examines 5 keys until it either finds or rejects a search key. However, the number of comparisons per average hit/miss search will be close to 1. Moreover, it will remain the same if we have 1 million lookup keys and about 1.7 million nodes, i.e. as long as the load factor remains the same. Therefore, although hashing does allow some key comparisons in the final stage of the searching, lookups and insertions occur in O(1) time. (Compare with the binary search with 4 comparisons for N=10, and 20 comparisons for N=1,000,000.)

The simplicity of the linear probing leads to simple code:

```
%let nodupes =  1 ; *0 if dupes to be pulled ;

data match (keep = key s_sat l_sat) ;
   retain nodupes &nodupes.. ;
   array hkey (0 : &hsize) _temporary_ ;
   array hsat (0 : &hsize) _temporary_ ;
   do until ( eof1 ) ;
      set small end = eof1 ;
      do h = mod (key, &hsize) by +1 ;
         if h = &hsize then h = 0 ;
         if hkey(h) = key and nodupes then leave ;
         if hkey(h) = . then do ;
            hkey(h) = key ;
            hsat(h) = s_sat ;
            leave ;
         end ;
      end ;
   end ;
   do until ( eof2 ) ;
      set large end = eof2 ;
      do h = mod (key, &hsize) by +1
            until ( hkey(h) = . ) ;
         if h = &hsize then h = 0 ;
         if hkey(h) = key then do ;
            s_sat = hsat(h) ;
            output ;
            if nodupes then leave ;
         end ;
      end ;
   end ;
   stop ;
run ;
```

If the NODUP parameter is set to false, all the satellites corresponding to duplicate keys in SMALL are extracted, else only the first one in order is picked.

The main advantage of this scheme is its profound simplicity. And if the table is sparse enough, it performs very well. As a rule of thumb, the linear probing will do the hashing job just right with load factors 0.5 or less, i.e. if the table is more than half sparse. As it gets fuller, performance deteriorates because of the *primary clustering*. Trying to find a place for a colliding key, we fill out the very first empty location we come across. Thus, the groups of adjacent occupied addresses tend to aggregate, forming clusters of keys, which in turn can bridge together forming bigger clusters. For an example, look at the addresses [00:06] in our sample table above. So, if the table is rather full, we may eventually have to travel practically over the entire table before finding an empty location, thus degenerating hashing into a sequential search.

The problem can be alleviated by stepping through the table more than one node at a time, i.e. making the probe increment greater than 1. Aided by a couple of tricks from the number theory, it leads to a method called *open addressing with double hashing* that eliminates primary clustering, so the same speed can be achieved in a fuller table, which results in better memory utilization. For a detailed discussion of the double hashing and its SAS implementation, see [1].

NON-INTEGER KEYS

Test results [1] show that hashing performs admirably by any account regardless of the collision resolution policy. However, above examples implied that the keys were numeric and natural. How do we hash them if they are not?

Let us note first that because in its final stage, hashing is comparison based, it renders the nature of the keys non-critical. Both hashes and traversals are used merely to minimize the number of comparisons necessary to carry out a search, yet the final hit-or-miss decision (if the hash address is not empty) is made by comparing some lookup keys in the table to the search key.

Therefore, we only have to figure out how to devise the hash function if the key is not a non-negative integer. It must basically satisfy simple rules:

- Hashing should involve as much of the key information as possible.
- It must produce an integer in [0:HSIZE-1] range.

Let us consider a number of distinct practical situations. If the lookup keys are *fractional signed SAS numbers*, we can simply rescale the key by multiplying by a suitable integer constant and/or adding a constant, then applying the MOD function as usual. If the lookup keys are *digit strings (character variables whose values consist of digits only)* is a simple matter of applying a standard numeric informat. For example, for a 16-digit string, MOD(INPUT(KEY,16.),HSIZE) will work just fine. Short digit strings (1 to 8 bytes long) can be hashed faster as a character variable in general (see below).

ARBITRARY CHARACTER KEYS

Numerous techniques have been developed to hash arbitrary character keys [2, 3, 4]. Almost all of them are based on breaking a character key apart and then involving the individual bytes into a sort of computation resulting in an integer in the range [0:HSIZE-1]. However, in SAS sub-stringing and concatenation are rather slow. Instead, we can let call for an integer binary informat to do the job:

```
H =  MOD (INPUT(KEY,PIBw.), HSIZE) ;
```

In a single shot, this obviates sub-stringing and converts KEY in to a (large) number that can be divided by HSIZE. The method has its limitations (see [1] for details). However, it works admirably for character keys more or less discriminated by their W first characters. If the leading W bytes tend to be blank, the LEFT function helps mitigate the situation.

Let us see, for example, how our linear probing would look if, say, we had KEY as a 9-byte character variable. First, the hash array HKEY must now be defined with the appropriate data type as $9, while the satellite array stays intact:

```
array hkey (0 : &hsize) $9 _temporary_ ;
array hsat (0 : &hsize)    _temporary_ ;
```

Secondly, testing for an empty node will be testing for a blank. With these amendments, the code for inserting a key into the table transforms into

```
do h = mod (input(key,pib6.),&hsize) by +1 ;
  if h = &hsize then h = 0 ;
  if hkey(h) = key and nodupes then leave ;
  if hkey(h) = '' then do ;
    hkey(h) = key ;
    hsat(h) = s_sat ;
    leave ;
  end ;
end ;
```

And the code for searching for a key similarly becomes

```
do h = mod (input(key,pib6.),&hsize) by +1
     until ( hkey(h) = '') ;
  if h = &hsize then h = 0 ;
  if hkey(h) = key then do ;
    s_sat = hsat(h) ;
    output ;
    if nodupes then leave ;
  end ;
end ;
```

The properties of the algorithm guarantee that both the insertion and search run in O(1), i.e. constant, time, as long as the load factor stays near 0.5. In other words, if it takes T time units to find/reject a search key for N=10 and HSIZE=17, it will take T time units to do it for N=100000 and HSIZE about 170000. The same is true for all direct-addressing methods. Additionally, if the hash function is good, and the table is sparse, both insertions and searches appear practically instant. This has a very important consequence, as it effectively gives the hash table an alternative meaning of an associative array indexed by a key of arbitrary type.

COMPOSITE KEYS

Quite often, a key is *composite*, i.e. it may consist of an arbitrary mixture of numeric and/or character variables. A natural inclination is to concatenate the components and hash the result. Principally, there is nothing wrong about it; however, there are two pitfalls. First, concatenation is slow. Secondly, the key components may concatenate into an integer lying beyond SAS integer precision. Usually, the programmer is left alone with the imagination and knowledge of the data. Principally, hashing a combination of random bytes selected from all the keys is a reasonable way to go. As the number of the components of a composite key grows, all Generation I hashing schemes quickly become more complicated. Aside from devising a decent hash function, we have to create a separate parallel array for all component keys and compare all the keys to their array counterparts in relevant conditionals. That is not to say that it cannot be done or should not be done if need be. However, we will see that in Generation II, the business of hashing data identified by composite keys becomes just a breeze.

HASHING: GENERATION II

SAS Version 9 has introduced many new features into the Data step language. Most of them, expanding existing functionality and/or improving its performance, are rather incremental. However, one novel feature stands out as a breakthrough: Associative arrays of hashes.

GENERATION II PROPAEDEUTICS

Perhaps the best way to get a fast taste of this mighty addition to the Data step family is to see how it can help solve our sample matching problem. Let us assume, for an extra kick, that KEY is a character variable of length 9:

```
data match ( drop = rc ) ;
   length key $9 s_sat 8 ;

   declare AssociativeArray hh () ;

   rc = hh.DefineKey  ( 'key'   ) ;
   rc = hh.DefineData ( 's_sat' ) ;
   rc = 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?

- LENGTH statement gives SAS the attributes of the key and data elements before the methods defining them could be called.
- DECLARE AssociativeArray 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 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:

```
data match ;
   set small point = _n_ ; *get key/data attributes ;
  *set small (obs = 1) ;   *this will work, too! ;
  *set small (obs = 0) ;   *but this will not! ;
  *if 0 then set small ;   *and neither will this! ;

   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 notable differences:

- Instead of the LENGTH statement, we can give the Define methods key and data attributes by reading a record from SMALL. Somewhat surprisingly, is not sufficient just to read a descriptor; there must be a record read at run-time.
- 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 of us typo-impaired, it means: When people speak, SAS 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 hash declaration.
- The HASHEXP named parameter 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

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.

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 get 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?

In Generation I, duplicate-key entries can be controlled programmatically by twisting the guts of the hash code. To achieve the desired effect in Generation II, 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 ;
```

Note that at this point, Generation II hashing does not provide a mechanism of storing and/or handling duplicate keys with different satellites in one and the same hash table. This difficulty can be principally circumvented, if need be, by discriminating the primary key by creating a secondary key from the satellite, thus making the entire composite key unique. All the more, it is further aided by the ease with which Generation II hash tables can store and manipulate composite keys.

COMPOSITE KEYS AND MULTIPLE SATELLITES

In Generation I, creating a composite hash key can be a breeze or a pain, depending on the type, range, and length of the component keys [1]. But in any case, the programmer needs to know the data beforehand and often demonstrate a good deal of ingenuity.

Generation II 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 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 execution time.

Perhaps it is a good idea to avoid hard-coding data values in a Data step, 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 what Master Ian Whitlock calls “wall paper”, and separates code from data.

HASH PARAMETERS AS EXPRESSIONS

The two steps above may have already given a hash-hungry reader enough to start munching mind-boggling 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:

```
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 be necessarily 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 (    ) ;
```

HASH ITERATOR

During both hash table load and lookup the sole question we need to answer is whether the particular search key is in the table or not. The FIND method gives 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 already stored in the table at the moment. How do we do that?

In Generation I it is simple since we had full access to the guts of the table: Merrily run through all table nodes sequentially and extract the keys and satellites

corresponding to all occupied nodes. For example, for the linearly probed table, all it takes is this:

```
do h = 0 to &hsize - 1 ;
   if missing ( hkey (h) ) then continue ;
   key = hkey (h) ;
   s_sat = hsat (h) ;
   < ... further processing ... >
end ;
```

In Generation II one answer could be to maintain an auxiliary array and store a key there every time it is loaded in the hash table. Then run through the keys in the array serially, using the FIND method to dump the table one key at a time.

Actually, it might make sense if the goal is to retrieve the data from the table in which the they have been entered. However, often times it is much more beneficial to be able to dump the table in a sorted key order. For this purpose, SAS provides the *hash iterator* object.

Let us consider a simple program that should make it all clear:

```
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_ ;
   set sample point = _n_ ;

   dcl hash  hh ( dataset: 'sample',
                  hashexp: 8       ,
                  ordered: 1       ) ;
   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 non-zero option ORDERED. Without such arrangement, the subsequent iterator object declaration

```
dcl hiter hi ( 'hh' ) ;
```

would fail. Note that the hash object symbol name must be passed to the iterator 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 a sorted order by key.

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 FIRST or 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.

At this point, it is not clear whether it is a design feature or something that will be addressed when Generation II hashing will emerge from the experimental stage and go real production. Either way, it can always be circumvented by the trick shown above.

ITERATOR PROGRAMMING EXAMPLE: ARRAY SORTING

The ability of a hash iterator 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 another object. The easiest and most apparent prey is a SAS array. Note, though, that since a Generation II hash table cannot hold duplicate keys, the arrays sorted below using the first array A as a key and the second array B - as its satellite - will be effectively unduplicated. That is, after each sorting loop, the array A will be sorted from lbound(A) to lboound(A)+n_unique (in this case, n_unique=86507):

```
data _null_ ;
   array a (-100000 : 100000) _temporary_ ;
   array b (-100000 : 100000) _temporary_ ;
   do j = lbound (a) to hbound (a) ;
      a (j) = ceil ( ranuni (1) * 1e5 ) ;
      b (j) = j ;
   end ;
   length ka 8 sb 8 ;
   declare hash  hh (hashexp: 0, ordered: 1 ) ;
   declare hiter hi ( 'hh' ) ;
   hh.DefineKey     ( 'ka'         ) ;
   hh.DefineData    ( 'ka' , 'sb' ) ;
   hh.DefineDone    () ;
   do j = lbound(a) to hbound(a) ;
      ka = a (j) ;
      if hh.check () = 0 then continue ;
      sb = b (j) ;
      n_unique ++ 1 ;
      hh.add () ;
   end ;
 * sort ascending ;
```

```
   rc = hi.first () ;
   do j = lbound (a) by 1 while ( rc = 0 ) ;
      a (j) = ka ;
      b (j) = sb ;
      rc = hi.next () ;
   end ;
 * sort descending ;
   rc = hi.last() ;
   do j = lbound(a) by 1 while ( rc = 0 ) ;
      a (j) = ka ;
      b (j) = sb ;
      rc = hi.prev() ;
   end ;
   stop ;
run ;
```

Note that HASHEXP=0 was chosen. 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.

Just to give an idea about this hash table performance in some absolute figures, this entire step runs in about 1.15 seconds on a desktop 933 MHz computer under XP Pro. The time is pretty deceiving, since 85 percent of it is spent inserting the data in the tree. The process of sorting 200,001 entries itself takes only scant 0.078 seconds either direction. Increasing HASHEXP to 16 reduces the table insertion time by about 0.3 seconds, while the time of dumping the table in order remains the same.

DATA STEP COMPONENT INTERFACE

Now that we have a taste of the Generation II hashing, let us consider it from a little bit more general viewpoint.

In Version 9, the hash table (associative array) introduces the first *component object* accessible via a rather novel thingy called 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 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 run-time, while an object accessible through DSCI - can. A Data step programmer who wants an object to perform some operation on its data, does not have to program it procedurally, but only to call an appropriate method.

THE HASH 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 Generation I direct-addressed tables satisfy this definition *in the strict sense*. We will see that the Generation II hash object 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 data 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.

The hash table methods are the functions it can perform, namely:

- 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 with new ones.
- Find. Search for the key. If it is found, extract the satellite(s) from the table and update the host Data step variables.
- Check. Search for the key. If it is found, just return RC=0, and do nothing more.

DATA STEP 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 ()
```

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.

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

GENERATION II: A PEEK UNDER THE HOOD

We have just seen the tip of the Generation II 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 2) resolving collisions within each bucket. Discussing collision resolution schemes, we had to reject 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 Generation II is 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 binary tree. If there were, for simplicity, only 4 buckets, the scheme might roughly look like this:

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 behavior regardless of the distribution of the key values.

The number of hash buckets is controlled by the HASHEXP parameter we have used above. The number of buckets allocated by the hash table constructor is 2**HASHEXP. So, if HASHEXP=8, HZISE=256 buckets will be allocated, or if HASHEXP=16, HSIZE=65536. As of the moment, it is the maximum. Any HASHSIZE specified over 16 is truncated to 16.

Let us assume HASHEXP=16 and try to see how, given a KEY, this structure facilitates hashing. First, a mysterious internal hash function maps the key, whether is it simple or composite, to some bucket. The tree in the bucket is searched for KEY. If it is not there, the key and its satellite data are inserted in the tree. If KEY is there, it is either discarded when ADD is called, or its satellite data are updated when REPLACE is called.

Like in Generation I, how fast all this occurs depends on the speed of search. Suppose that we have N=2**20, i.e. about 1 million keys. With HSIZE=2**16, there will be on the average 2**4 = 16 keys hashing to one bucket. Since N > HSIZE, the table is overloaded, i.e. its load factor is greater than 1. However, binary searching the 16 keys in the AVL tree requires only about 5 keys comparisons. If we had 10 million keys, it would require about 7 comparisons, which practically makes almost no difference.

Thus the Generation II hash table behaves as O(log(N/HSIZE)). While it is not *exactly* O(1), it can be considered such for all practical intents and purposes,as long as N/HSIZE is not way over 100. Thus, by choosing HASHEXP judiciously, it is thus possible to tweak the hash table performance to some degree and depending on the purpose.

For example, if the table is used primarily for high-performance matching, it may be a good idea to specify the maximum HASHEXP=16, even if some buckets end up unused. From our preliminary testing, we have not been able to notice any memory usage penalty exacted by going to the max, all the more that as of this writing, the Data step does not seem to report memory used by an object called through the DSCI. At least, experiments with intentionally large hash tables show

that the memory usage reported in the log is definitely much smaller than the hash table must have occupied, although it was evident from performance that the table is completely memory-resident, and the software otherwise has no problem handling it. However, with several thousand keys at hand there is little reason to go over HASHEXP=10, anyway. Also, if the sole purpose of using the table is to eject the data in a key order using a hash iterator, even a single bucket at HASHEXP=0 can do just fine, as we saw earlier with the array sorting example. On the other hand, if there is no need for iterator processing, it is better to leave the table completely iterator-free by not specifying a non-zero ORDERED option. Maintaining an iterator over a hash table obviously requires certain overhead.

HASH TABLE AS A DYNAMIC DATA STEP STRUCTURE

Generation II hash tables represent 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 either by pre-processing data, 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.

What we cannot do dynamically (at least, for now) is to make a hash table shrink by deleting its nodes. However, it is hardly necessary. Firstly, if a keyed entry has been inserted in the table, there must be a use for it for as long as the table exists. Secondly, 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 ) ;
204        dcl hiter hi  ( 'hh'  ) ;
205        hh.DefineKey  ( 'k'   ) ;
206        hh.DefineData ( 'sat' ) ;
207        hh.DefineDone () ;
208        return ;
209     end ;
210     link declare ;
211        rc =  hi.First  () ;
212        put k= sat= ;
213        rc = hh.Delete () ;
214        rc = hi.Delete () ;
215     link declare ;
216        rc = hh.Delete () ;
217        rc = hi.Delete () ;
218     stop ;
219  run ;

k=1 sat=sat
```

Of course, the most natural and trouble-free way to declare a table, process it, free, and declare the same table from scratch again is to place the entire process in a loop. This way, the declaration is easily placed ahead of references, and the compiler sees the declaration just once. In a moment, we will see an example of 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 rapidly update its satellite?
4. If the key is no longer needed, how to delete it?

Generation I programming 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 what Generation II can do to 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 a Generation II hash table can eject keyed data in a specified order, it can be used to solve the problem *in a single step*:

```
data id3max (keep = id transid sum) ;
   length transid sum 8 ;
   dcl hash  ss  (hashexp: 3, ordered: 1) ;
   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.

CONCLUSION: GENERATION I + GENERATION II

It has been proven through testing and practical real-life application that direct-addressing methods can be a great efficiency tool if used wisely. Before the advent of Version 9, the only way of implementing these methods in a SAS Data step was coding them by hand. This is what we term Generation I in this paper. While it is a lot of fun and produces great results, it is primarily efficient from the standpoint of the machine time. The main principles of Generation I are programming ingenuity, algorithmic knowledge, and “thou shalt know thy data”, “thy data” being chiefly the properties of the hash keys involved in the process.

Generation II provides an access to algorithms of the same type and hence with the same high-performance potential via an object. While being aware of its guts does not hurt, it is not necessary for a programmer to know the details, for great results - on par or better than those of Generation I - can be achieved just by following syntax rules and learning which methods cause the black box called a hash table to produce coveted results. Thus, along with improving computer efficiency, Generation II also makes great strides in programming efficiency.

If this is the case, does it mean that Generation II makes Generation I and custom hash coding obsolete?

Not necessarily. Surely it will prompt some folks, who have never touched hashing because it has not been a canned function, to start using it now. However, those very folks are likely to discover that a sizeable direct-addressing territory is better covered by the traditional Generation I hand coding. Below is a short Generation I vs. Generation II comparison list intended to outline the areas where one or the other dominate and/or coexist:

- Simple numeric key falling in a limited range. SAS date and time values are good examples. This is the area where Generation I key-indexed search completely dominates the competition both in computer and programming efficiency.
- Simple numeric key with the range up to 9 digits; no satellites needed. Bitmapping is king.
- Simple numeric key or short (up to 8-10 bytes) character key. Both generations do well. If ultimate speed is the issue, hand-coding still does better, but not by much. Generation II may have the advantage of coding simplicity.
- Composite keys. This is mainly Generation II territory. Generation I is better if the keys can be rapidly combined in a short integer. If the composite key is of the mixed type, Generation II is king.
- Retrieving data by key from a hash table in order. Generation I can provide such functionality only through array sorting. Generation II provides a hash iterator object specifically for this purpose, and once the data are in the table, it works very fast.
- Storing and handling duplicate key entries in a hash table. Generation I is more flexible here. Generation II only lets you control which duplicate takes over, but its table must be keyed uniquely.
- Dynamic Data step dictionaries. Generation II is the ideal tool here. Its table grows at run-time as new entries are added, so it is unnecessary to allocate giant memories beforehand “just in case”.

Finally, it should be noted that at this moment of the Version 9 history, the hash object and its methods are an experimental feature. 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.
- Memory usage reporting. Currently the memory occupied by a hash table appears to not be reported in the log.
- An iterator does not write key values directly into a host key variable when the NEXT and PREV methods are used. Defining the key variable additionally as a satellite data element works but looks awkward.

Putting all this aside, the advent of the Generation II hash table as the first dynamic Data step structure is nothing short of a long-time breakthrough. Hashing has always been fun, but it has never been as much fun as now.

REFERENCES

1. P.Dorfman, Table Lookup by Direct Addressing. SUGI 26, Long Beach, CA, 2001.
2. P.Dorfman, G.Snell, Hashing Rehashed. SUGI 27, Orlando, FL, 2002.
3. D. E.Knuth, *The Art of Computer Programming*, 3.
4. J.Secosky. The Data step in Version 9: What’s New? SUGI 27, Orlando, FL, 2002.

TRADEMARK CITATION

CONTACT INFORMATION

Paul M. Dorfman
4427 Summer Walk Ct., Jacksonville, FL 32258
(904) 260-6509
sashole@bellsouth.net

Gregg P. Snell
5632 Noland Road, Shawnee, KS 66216-1674
(913) 638-4640
gsnell@DataSavantConsulting.com