SESUG Paper 203-2023

Hashes From The Ashes

Paul M. Dorfman, Independent SAS® Consultant

Richard A. DeVenezia, Independent SAS Consultant

ABSTRACT

In SAS programming, using hashing for memory-resident storage and lookup began in 1998 as array implementations of hash algorithms in the SAS language. After they had been shown to seriously outperform same-purpose techniques available at the time, quite a few SAS programmers included array hashing in their repertoire. However, with the advent of the SAS hash object in 2003, array hashing has been falling into obscurity. The true merits of the hash object is one reason; but the other is the baseless and yet widespread notion that the array hashing algorithm and code is hairy beyond the average SAS user. It is quite unfortunate since under many scenarios, array hashing is simpler and much faster than the hash object. In this paper, we intend to prove, based on a simple hash scheme, that array hashing is easy to comprehend and code. We will also show that array hashing can be macro-encapsulated to be used as a canned routine largely in the same manner as the hash object. In other words, we want to raise the array hashes from ashes!

WHY HASHING?

A short answer could be this: A hash table can be *rapidly* searched, inserted into, and updated in *O(1)*, or *constant* time. This property ideally lends itself to several common data processing scenarios where other data storage table lookup organizations available in SAS come with one deficiency or another. For a detailed answer, let us consider a couple of data processing scenarios.

SORTLESS STABLE UNDUPLICATION

Imagine that we have a SAS data file *DUP* with a key variable *K* containing some duplicate key-values. We want to *unduplicate* the file by *K* using the following specs: (a) a single pass through DUP, (b) no sorting (hence *sortless*) and (c) the output records retain their relative input order (hence *stable*). To attain that, we need, while reading DUP, to be able to tell whether the value of K in the current record has already been seen in the previous records. It means that we need to keep *some kind* of lookup table *HK* keyed by K. Then we can proceed as follows:

1. Read the next record from DUP.
2. Search table HK for the key-value K.
3. If K is not found in HK, this is the first time we have seen it, so (a) output the record and (b) insert K into table HK.
4. If there are still unread records in DUP, go to 1.

The corresponding pseudo-coded DATA step would look something like this:

```
data nodup ;
  if _n_ = 1 <...organize lookup table HK...>
  set dup ;
  <...search table HK for K...>
  if <…K is not found in HK…> then do ;
    <...code to insert K into table HK...>
    output ;
  end ;
run ;
```

DATA AGGREGATION

In this scenario, we have the same file DUP with the same keys K and extra variable D (aka "data"). We need to summarize the data-values of D for each corresponding key-value K in a DATA step (a) in a single pass and (b) without sorting. To do so, we need to organize and keep a table, let us call it HKS, keyed by

K with a data variable SUM. Then we can proceed as follows:

1. Read the next record from DUP.
2. Search table HKS for the key-value K.
3. If K is found in HKS, add D from the current record to SUM in the table.
4. If there are still unread records in DUP, go to 1.
5. Else read the table and for each K, output K and the computed SUM.

The pseudo-coded DATA step is as follows:

```
data sum ;
  if _n_ = 1 <...organize lookup table HKS with K and SUM...>
  set dup end = last-record ;
  <...search table HKS for K...>
  if <... is found in HKS…> then do ;
    <...code to add D to SUM for K in table HKS...>
  end ;
  if last-record then do until (end-of-HKS) ;
    <...read HKS...>
    <... assign K and SUM from HKS to K and SUM in PDV...>
    output ;
  end ;
run ;
```

Now let us ponder what kind of table we need in this scenarios for better run time efficiency. First, it is pretty self-evident that disk storage is not good for the purpose. (Even though a SAS file indexed by K can be used to search for K more or less rapidly, adding a new key-value K to the lookup file using the MODIFY statement at run time is painfully slow. A curious reader can try it using the MODIFY statement in the DATA step to experience it first-hand.) Hence, the storage medium for the table should be RAM.

Second, since our aim is run-time efficiency, let us think what kind of search algorithm we need to implement (or use if it is available in a canned form) to organize the table. Basically, we have three choices:

1. Disordered sequential table.
2. Ordered table.
3. Direct-addressed tables.

Let us look at their pros and cons briefly with respect to our sample task.

Disordered sequential table. This kind of table (let us call it HK) can be searched only sequentially; however, it has two advantages. First, sequential search is utterly simple to code. Second, when a new key K needs to be added to HK, the insert operation is exceptionally fast: Indeed, all we need to insert a new key is to assign HK[x]=K, where X is the index of the first unoccupied table address. Thus, the time required by this operation does not depend on the number of key N already in the table. Using the formal "big *O*" notation, this fact can be expressed by saying that inserting a key into the sequential table HK runs in *O*(1), or constant, time.

However, searching a sequential table with N keys for a search-key K requires on the average N/2 comparisons of table keys to K if K is in HK - and N comparisons if K is not in HK. In other words, sequential search runs in *O*(N) time, and its search performance rapidly deteriorates as N grows. Hence, we have to reject sequentially searched table as the storage medium for our task.

Ordered table. Since the keys in the table are sorted, it lends itself to well-known binary search. Its search operation performance is very good, for it needs about log2(N)+1 comparisons between K and keys in HK to find or reject K. Thus, if for example N=2**20 ("binary million"), searching for any K, hit or miss, requires no more than 21 comparisons between keys, as opposed to about 500,000 in case of sequential search.

For our task, however, the insert operation must be as fast as the search operation, and this is where binary search falls short. Indeed, whenever we insert a new key K in the table, we need to keep the table sorted – we cannot just append a new K to the end of the table, as it may break the sorted order. Doing so in a single array means having to move on average N/2 keys in the table HK one address rightward to make room for K to be inserted – which means that the insertion operation with such (naïve) approach runs in *O*(N) time. Hence, a simple ordered table cannot serve our purpose adequately, either.

Direct-addressed tables. These tables come in different forms, such as: *key-indexed tables*, *bitmaps*, and *hash tables*. However, regardless of the variety, they are always organized to use digital properties of the keys to perform the search, insert, and update operations. Because of that, these operations over the direct-addressed tables either do not rely on comparisons between keys at all (as with key-indexing and bitmapping) or are reduced to a handful of key comparisons only at the final stage of search. The number of these final key comparisons on the average is constant irrespective of the number of keys in the table.

Thus, the time needed to perform a search does not depend on N, and so it runs in *O*(1) time. So does the update operation, since once the search-key has been found, its location is known, and the non-key data can be overwritten in its parallel location (such as a parallel array). Furthermore, the insertion operation also runs in *O*(1) time because finding an empty address for a new key goes through exactly the same process as searching. Thanks to the *O*(1) run time of all three operations, their performance over direct-addressed tables does not plummet as more keys are added and stays the same throughout the process of unduplication. Additionally, if the table is completely memory-resident, the absolute time of each operation is very fast in itself.

WHICH HASHING?

Thus, a direct-addressed table (replete with the underlying algorithms) is what we want for our sample task (and, as we will see later, other tasks of the same nature). Since we have agreed on using pure RAM for table storage, we are down to two choices:

1. Using the SAS hash object.
2. Using direct-addressed tables based on SAS arrays.

1. USING THE HASH OBJECT

Nowadays, most SAS programmers would most likely automatically opt for the hash object since it is a canned documented instrument requiring no custom coding and/or detailed understanding of its underlying algorithm. The SAS hash object is well documented by SAS, and there is plenty of literature on the subject, including two SAS Press books. For the endeavor at hand, however, even the mere SAS Object Tip Sheet:

https://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf

offers enough information to code a solution.

First, let us concoct some sizable random sample data with duplicate key-values K and satellite data-values D. The step below writes a sample data set DUP with 10 million records. The key-values K are natural numbers up to 8-digit long with about 500,000 duplicate values; the data values D are the natural numbers from 1 to 10 million.

```
data dup ;
  retain K D ;
  do D = 1 to 1e7 ;
    K = ceil (ranuni(1) * 1e8) ;
    output ;
  end ;
run ;
```

Now, we can proceed to solve our sample problems using the hash object.

Unduplication

```
data nodup_hash_object ;
  if _n_ = 1 then do ;
    dcl hash hk () ;           /* create an instance of hash */
    hk.definekey  ("k") ;      /* key by K                   */
    hk.definedone () ;         /* initialize instance        */
  end ;
  set dup ;                    /* read next record from DUP  */
  if hk.check() ne 0 then do ; /* if K is not in HK          */
    output ;                   /* write record out           */
    hk.add() ;                 /* add K to HK                */
  end ;
run ;
```

Note that alternatively, the last IF-DO-END construct can be replaced with more concise:

```
if hk.add() = 0 ;
```

since a successful ADD method call means that SK is not in H and is thus being added to H, while the subsetting IF outputs the record and moves program control to reading the next.

Aggregation

```
data sum_hash_object (keep = k sum) ;
  if _n_ = 1 then do ;
    dcl hash hks () ;                 /* create instance if hash HKS     */
    hks.definekey  ("k") ;            /* key by K                        */
    hks.definedata ("k", "sum") ;     /* K and sun=sum(D) per K          */
    hks.definedone () ;               /* initialize instance             */
    dcl hiter hi ("hks") ;            /* iterator to harvest K and SUM   */
  end ;
  set dup end = z ;                   /* read next record from DUP       */
  if hks.find() ne 0 then SUM = d ;   /* if K is not in HK, init SUM=D   */
  else                    SUM + d ;   /* else set SUM=SUM+D in PDV       */
  hks.replace() ;                     /* updt SUM in HKS with SUM in PDV */
  if z then do while (hi.next() = 0) ; /* iterate to harvest K and SUM    */
    output ;
  end ;
run ;
```

Notes

The hash object approach definitely works – who would doubt it? – so, we have at least one way to have working solutions via hash tables. We will discuss its relative efficiency later after having seen how the same thing can be done using SAS arrays. Meanwhile, the reader is encouraged to run the two steps above with the option FULLSTIMER turned on to get a feel of the run times and memory usage.

2. DIRECT ADDRESSING USING ARRAYS

Historically, the hash object was not the first foray into hashing in SAS programming. Before the advent of its experimental version in SAS 9.0 in 2003, techniques based on direct addressing algorithms using SAS arrays had already been developed by the authors of this paper and successfully used practically by a number of enthusiastic SAS programmers.

The impetus for their development circa 1998 was the practical necessity to join large files via table lookup efficiently, without sorting the inputs for match-merging either explicitly or implicitly. Using the array-based direct addressing techniques for the purpose quickly demonstrated that they significantly

outperformed other table lookup methods available at the time, such as SAS indexes or formats.

Unfortunately, the fact that direct-addressing array techniques handily outperformed canned methods has somehow made them perceived as some kind of wonder and their code – as mysteriously complex. Perhaps the fact that they were initially embraced mainly by well-known advanced SAS programmers also played a part in forming that perception. Thus, after the advent of the SAS hash object and its immediate enthusiastic adoption by the original developers of the direct-addressing array methods themselves, they started fading out and eventually almost disappeared from the SAS programming scene.

This is unfortunate because under many practical data management scenarios, array-based direct addressing is quite a bit faster and easier on memory than the hash object. This should come as no surprise since the hash object is a wonderful versatile run-time tool way beyond simple table lookup – a sort of a Swiss Army knife of dynamic DATA step programming. But its versatility, like that of any multi-purpose tool, comes at a price. By contrast, specific direct-addressing array-based techniques are better tailored to specific data processing scenarios and thus lighter and wieldier under the right circumstances. Our sample task of stable sortless unduplication is typical of one of those scenarios.

Also, the coding complexity of array-based direct addressing is actually a myth. Of the three direct-addressing array techniques – key-indexing, bitmapping, and hashing – only bitmapping can be considered technically somewhat beyond average. There is no simpler table lookup thing than key-indexing, and array hashing – its logical extension – is also quite uncomplicated, as we shall see shortly.

To prove it, let us now, without further ado, reveal how *array hashing* can be used to solve the sample problems at hand. Both solutions offered below are based on an implementation of the array hashing search-and-insert scheme called *open addressing with linear probing*. Note that this and other different schemes of array hashing have been presented in a number of SAS papers since 1999 with all their gory details. They are listed in References at the end of this paper, so it would be entirely superfluous to reproduce their contents here. Also, the programs below are annotated, and some notes will be added where they may be due.

Unduplication

```
%let h = 20000003 ; /* first prime number ≥=> 2 * nobs(DUP) */

data nodup_hash_array (keep=K D) ;
  array hk [&h] _temporary_ ;                 /* hash table HK, sized 1:&H    */
  set dup ;                                   /* read next record from DUP    */
  do x = mod (k, &h) + 1                      /* hash K into 1:&h range       */
    by 1 until (hk[x] = k or cmiss (hk[x])) ; /* go up HK til hk[x]=K or null */
    if x > &h then x = 1 ;                    /* if x > dim(hk), circle back  */
  end ;
  if cmiss (hk[x]) then do ;                  /* if K is not found in HK      */
    output ;                                  /* write record out             */
    hk[x] = k ;                               /* and add K to table HK        */
  end ;
run ;
```

Notes:

- The macro variable H is used to size the hash key array HK. To get H, we set it to the first prime number greater than take twice the number of records in DUP. Because of that, the array HK will always be at least half-empty, and the divisor of the MOD function on line 4 will be prime.
- MOD(k,&h)+1 is our hash function. It maps its argument K to the remainder of the division shifted up by 1, i.e. always within the bounds of the array HK. Prime H makes the mapping highly random, which helps avoid mapping too many distinct keys to the same address X.
- Some keys will inevitably map to the same address X; this is expected. When it happens, it is called a *collision*. In our code, we know a collision occurs if at the address X computed by hash function, we find that HK[x] is not missing – in other words, the *address is occupied*. Something needs to be done to *resolve the collisions* since we cannot store two distinct keys at the same address X. That something is termed a *collision resolution policy*.
- There exist many such policies; the one employed here is the simplest one called *linear probing*. It

means that if a collision occurs at address X, we simply go up the table and *probe* one address at a time. If in the process, X overflows the upper array bound, we circle back to the beginning of HK by setting X=1.

- When we bump into an empty address, we stop the DO loop and place our key. On the other hand, if while probing, we find HK[x]=K, the K is already in the table, so we stop the DO loop.
- This DO loop, though comprising only 3 lines of code, is a full implementation of the hash search linear probing algorithm. After the loop, we have only 2 mutually exclusive outcomes: (a) HK[x] is missing, i.e. K is not found in the table, and (b) HK[x]=K, i.e. the key is found. Since we also know the index X where either occurred, we can now do what we need depending on the task.
- For example, if we only need to find if K is in the table, we are done. For our task at hand (unduplication) we need to store HK[x]=K if K is not yet in the table, so that if the same key-value should occur downstream, we will know it is already in the table.
- Instead of hard-coding H outside the program, we can instead compute it programmatically:

```
data _null_ ;
  retain load_factor 0.5 ;
  do p = ceil (nobs / load_factor) by 1 until (j = up + 1) ;
    up = ceil (sqrt (p)) ;
    do j = 2 to up until (not mod (p,j)) ;
    end ;
  end ;
  call symputx ("H", p) ;
  stop ;
  set dup nobs=nobs ;
run ;
```

The value of LOAD_FACTOR determines how sparse (or full) the hash table is – in other words, how many addresses will remain empty when it is loaded with keys: The smaller the load factor, the sparser the table. Sparsity reduces collisions and makes the hash search faster, yet it also increases the memory footprint by making the table larger. In our case, the load factor is 0.5, meaning that at least 50 percent of the addresses in HK will remain unoccupied.
With linear probing, the load factor of 0.5 or even less is recommended to avoid unpleasant effects, such as open addressing clustering (out of scope for this paper). The load factor of 0.5 is a good choice, making the table neither too tight nor too heavy on memory. Other, more complex, collision resolution policies allow for less sparse tables and hence better memory utilization. However, the utter simplicity and tight code of linear probing makes it very hard to beat practically.

Aggregation

```
%let h = 20000003 ;

data sum_hash_array (keep=k sum) ;
  array hk [&h] _temporary_ ;                 /* hash key  array              */
  array hd [&h] _temporary_ ;                 /* hash data array, parallel    */
  set dup end = z ;                           /* read next obs from DUP       */
  do x = mod (k, &h) + 1                      /* hash K into 1:&h range       */
    by 1 until (hk[x] = k or cmiss (hk[x])) ; /* probe HK til hk[x]=K or null */
    if x > &h then x = 1 ;                    /* if x > &h circle back to x=1 */
  end ;
  hk[x] = k ;                                 /* add K to HK                  */
  hd[x] + d ;                                 /* add D to hd[x] in place      */
  if z then do x = 1 to &h ;                  /* harvest K and SUM from HK/HD */
    if cmiss (hk[x]) then continue ;          /* ignore empty array addersses */
    k   = hk[x] ;                             /* get K and SUM from HK/HD     */
    SUM = hd[x] ;                             /* write record out             */
    output ;
  end ;
run ;
```

Notes

After the first (hash search) DO loop ends, X points at either empty HK[x] or occupied by the key K.
HK[x]=K adds K if it is not in the table and does not change HK[x] if it is.
HD[x]+D updates the data array directly in place by adding D to the array item HD[x]. Note that the same cannot be done with the hash object.
The last DO loop simply harvests K and SUM from the table. The addresses where no keys (and hence data) have been stored are ignored.

RELATIVE PERFORMANCE

All of the above code was tested under 64-bit Windows10 Pro (SAS 9.4 TS 1M6). Here are the results:

	Unduplication		**Aggregation**	
	Time, seconds	**RAM, MB**	**Time, seconds**	**RAM, MB**
Hash object	15.0	768	17.8	1024
Hash array	2.5	174	4.2	305

It appears that array hashing makes enough difference in efficiency to consider it as the tool of preference for the task at hand. At least, instead of mindlessly reaching in the pocket for the proverbial Swiss knife (the hash object), one might consider the effort of walking a few feet to the old long-forgotten toolbox covered with dust and ashes and pulling out a drawer to find a sharper knife. It may seem a mystery that a piece of code outwardly looking so simple and much like sequential search outperforms a really great and fast canned tool by a huge margin. There is no mystery, of course; rather, there are reasons, for example:

- **Search.** Given a key to search for, the ***hash object*** calls the internal hash function to identify the only AVL tree where the key can be found and traverses the tree to find or reject the key, essentially performing *binary search*.

  By contrast, with the ***hash array****,* the hash function MOD maps the key to one of the array items HK[x] and then probes if HK[x]=K, H[x+1]=K, H[x+2]=K, etc. to find the key or reject it by bumping into an empty HK[x]. Thus essentially, it performs *sequential* search. However, because of the numeric properties of the hash function and HK being always at least half-empty, this search has to probe, on the average, no more than 3 locations regardless of the key and whether it is found or not.

  Since sequential search always outperforms binary search at N < 7 (N is the number of keys in the table), the hash array results in faster search times. Indeed, comparison tests we have concocted to search for 100,000,000 keys (half in the table and half – not) show that on average, searching the hash array is more than 4 times faster than searching the hash object.

- **Insertion**. Given a key to insert in the table, the ***hash object*** ultimately inserts it into an AVL binary tree, which involves node rotations to keep the tree ordered and balanced.

  By contrast, inserting a key into the ***hash array*** HK (after searching it to identify an available location X) involves nothing more than the assignment HK[x]=K. Comparison tests we have run show that it is almost 6 times faster than adding a key into the hash object (despite the fact that the underlying AVL tree operations are coded in C efficiently as possible).

- **Aggregation.** Aggregation means that we have a key whose corresponding data in the table needs to be updated using the current table data-value – for example, by adding a value to it. With the ***hash object***, its corresponding *data-value cannot be modified in the table directly* by performing an operation on it. Instead, we need to (a) call the FIND method to *retrieve* the corresponding data-value from the table into the PDV host variable, (b) update the PDV variable – e.g., by adding a value to it, and (c) call the REPLACE method to overwrite the old data-value in the table with the new value in its host variable. Thus, data aggregation with the hash object involves two-way traffic: First, from the hash table to the PDV and then - from the PDV to the hash table.

  By contrast, with the ***hash array***, to modify a data-value in a data array HD[x] *parallel* to the key array HK for a given key K found at HK[x]=K, all we need to do for updating the table for this key with

a new value D is to assign HD[x]=D where HD is a parallel array containing table data (see the data aggregation example below). In other words, we directly overwrite the value in the array HD once the key is found in HK[x].

PROS, CONS, CAVEATS

Since there exists no free lunch, what array hashing gains in performance compared to the hash object it loses in versatility. The question is how some of this deficiency can be addressed – and at what price. Let us discuss several aspects of the issue.

Character Keys

One of the reasons array hashing is so efficient is its hash function:

X = mod (K, &h) + 1

For any hashing scheme to perform well, its hash function must be (a) very fast and (b) minimize collisions. Generally speaking, these two requirements are contradictory. In order to minimize collisions, we want to involve as many bits of the key as possible in the hash function computation to avoid presenting two different keys to the function as the same argument. But on the other hand, having to process more key bits slows the hash function down.

The hash function above is an exception because it both consumes all the key bits and is as fast as the MOD function itself. The division by a prime number is known to result in a random remainder (the reason why prime numbers are used in random number generators). However, this lucky confluence of all needed features is based on the fact that K is numeric, and so all of its 64 bits are involved in computing X.

If the key K is character, it of course also represents a number – a base 256 number, that is. For up to 6 characters, this number can be calculated as input(K,pib6.), which maxes out at 256**6, i.e. 2.8E14; but above 6, the expression loses its integer precision. However, the hash function:

```
X = mod (input (K, pib6.), &h) + 1
```

is still a good and reasonably fast computing choice if the first 6 bytes of different keys are not too similar. Of course, if you know (e.g. from business rules) that K is a digit string (i.e. consists of digits only), you can use the best informat with the INPUT function, for instance:

```
X = mod (input (K, best15.), &h) + 1
```

However, both suggestions mean making assumptions about input data. This is not a very good idea since it narrows the area of applicability of our code. A better idea is to involve all the bits of the key to produce a highly random result and only then use the result with the PIB informat to generate an integer number. This can be done with the aid of the MD5 function:

```
X = mod (input (md5 (K), pib6.), &h) + 1
```

It is doubtless quite a bit slower than the straight modulo or applying PIB6 directly. However, it guarantees that two keys differing even in a single bit will be fed to the INPUT function as two *vastly different* MD5 responses. This is because MD5 itself is a hash function, albeit in a different sense – namely, it outputs a highly random 16-byte hex digit character string, always unique to any possible input character string (well, in all candor: almost always, but it does not matter in our context since we allow collisions anyway).

To get a feeling of how much slower array hashing is with a simple character key compared to numeric, let us first rewrite the data set DUP:

```
data dup ;
  retain K D ;
  do D = 1 to 1e7 ;
    K = put (ceil (ranuni(1) * 1e8), best.) ;
    output ;
  end ;
run ;
```

Now let us try the new hash function using the unduplication task as a test mule. We need to (a) alter the hash function as discussed and (b) change the key array HK from numeric to character:

```
data nodup_hash_array (keep=K D) ;
  array hk [&h] $16 _temporary_ ;
  set dup ;
  do x = mod (input (md5 (k), pib6.), &h) + 1 by 1
    until (hk[x] = k or cmiss (hk[x])) ;
    if x > &h then x = 1 ;
  end ;
  if cmiss (hk[x]) then do ;
    output ;
    hk[x] = k ;
  end ;
run ;
```

This step runs in 7.8 seconds using 610 MB of RAM. So, compared to the case with the numeric key, it is 3 times slower and 3 times hungrier for memory. Note, though, that not all of this is due to the heavier hash function. It is also partly due to the fact that comparisons between $12 character keys are slower than between numeric keys. With respect to the increased memory usage, a $16 character array occupies 3 times the memory of an equi-dimensional numeric array.

Still, even with the character K, the hash array is much faster and lighter compared with the hash object since the latter runs in 19 seconds consuming 1024 MB of RAM. However, it has the advantage that its code is the same regardless of the key data type.

Composite keys

The hash object handles the composite keys seamlessly: You merely call the DEFINEKEY method to define them, and the internal hash function handles the rest. To accommodate composite keys with array hashing, we have to adjust the hash function to consume all the key variables as arguments. At first glance, we would also need, for every additional key variable, to add a proper data type array parallel to HK - and as many corresponding comparisons to HK[x]=K. Luckily, this is not the case: Again, the MD5 function comes to the rescue.

First, a new test data set DUP with the composite key [KN,KC]:

```
data dup ;
  retain KN KC D ;
  do D = 1 to 1e7 ;
    KN = ceil (ranuni(1) * 1e8) ;
    KC = put (kn, 12.) ;
    output ;
  end ;
run ;
```

Now, the updated unduplication step to work against the composite key:

```
data nodup_hash_array (keep=KN KC D) ;
  array hk [&h] $16 _temporary_ ;
  set dup ;
  k = put (md5 (catx (" ", KN, KC)), 16.) ;
  do x = mod (input (k, pib6.), &h) + 1
    by 1 until (hk[x] = k or cmiss (hk[x])) ;
    if x > &h then x = 1 ;
  end ;
  if cmiss (hk[x]) then do ;
    output ;
    hk[x] = k ;
  end ;
run ;
```

So, why do we not need a separate array and comparison for KN and KC? The trick is to compute a single simple key K from the component key variables KN and KC and use K both for the array key-values and

search comparisons. Needless to say, it is based on the strict one-to-one correspondence between the MD5 argument and its response. (It is well-known that purely theoretically, it can be broken by intentionally contriving the argument. Practically, however, it is impossible to do by pure chance.)

The hash object unduplication code needs only one-line change (shown in boldface):

```
data nodup_hash_object ;
  if _n_ = 1 then do ;
    dcl hash hk () ;
    hk.definekey  ("KN", "KC") ;
    hk.definedone () ;
  end ;
  set dup ;
  if hk.check() ne 0 then do ;
    output ;
    hk.add() ;
  end ;
run ;
```

The array hashing step runs in 8.3 seconds using 610 MB of RAM. The hash object step runs in 18.3 seconds and uses 1281 MB of RAM. For the aggregation sample task, recoding for the character and composite keys results in similar relative statistics. A curious reader can undertake such recoding as an exercise.

Based on the test results presented above, we can reasonably conclude that array hashing holds a sizable performance edge over the hash object with respect to such operations as search, insert, and update.

Hash Join

An alert reader would almost inevitably ask why such a staple as joining data sets via hashing is not included in the test schedule. Well, in all candor, we hardly need to test it directly since we already know that the insert and search operations are much faster with array hashing, ergo joining data sets – essentially an insert-and-search endeavor – will follow suit. But just for the sake of completeness, let us go ahead, concoct some test data and run the hash object and hash array code back to back:

```
data lookup ;
  do _n_ = 1 to 1e7 ;
    K = ceil (ranuni(1) * 1e8) ;
    output ;
  end ;
run ;

data driver;
  retain K D ;
  do K = 1 to 2e8 ;
    D = put (K, z9.) ;
    output ;
  end ;
run ;

data join_hash_object (keep=K D) ;
  if _n_ = 1 then do ;
    dcl hash hk () ;
    hk.definekey  ("K") ;
    hk.definedone () ;
    do until (z) ;
      set lookup end = z ;
      hk.ref() ;
    end ;
  end ;
  set driver ;
  if hk.check() = 0 ;
run ;
```

```
%let h = 20000003 ;

data join_array_hash (keep=K D) ;
  array hk [&h] _temporary_ ;
  if _n_ = 1 then do until (z) ;
    set lookup end = z ;
    link search ;
    hk[x] = k ;
  end ;
  set driver ;
  link search ;
  if hk[x] = k ;
  return ;
  search: do x = mod (k, &h) + 1 by 1 until (hk[x] = k or cmiss (hk[x])) ;
            if x > &h then x = 1 ;
          end ;
run ;
```

Adding the running stats for the above to the comparison table showed earlier, we have:

	Unduplication		Aggregation		Equi Join	
	Time, seconds	**RAM, MB**	**Time, seconds**	**RAM, MB**	**Time, seconds**	**RAM, MB**
Hash object	15.0	768	17.8	1024	125	768
Hash array	2.5	174	4.2	305	37	174

Memory handling

Run time memory allocation is one of the greatest assets of the SAS hash object. This is what makes it a truly dynamic DATA step structure capable of emerging, disappearing, expanding, shrinking, store pointers to other hash object instances as data (see, e.g., Loren and DeVenezia, 2011), etc. – all during run time. In the context of simple tasks like demonstrated above, it means that there is no need to calculate or guesstimate how much memory to allocate in order to not run out of it at run time.

However, like many great things, dynamic allocation comes with caveats:

- First, memory is acquired one hash item at a time, meaning that the underlying memory allocation function is called as many times as there are hash items – which in heavy applications can approach billions. Each time it needs more memory, the hash object has to compete with other system processes doing the same thing.
- Second, the closer the already allocated hash object's memory nears the MEMSIZE limit, the longer it takes it to rummage around the system to find memory for the next hash item, and it can slow the hash object performance down to a painful crawl (as the authors have had the misfortune to witness more than once).
- Finally, the hash object can – and often does – eventually run out of memory with the corresponding error message in the log. Unfortunately, in heavy applications like preaggregation of enterprise analytic data it can occur (and does at the worst possible time) after hours of data processing – which means that the dearth of system memory has to be addressed and the process needs to be started from scratch.

By contrast, the hash array is allocated at compile time. It means that we need to do some preliminary work to estimate or calculate how many array items we need for the job. A little program shown above that calculates the prime number for the array dimension based on the maximum number of unique keys and selected load factor is an example. Of course, needing to do such work is a disadvantage. Also, the need for memory can be overestimated, and one may tend to grab too much aiming to cover all bases at the expense of other users in the system.

However, compile time array memory allocation also has its marked advantages, to wit:

- If the system does not have enough RAM to allocate the array, the DATA step abends at once, i.e., before any DATA step processing could even begin. It eliminates the distressing possibility of running out of memory when data processing is almost over.
- If there is any competition with other memory grabbers, it all occurs very quickly at compile time. For example, for successful allocation of 2 GB of array memory, the DATA step needs less than 1 second. And once the array memory is allocated, it is locked in exclusive use by the DATA step for its entire duration.
- As an icing on the cake, hash arrays need significantly less memory per item than the hash object. The reason is that the hash object is vastly richer in functionality, and so more memory is needed to make it work. But if we do not need the entire Swiss knife with all its bulk, a separate dedicated tool can be better for the job.

ENCAPSULATION

Heretofore, the narrative in this paper has been aimed to convince anyone interested in the topic that array hashing code is no more complex that anything an average SAS programmer can muster. However, if we have nonetheless failed despite our best efforts, there is still a remedy.

After the development of array hashing by SAS users circa 1998-2000, one benevolent SAS programmer (known to some as RAD) realized that some folks would be still intimidated by having to code hashing by hand (even via cannibalization) and thus miss out on many of its performance benefits - all the more that at the time, it was the only form of hashing in SAS (the hash object came out later).

RAD came to the rescue of those suffering from *hashcodefobia* by creating a system of macros encapsulating different hash table operations. He then formulated simple rules of applying the macros to solving various data processing tasks. Furthermore, he provided parameters for tuning hashing performance. For example, the user can select between different collision resolution policies. Above, only the linear probing policy was discussed; however, using RAD system, you can also choose more than one. Some work better with tables for load factors greater than 0.5 and thus can save some memory. Needless to say, the user can independently choose a value of the load factor itself.

RAD eventually published his array hashing macro system free for anyone to use on his web site www.devenezia.com here:

SAS Macros - by Richard A. DeVenezia - hash-macros.sas

HASH ARRAY MACRO API

The requirements of this API were abstracted from the Dorfman papers and SAS-L threads about hash array. The API would need to provide macros for hash array declaration and utilization. Because the macros are used to generate inline data step code parts none of the macros could be implemented in a way that caused a step boundary. The impact of this requirement can be seen in the implementation of the `hashSize` macro.

One of the trickier implementation issues was that the hash array name in the declaration could not be presumed in any way, and that same name would need to be used in the utilization macros. The decision was made to use the declared hash array name as part of global macro variable names as the modus operandi. The details are not particularly important to an API user but understanding leads to better management of the global macro symbol table in a large system of code.

DECLARATION

```
%macro declareHash (hash, size, policy=LP, allowDuplicates=NO);
%macro declareHashKey (hash, key);
%macro declareHashData (hash, var);
```

API reserved globals that use hash name

```
  %global hash_size_&hash;
  %global hash_policy_&hash;
```

```
  %global hash_dups_&hash;
  %global hash_add_count_&hash;
  %global hash_lp_increment_&hash;
  %global hash_key_&hash;
```

The preface `hash_lp_increment_` is 18 characters long, so the hash name should be no longer than 14 characters, otherwise the error "invalid symbolic variable name" will occur.

The brave code jockey looking into the API source will see the resolution of these symbols uses the double ampersand syntax, for example: `&&hash_size_&hash`

UTILIZATION

The hash array is breathed utility through these macros:

```
%macro hashAddKey (hash, rc=);
%macro hashFetch (hash, rc=);
```

The above macros symbolically vector into API internal policy specific macros that emit the linear probing (LP) and coalesced linking (CL) data step code parts:

```
%macro hashAddKey_LP (hash, rc=);
%macro hashAddKey_CL (hash, rc=);
%macro hashFetch_LP (hash, rc);
%macro hashFetch_CL (hash, rc);
```

The code parts emitted by the declaration and utilization macros reserve PDV variables that are unlikely to collide with data set variables.

```
&hash._keys
&hash._max_depth
&hash._key_addr
&hash._&var.
&hash._addkey_end_block_&count.
&hash._depth
```

Further discussion of the API is outside the scope of this paper.

CONCLUSION

When one has a data processing task that looks like it can be best solved by using a fast lookup table with *O*(1) search and insert operations, a hash table of some kind is the only choice. Choosing the hash object is natural since its hash table is canned. However, for many tasks, an array hash table is so much more efficient that it is worth helping 25 year old code rise from ashes – at least the authors are sure of it.

There exists a myth that coding hashing via arrays is a narrow niche for the brainiest: In actuality, it requires a basic understanding of the algorithm and DATA step competency. Moreover, array hashing code is abundant in SAS user literature and thus open for cannibalization. Furthermore, it is encapsulated in an openly published system of operational macros referenced above.

REFERENCES

Dorfman, Paul. 1999. "Array Lookup Techniques: From Sequential Search to Key-Indexing (Part1)". Proceedings of the SESUG 1999 Conference, Mobile, AL: SAS Press.

Dorfman, Paul. 1999. "Array Lookup Techniques: From Key-Indexing To Hashing (Part2)". Proceedings of the SESUG 1999 Conference, Mobile, AL: SAS Press.

Dorfman, Paul. 2001. "Table Lookup by Direct Addressing: Key-Indexing, Bitmapping, Hashing". Proceedings of SUGI 2001 Conference, Long Beach, CA: SAS Press.

Snell, Gregg and Dorfman, Paul. "Hashing Rehashed". Proceedings of SUGI 2002 Conference, Orlando, FL: SAS Press.

Snell, Gregg and Dorfman, Paul. "Hashing: Generations". Proceedings of SUGI 2003 Conference, Seattle, WA: SAS Press.

Loren, Judy and DeVenezia, Richard. “Building Provider Panels: An Application for the Hash of Hashes”. Proceedings of SAS Global Forum 2011, Las Vegas, NV: SAS Press.

Vyverman, Koen and Dorfman, Paul. “Black Belt Hashigana”. Proceedings of SAS Global Forum 2010, Seattle, WA: SAS Press.

Dorfman, Paul and Shajenko, Lessia. “Efficient DATA Step Random Sampling Out of Thin Air”. Proceedings of SESUG 2018, St. Pete Beach, FL: SAS Press.

Dorfman, Paul. “Array Searching Algorithms and Techniques”. Proceedings of SESUG 2022, Mobile, AL:SAS Press.

Henderson, Don and Dorfman, Paul. “Dynamic Programming With The SAS Hash Object”. Proceedings of SAS Global Forum 2020 Virtual Conference: SAS Press.

CONTACT INFORMATION

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

Paul M. Dorfman
sashole@gmail.com

Richard A. DeVenezia
rdevenezia@gmail.com