Paper 2000-2015

Data Aggregation Using the SAS® Hash Object

Paul M. Dorfman, Independent Consultant
Don Henderson, Henderson Consulting Services, LLC

ABSTRACT

Soon after the advent of the SAS hash object in Version 9.0, its early adopters realized that the potential functionality of the new structure is much broader than basic *O(1)*-time look-up and file matching. Specifically, they went on to inventing methods of data aggregation based on the ability of the hash object to quickly store and update key summary information. They also demonstrated that the DATA step aggregation using the hash object offered significantly lower run time and memory utilization compared to the SUMMARY/MEANS or SQL procedures, coupled with the possibility of eliminating the need to write the aggregation results to interim data files and programming flexibility allowing to combine sophisticated data manipulation and adjustments of the aggregates within a single step. Such developments within the SAS user community did not go unnoticed by the SAS R&D, and for Version 9.2 the hash object had been enriched with tag parameters and methods specifically designed to handle aggregation without the need to write the summarized data to the PDV host variable and update the hash table with new key summaries, thus further improving run-time performance. As more SAS programmers applied these methods in their real-world practice, they developed aggregation techniques fit to various programmatic scenarios and ideas for handling the hash object memory limitations in situations calling for truly enormous hash tables. This paper presents a review of the DATA step aggregation methods and techniques using the hash object. The presentation is intended for all situations where the final SAS code is either a straight Base SAS DATA step or DATA step generated by any other SAS product.

INTRODUCTION

Rather than speculating what "data aggregation" means in a broad sense, let us define it in terms of the scope of this paper. Consider a data set DETAILS with a *key variable* KEY and *satellite variable* VAR:

```
data DETAILS ;
   input KEY VAR ;
cards ;
1   1
1   1
1   2
1   2
2   1
2   2
2   2
2   3
;
run ;
```

The data set above is a simplification intended to show the concept. Real data sets may contain more than one key variable forming a *composite key,* and/or more satellite variables in addition to VAR.
The purpose of data aggregation is to determine certain collective data set characteristics for each key (simple or composite) based on the values of the satellite variable(s). For example, such characteristics

may include descriptive statistics. For example, using the standard SAS notation, it can be SUM(VAR) (equal to 6 for KEY=1 and to 8 for KEY=2, above), N(VAR) (equal to 4 for both keys), or, the number of distinct values of VAR for each key (equal to 2 for KEY=1 and to 3 for KEY=2).

The sum and the number of unique values are the two characteristics (statistics) we will deal in this paper (indirectly, it includes N because calculating it is no different from that of SUM).

However, data aggregation in a broader sense can answer other business questions. For example, if the data set records represent certain events, it may be of interest to know whether two or more events occurred on the same date or, conversely, the same event occurred on more than one date. The SAS hash object is an extremely powerful tool for data aggregation understood in this sense, and the corresponding techniques (especially those dealing with *partially ordered* data) by themselves can be a topic for a sizable article. In this paper, we will leave them outside of its scope.

This paper, along with the PowerPoint and sample progams, has a corresponding article on sasCommunityr.org. The article Data Aggregation Using the SAS Hash Object will be updated by the authors based on the questions we receive as a result of this presentation.

TWO PRINCIPAL METHODS OF AGGREGATION

In general, all aggregation methods can be broken down into two broad categories:

1. The methods based on sorting and control-break (by-processing)
2. The methods based on table look-up

Our little sample data set can help illustrate the principal difference.

1. SORT-CONTROL-BREAK

First, let us consider the *method based on sorting*. If the data set is sorted by [KEY, VAR] (data set DETAIL is already ordered this way *intrinsically*), both the sum and the count of unique values of VAR for each KEY can be computed by making use of the fact that the same values of KEY and VAR are physically separated into distinct groups of records, in other words, into by-groups. Such a separation facilitates what is usually termed *control-break* programming. For example, SAS code based on pres-sorting may look like this:

Example 1

```
data AGGREGATE (drop = VAR) ;
   do until (last.KEY) ;
      set DETAILS ;
      by KEY VAR ;
      SUM_VAR = sum (SUM_VAR, VAR) ;
      CNT_VAR = sum (CNT_VAR, first.VAR) ;
   end ;
run ;
```

Essentially, the same occurs behind-the-scenes if you use the SUMMARY/MEANS procedure with the BY statement to compute the sum or the SQL procedure with the GROUP BY clause to compute both the sum and count (distinct VAR).

2. TABLE LOOK-UP

Second, let's assume that DETAILS data set is not sorted (inherently or otherwise) by [KEY, VAR] and consider the *table look-up method*. An aggregation algorithm can be described as follows. Set up two look-up tables:

Table H

Key Portion	KEY
Data Portion	KEY, SUM, UNQ

Table U

Key Portion	KEY, VAR

Suppose that tables H and U reside completely in RAM and are organized for rapid table look-up. The latter means that a certain quick mechanism exists that can be used to: (a) find a key in the table or determine that it is not there yet, (b) if the key is in the table, retrieve the associated values of the data portion variables and, if necessary, replace them with different values, and (c) insert a new key and the associated data portion values into the table. The "quick mechanism" indicates, more specifically, that the operations (a), (b), and (c) can be performed in *O(1)* time. Just in case it sounds cryptic, *O(1)* merely means that the time necessary to perform any such operation does not depend on the number of distinct keys in the table. In other words, if the table comprises 1,000 key entries and it takes 1 microsecond to check if a key is in the table (actually, a realistic ballpark figure by the order of magnitude), it will also take about 1 microsecond for the same table with 1,000,000 key entries. Jumping a bit ahead, look-up tables possessing such a property are known as hash tables. In SAS, they can be facilitated using arrays (which requires a deep understanding of hash algorithms and advanced programming skills) or the "canned" SAS hash object (the only thing required is learning how to push its buttons).

Now, if we have the tables H and U with such properties in place, we can solve the aggregation problem as follows. For every record read from data set DETAILS:

1. Look for the value of KEY from this record in table H. If it is not there, set SUM and UNQ to missing values. Add VAR to SUM already in the table for this KEY.
2. If compound key (KEY,VAR) is not in table U, add 1 to UNQ, then insert (KEY,VAR) into table U. Replace the values of SUM and UNQ in table H for the current value of KEY.
3. After the last record has been read from DETAILS, table H contains all distinct values of KEY and the associated computed values of SUM and UNQ, so output its content to data set HASH_AGG.

SAS code using the SAS hash object to organize and use tables H and U in order to aggregate the data with the same end result as the sorting-based method, may look like this:

Example 2

```
data _null_ ;
   dcl hash H (ordered: "A") ;
   h.definekey  ("KEY") ;
   h.definedata ("KEY", "SUM", "UNQ") ;
   h.definedone () ;

   dcl hash U () ;
   u.definekey ("KEY", "VAR") ;
   u.definedone () ;
```

```
   do until (end) ;
      set DETAILS end = end ;
      if h.find() ne 0 then call missing (SUM, UNQ) ;
      SUM = sum (SUM, VAR) ;
      if u.check() ne 0 then do ;
         UNQ = sum (UNQ, 1) ;
         u.add() ;
      end ;
      h.replace() ;
   end ;

   h.output (dataset: "hash_agg") ;
   stop ;
run ;
```

(As a side note, the table look-up method is employed, behind the scenes, by any procedure using the CLASS statement to calculate the sum. The look-up tables used by those procedures are based on an underlying-software internal *AVL tree*, very efficiently and tightly coded. Incidentally, the same tree type is used to implement the SAS hash object.)

SINGLE-PASS AGGREGATION

In Example 2, the aggregation task is accomplished in a single pass through the input data. Even more importantly, if we had another variable (named VAR2, say) and had to compute the same aggregates for both VAR and VAR2 independently, all we would need to is to ape the code already in place for VAR. It can be illustrated with the following example:

Example 3

```
data DETAILS2 ;
   input KEY VAR VAR2 ;
cards ;
1   1   2
1   1   3
1   2   3
1   2   4
2   1   7
2   2   7
2   2   5
2   3   5
;
run ;

data _null_ ;
   dcl hash H (ordered: "A") ;
   h.definekey  ("KEY") ;
   h.definedata ("KEY", "SUM", "UNQ", "SUM2", "UNQ2") ;
   h.definedone () ;

   dcl hash U () ;
   u.definekey ("KEY", "VAR") ;
   u.definedone () ;
```

```
   dcl hash U2 () ;
   u2.definekey ("KEY", "VAR2") ;
   u2.definedone () ;

   do until (end) ;
      set DETAILS2 end = end ;
      if h.find() ne 0 then call missing (SUM, SUM2, UNQ, UNQ2) ;
      SUM  = sum (SUM, VAR) ;
      SUM2 = sum (SUM2, VAR2) ;
      if u.check() ne 0 then do ;
         UNQ = sum (UNQ, 1) ;
         u.add() ;
      end ;
      if u2.check() ne 0 then do ;
         UNQ2 = sum (UNQ2, 1) ;
         u2.add() ;
      end ;
      h.replace() ;
   end ;

   h.output (dataset: "hash_agg2") ;
   stop ;
run ;
```

Note that the aggregates for both VAR and VAR2 are still calculated in a single pass through the input file, while the sort-control-break (implicit or explicit) would obviously require resorting DETAILS by [KEY,VAR2] and mean at least a fourfold increase in I/O traffic - plus temporary disk storage. The price of single-pass luxury is much higher RAM usage, for after the last input record has been read, tables H, U, and U2 will have contained all the unique keys and satellites with which they are defined. This issue will be specifically addressed later.

Other approach to handling multiple analysis variables, for example, by logically transposing the data is addressed in the online-version of this paper.

SORT-CONTROL-BREAK METHOD VERSUS TABLE LOOK-UP METHOD

The principal difference between the sorting method and look-up table method lies in the ways they use computer resources.
Sorting requires relatively small use of RAM but quite significant disk or tape storage for internal bookkeeping, typically 3 times the size of DETAILS data set. This is the very reason why from early to mid-age computing, sorting coupled with control-breaks had been the prevalent aggregation technique. Up until a certain, relatively recent, point, RAM storage had been extremely expensive and/or insufficient to hold large look-up tables. At the same time, efficient algorithms had been developed (merge sort, quick sort, heap sort, etc.) to facilitate sorting using very small amount of RAM. Even though in some extreme cases, input requires repeated passes and storing chunks of temporary data on tapes (most notably, on the mainframe), it *still* can be done - at the expense of the execution time, of course.

Table look-up, on the other hand, requires no disk space storage beyond that necessary to write data set HASH_AGG. Also, the task is accomplished in a single pass through data set DETAILS which fosters better performance in addition to not having to sort. However, it does require enough RAM to hold both

tables H and U. Depending on the cardinality of variables KEY and VAR (or their composite equivalents), the RAM footprint can range from insignificant to overwhelmingly important. With big data - understood in terms of the key cardinality, total variable length, and number of table entries - this may become a paramount issue. We shall dwell on its details later.

WHY USE THE SAS HASH OBJECT FOR AGGREGATION?

There are good reasons why the hash object is currently the most convenient tool for organizing the mechanics of aggregation based on table look-up:

- Thanks to its internal structure and efficient underlying code, the hash object's search (*check method*), extract (*find method*), insert (*add method*), and update (*replace method*) table operations are very quick.
- All these operations are facilitated in *O(1)* time. Hence, their performance does not depend on the hash table size.
- The associated *hash iterator object* supports the enumeration operation in *O(N)* time. Since each act of enumeration is equivalent to the extract operation, the time needed to get already computed aggregate information from the table for each unique key (*first, next, last, and prev iterator methods*) is just as quick. O(N) simply means that the time is the same for every key irrespective of the table size.
- The properties listed above make the hash object very machine-efficient as a result.
- Unlike some other aggregation methods based on table look-up, the hash object's key and data portions carry only the baggage necessary for the task at hand. For example, using the hash object to calculate simple statistics like SUM and N results in about twice the speed and half the RAM load compared to the SUMMARY procedure. This happens because (a) the hash object can use more than one AVL tree to store its entries and (b) it does not need to store information not immediately related to the task. (In addition, the procedure cannot be used to compute "count distinct", while the hash object can.)
- Aggregating with the hash object is programmer-efficient, as the gory details of the algorithms on which its operations are based are hidden behind syntactically simple method calls. By contrast, implementing a table with similar properties overtly (for example, using arrays) requires plenty of sophisticated SAS code at a very advanced level.
- The hash object being part of the DATA step lends itself to a good deal of programming flexibility coupled with the ability to perform, in addition to aggregation, other data processing tasks without crossing step boundaries and/or the necessity to store intermediate results on disk. In certain cases you may be able to prepare input to be aggregated within the same step where aggregation occurs. For example, if extra keys need to added to the aggregation process via table look-up not directly related to the process, it can be done via another hash table or format search. This ancillary flexibility dovetails into the next point.
- There exist situations where input data are structured in such a manner that neither the explicit sort-control-break technique nor an implicit one (e.g. via the SQL procedure sorting behind-the-scenes) can be realistically employed to aggregate data. For example, the process may take practically forever or be too taxing from the standpoint of machine resources to make it viable; yet using the hash object as an alternative can save the day. We will examine an example of such a situation below.

"SAS HASH OBJECT TO RESCUE"

Let us describe a situation alluded to in the two last items of the list above. Imagine that in addition to the file DETAILS, we have another file NEWKEY:

```
data NEWKEY ;
   input KEY NEWKEY ;
cards ;
1   1
1   2
1   3
2   1
2   2
2   3
2   4
2   5
;
run ;
```

Suppose that now we need to somehow add variable NEWKEY to the aggregation process for every matching KEY in file DETAILS and summarize by a new compound key [KEY, NEWKEY]. With miniscule files like our sample data sets DETAILS and NEWKEY it can be done, for example, using SQL:

```
proc sql ;
   create table sql_agg_newkey as
   select KEY
        , NEWKEY
        , sum (var) as SUM
        , count (distinct VAR) as UNQ
   from   (select a.KEY, a.VAR, b.NEWKEY from DETAILS a, NEWKEY b
           where  a.KEY = b.KEY)
   group  KEY, NEWKEY
   ;
quit ;
```

Another way would be to sort the files beforehand, match-merge them, and aggregate the result using the sort-control-break method. Both methods require sorting, either explicitly (sort-control-break) or in the background (SQL). In real life, it can be prohibitively expensive. For example, the authors have encountered a number of situations when real-life equivalents of data set DETAILS have contained billions of records and, instead of simple KEY, had wide composite keys with a dozen or more components. Under such circumstances, a frontal SQL or match-merge attack would have usually failed because of insufficient application server resources. Some typical use-cases for this facility the authors have encountered relate to processing of historical transaction data, e.g.:

1. Point-of-Sale data where there is a need to look up the segment to which a consumer has been assigned.
2. Insurance claims data where there is a need to create aggregates using categories (potentially overlapping and not exhaustive) to which a transaction belongs.

However, there is an even bigger problem lurking beneath the surface - namely, a quasi-Cartesian explosion. If there are billions of records in DETAIL and many values of NEWKEY for every KEY, preparing input for aggregation requires creating a file an order of magnitude or more (depending on the

content of NEWKEY) larger than DETAILS. If we let the internal mechanisms of SQL handle it, it will end up as a quasi-Cartesian explosion into temporary space file and sorting it behind-the-scenes before actual aggregation can be done. If unique count aggregates were requested for different variables (such as VAR and VAR2 in the example above), it would exacerbate the problem even further by the need to re-sort. Basically, such a straightforward sledge-hammer SQL approach could mean a dead end even for a sizeable enterprise SAS server. Needless to say, an explicit sort-control-break attack could meet a similar fate.

The advantage of using the hash object in this situation is its DATA step flexibility. There is no need to prepare the exploded input file beforehand because extra keys for aggregation can be added on the fly by searching some RAM incarnation of file NEWKEY for each KEY read from DETAILS. The simplest way to do it is to store NEWKEY in another hash table keyed by KEY, as in the following example:

Example 4

```
data _null_ ;
   dcl hash X  (dataset: "NEWKEY", multidata: "Y") ;
   x.definekey ("KEY") ;
   x.definedata ("NEWKEY") ;
   x.definedone () ;

   dcl hash H (ordered: "A") ;
   h.definekey  ("KEY", "NEWKEY") ;
   h.definedata ("KEY", "NEWKEY", "SUM", "UNQ") ;
   h.definedone () ;
   dcl hash U () ;
   u.definekey ("KEY", "NEWKEY", "VAR") ;
   u.definedone () ;

   do until (end) ;
      set DETAILS end = end ;
      call missing (NEWKEY) ;
      do rc = x.find() by 0 while (rc = 0) ;
         if h.find() ne 0 then call missing (SUM, UNQ) ;
         SUM = sum (SUM, VAR) ;
         if u.check() ne 0 then do ;
            UNQ = sum (UNQ, 1) ;
            u.add() ;
         end ;
         h.replace() ;
         rc = x.find_next() ;
      end ;
   end ;

   h.output (dataset: "hash_agg_newkey") ;
   stop ;
run ;
```

Above, file NEWKEY is stored by KEY as a multi-data (duplicate key) hash table X. For each row read from DETAILS, table X is searched for KEY and, if a match is found, retrieves every NEWKEY value for this KEY into its respective PDV host variable, thus making it available as part of the composite key [KEY,NEWKEY] for hash tables H and U. Such an approach obviates the need to create "exploded" input,

for the new key variable is added in the process. Of course, it comes at the cost (nothing is free!) of looking for KEY in hash table X and cycling through the entries with the same KEY value. However, if our experience is any indication, the searching and cycling operations are extremely efficient and quick, and resorting to them is certainly worth attaining the goal within the limit of the machine resources and saving the day.

HASH MEMORY MANAGEMENT

The tallest hurdle on the way of using the hash object as a replacement of other aggregation methods is its RAM usage. In industrial practice, as the number of composite keys' components and their cardinality grow, the memory footprint imposed by the hash object(s) can reach into the territory of tens and even hundreds gigabytes. The situation is aggravated by the fact that a given hash object runs on a single CPU core, and therefore the system must be configured to be able to allocate lots of RAM to that single core. Albeit it is usually feasible (at least under UNIX/Linux systems), it requires certain administrative measures, red tape, and money for extra memory. Nowadays, however, the latter is relatively inexpensive. If it is vital for a (large) business to make decisions based on frequently aggregated data, it is well worth a few thousand dollars invested in more RAM, all the more that it makes the enterprise server as a whole much more responsive for other processes unrelated to hash-based aggregation.

That having been said, it is irresponsible on part of the hash coder (or any other coder, for that matter) to merely throw code that has not been optimized at the machine hoping that the hardware will handle it. In real-life scenarios of using the hash object for aggregation, no effort should be spared to reduce its RAM footprint - not only out of common programming ethics but also because in cases of high-cardinality, long-composite-key input, it can spell the difference between success and failure.

PARTIALLY PRE-SORTED HIGH-LEVEL KEY(S)

In many real-life cases, input is pre-sorted by a high-level key component(s) of the compound key on which aggregation is to be based. Making use of this circumstance may greatly reduce the amount of RAM necessary to keep the keys and aggregated values. This is because in this case, the aggregates can be calculated separately for each by-group, and RAM usage by the hash objects involved in the calculation can be limited only to that needed to hold the information corresponding to the largest by-group. As an example, let us construct a data set similar to DETAILS but with another high-level key ID:

```
data DETAILS3 ;
   input ID:$1. KEY VAR VAR2 ;
cards ;
A  2   2   5
A  1   1   3
A  1   2   4
A  2   1   7
A  1   2   3
A  2   2   7
A  2   3   5
A  1   1   2
B  2   2   5
B  1   1   3
B  2   1   7
B  1   2   4
B  2   2   7
```

```
B  1   2   3
B  1   1   2
B  2   3   5
;
run ;
```

For each ID group, the content is exactly the same as in DETAILS2, except that it is not pre-ordered by KEY. Now suppose that we need to perform the same aggregation as in Example 4, but the aggregation must be grouped by composite key {ID,KEY]. We can merely modify the code from Example 4 by including the extra key ID in front of KEY in the hash table definitions - and it will work. However, it will also mean that after the last input record has been processed, the hash tables will contain an entry per each unique value of the [ID,KEY] pair.

Let us observe, however, that DETAILS3 is intrinsically pre-sorted by ID, which lends itself to partial by-processing at the ID level. Hence, we can proceed as follows:

- Before the first record from each ID by-group is read in, clear the hash tables
- Compute the aggregates for this by-group aggregating by KEY only
- Output then content of table H using an associated iterator
- Proceed to the next ID by-group

With that in mind, let us take advantage of the intrinsic pre-sorting by ID:

Example 5

```
data hash_agg_presorted (keep = ID KEY SUM UNQ SUM2 UNQ2) ;
   if _n_ = 1 then do ;
      dcl hash H (ordered: "A") ;
      dcl hiter I ("H") ;
      h.definekey  ("KEY") ;
      h.definedata ("KEY", "SUM", "UNQ", "SUM2", "UNQ2") ;
      h.definedone () ;

      dcl hash U () ;
      u.definekey ("KEY", "VAR") ;
      u.definedone () ;

      dcl hash U2 () ;
      u2.definekey ("KEY", "VAR2") ;
      u2.definedone () ;
   end ;
  *clear hash tables ;
   h.clear() ;
   u.clear() ;
   u2.clear() ;
  *loop over next ID by-group ;
   do until (last.ID) ;
      set DETAILS3 ;
      by  ID ;
      if h.find() ne 0 then call missing (SUM, SUM2, UNQ, UNQ2) ;
      SUM  = sum (SUM, VAR) ;
      SUM2 = sum (SUM2, VAR2) ;
      if u.check() ne 0 then do ;
```

```
         UNQ = sum (UNQ, 1) ;
         u.add() ;
      end ;
      if u2.check() ne 0 then do ;
         UNQ2 = sum (UNQ2, 1) ;
         u2.add() ;
      end ;
      h.replace() ;
   end ;
  *output aggregates for current ID by-group ;
   do while (i.next() = 0) ;
      output ;
   end ;
run ;
```

This code is remarkably similar to that in Example 3 - and it should be no surprise because it follows the same general scheme. Only now the *DoW-loop*, instead of cycling through the entire input file (think of it as a single by-group), cycles through each ID by-group consecutively, clearing the hash tables before and adding aggregated output after. From the standpoint of the intended reduction of hash RAM footprint, it has two consequences:

- The hash tables don't have to contain entries with all distinct values of the [ID,KEY] pair but only those for KEY alone. Thus their RAM footprint is driven, not by aggregate output of the entire file, but the largest aggregate size among all ID by-groups.
- The extra key ID, by which input is pre-sorted, is absent from both the key portion and the data portion of the hash tables. It further reduces the load on RAM by shortening the hash table entries by two ID lengths. Obviously, the more high-level keys are included in the pre-sorted order, the better.

The added IF _N_=1 logic (absent from Example 3) is needed because with this arrangement, program control reached the top of the implied loop after each ID by-group. The program would still work without it, however it would be also destroying and recreating the hash tables before each by-group, which is orders of magnitude more computationally expensive that merely clearing their contents.

SHORTENING HASH ENTRIES USING THE MD5 FUNCTION

When we aggregate grouping by a multi-term compound key and use any of the schemes we have discussed thus far, all components of the composite key must be included in the hash tables (except for the high-level pre-grouped keys as outlined in the previous section). For table H, it means both the key portion and the data portion; for the U-tables, it means the key portion only. In real-life situations, a long composite grouping key can easily result in a hash entry so wide that the hash object may run out of memory well before the aggregation process is finished.

It would be nice, therefore, if there existed a method allowing to *"scramble"* a long composite key into a single relatively short key entry whose values would have one-to-one relationship with those of the composite key. Fortunately, such a method exists. To illustrate the concept, let us first construct a fairly sizeable data set MULTIKEY with a fairly wide composite key [KN1,KN2,KN3,KC1,KC2,KC3]. The component keys KN are numeric and KC - character with the length of 16, 24, and 32 bytes.

```
data MULTIKEY (drop = X:) ;
   length KN1-KN3 KC1 $16 KC2 $24 KC3 $32 ;
   do KN1 = 1 to 3 ;
```

```
     do KN2 = 1 to 5 ;
       do KN3 = 1 to 7 ;
         do x1 = 1E15 to 1E15 + 4 ;
           do x2 = 1E15 to 1E15 + 6 ;
             do x3 = 1E15 to 1E15 + 8 ;
               KC1 = put (x1, z16.) ;
               KC2 = put (x2, z16.) ;
               KC3 = put (x3, z16.) ;
               do VAR = 1 to ceil (ranuni(1) * 10) ;
                 do x = 1 to ceil (ranuni(1) * 11) ;
                   output ;
                 end ;
               end ;
             end ;
           end ;
         end ;
       end ;
     end ;
   end ;
run ;
```

This step creates 1,090,367 records. Let us aggregate this file by the compound key [KN1,KN2,KN3,KC1,KC2,KC3] using the hash object to get the result completely identical to this SQL:

```
proc sql ;
  create table sql_agg_multikey as
  select kn1, kn2, kn3, kc1, kc2, kc3
       , sum (VAR)            as SUM
       , count (distinct VAR) as UNQ
  from   multikey
  group  1, 2, 3, 4, 5, 6
  ;
quit ;
```

Running this SQL step reveals that the cardinality of the compound key is 33,075. The following hash code is merely a slight variation of Example 2 where the simple key KEY is replaced with the composite key [KN1,KN2,KN3,KC1,KC2,KC3]:

Example 6

```
data _null_ ;
   dcl hash H (ordered: "A") ;
   h.definekey  ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3") ;
   h.definedata ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3", "SUM", "UNQ") ;
   h.definedone () ;

   dcl hash U () ;
   u.definekey ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3", "VAR") ;
   u.definedata ("_N_") ;
   u.definedone () ;

   do until (end) ;
      set MULTIKEY end = end ;
      if h.find() ne 0 then call missing (SUM, UNQ) ;
      SUM = sum (SUM, VAR) ;
      if u.check() ne 0 then do ;
         UNQ = sum (UNQ, 1) ;
```

```
         u.add() ;
      end ;
      h.replace() ;
   end ;

   h.output (dataset: "hash_agg_multikey") ;

   h_size = h.item_size ;
   u_size = u.item_size ;
   put h_size= u_size= ;
   stop ;
run ;
```

Variables H_SIZE and U_SIZE are here to tell us what the length of each hash table entry is in bytes and judge what kind of improvement in hash RAM usage can be achieved by using the MD5 function to scramble the composite key. Before explaining the concept of making use of it here, let us recall what the function does. It takes a character argument and generates a 16-byte character string whose value is related to the argument as one-to-one. In other words, for any value of the MD5 argument, there is one and only one unique response. (Well, almost. It has been proven that it is theoretically possible to break this uniqueness property of MD5, however for all practical intents and purposes it is more unlikely that for the molecules of air confined in a basketball to spontaneously collect in one of its halves. At least, as far as our application is concerned, it can never happen.) It means that if the cardinality of the argument is N, then the cardinality of the response if also N. The response is often called one-way hash signature because there exists no way to recreate the argument based on the response: Its original value is thus "scrambled". So, how can we use MD5? This is how:

- Concatenate the components of the composite key as CATS(KN1,KN2,KN3,KC1,KC2,KC3). This has an advantage of not worrying about whether the components are numeric of character. If their total length can exceed 200, a separate variable with a predefined length can be created as a result.
- Feed the result into MD5 as an argument. This way, no matter what the combined length of the composite key is, the response of the function, MD5=MD5(CATS(KN1,KN2,KN3,KC1,KC2,KC3)) will always be a 16-byte string.
- Key table H with MD5 variable instead of the composite key while keeping all composite key components and the aggregated variables SUM and UNQ in the data portion. Doing so will result in shortening the key portion to mere 16 bytes.
- Key table U with MD5 variable instead of the composite key and with variable VAR. Doing so will result in shortening the key portion to mere 16 bytes plus whatever length variable VAR has.

Now let us translate the plan into SAS code:

Example 7

```
data _null_ ;
   dcl hash H (ordered: "A") ;
   h.definekey  ("MD5") ;
   h.definedata ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3", "SUM", "UNQ") ;
   h.definedone () ;

   dcl hash U () ;
   u.definekey ("MD5", "VAR") ;
   u.definedata ("_N_") ;
```

```
   u.definedone () ;

   do until (end) ;
      set MULTIKEY end = end ;
      length MD5 $ 16 ;
      MD5 = MD5 (cats (of KN:, of KC:)) ;
      if h.find() ne 0 then call missing (SUM, UNQ) ;
      SUM = sum (SUM, VAR) ;
      if u.check() ne 0 then do ;
         UNQ = sum (UNQ, 1) ;
         u.add() ;
      end ;
      h.replace() ;
   end ;

   h.output (dataset: "hash_agg_multikey_md5") ;

   h_size = h.item_size ;
   u_size = u.item_size ;
   put h_size= u_size= ;
   stop ;
run ;
```

The PUT statements and FULLSTIMER option present the following picture in the SAS log (on the X64_7PRO platform; on other platforms the results are slightly different but not by much):

	Example 6 (no MD5)	**Example 7 (MD5)**
Table H entry length, bytes	240	144
Table U entry length, bytes	160	64
RAM usage, kilobytes	54,227	28,648

As we see, using MD5 to shorten the hash entries results in the approximately 50 per cent reduction in both total hash entry length and RAM usage. The authors have learned *the hard way* that in industrial, real data size applications of the technique where hash RAM usage can reach hundreds of gigabytes for a single aggregating DATA step, the twofold difference can be truly paramount. In fact, the real-life difference is often even greater because the total hash key length may far exceed the relatively puny key length used in these examples.

Multi-Scan Where Clause Split: The Concept

Reducing hash RAM usage by taking advantage of high-level key(s) pre-sorted order is possible because pre-sorting separated the input data into physically distinct parts, making sure that no two parts share the same key on which aggregation is based. Thus, we can aggregate within each by-group separately and add the results to the final aggregate file.

However, under practical circumstances the input file is often not pre-sorted in any way. If it also, as it frequently happens today with large data collections, too big and/or expensive to sort (explicitly or behind-the-scenes) , a frontal-attack attempt to aggregate it using the hash object may result in the latter running out of memory, even if good use is made of tricks of trade (such as the MD5 method described above) to shorten the length of the hash entries.

The question is, is there a way out of such a seemingly dead-end situation? The answer is yes, and it is based on the following idea. Suppose that we can somehow theoretically (not physically) separate the unsorted input records into *more or less equal* groups of records not sharing each other's keys. For example, we may know a priori (or by running a frequency) that (a) the key takes on a few specific values and (b) each value of the key is found in more or less the same number of records. For instance, imagine a data set similar to DETAILS above but completely disordered with respect to KEY:

```
data DETAILS ;
   input KEY VAR ;
cards ;
2   3
1   2
2   2
1   2
2   1
1   1
2   2
1   1
;
run ;
```

We know, however, from its structure that it has approximately 1/2 of the records with KEY=1 and KN1=2, each. Armed with this knowledge, we can proceed as follows:

1. Read only the records with KEY=1, aggregate them, and add the results to the output using the hash iterator.
2. Do the same for KEY=2.

Translated into SAS, it may look like this (for simplicity sake, only the aggregates for VAR are calculated):

Example 8

```
data hash_agg_where (keep = KEY SUM UNQ) ;
   if _n_ = 1 then do ;
      dcl hash H (ordered: "A") ;
      dcl hiter I ("H") ;
      h.definekey  ("KEY") ;
      h.definedata ("KEY", "SUM", "UNQ") ;
      h.definedone () ;
      dcl hash U () ;
      u.definekey ("KEY", "VAR") ;
      u.definedone () ;
   end ;

   h.clear() ;
   u.clear() ;
   do end = 0 by 0 until (end) ;
      set UNSORTED (WHERE = (KEY = 1)) end = end ;
      if h.find() ne 0 then call missing (SUM, UNQ) ;
      SUM  = sum (SUM, VAR) ;
      if u.check() ne 0 then do ;
         UNQ = sum (UNQ, 1) ;
         u.add() ;
```

```
      end ;
      h.replace() ;
   end ;
   do while (i.next() = 0) ;
      output ;
   end ;

   h.clear() ;
   u.clear() ;
   do end = 0 by 0 until (end) ;
      set UNSORTED (WHERE = (KEY = 2)) end = end ;
      if h.find() ne 0 then call missing (SUM, SUM2, UNQ, UNQ2) ;
      SUM  = sum (SUM, VAR) ;
      if u.check() ne 0 then do ;
         UNQ = sum (UNQ, 1) ;
         u.add() ;
      end ;
      h.replace() ;
   end ;
   do while (i.next() = 0) ;
      output ;
   end ;
run ;
```

Effectively, we have cut hash RAM memory usage by half, just as we would if the file were pre-sorted by KEY and we made use of it as in Example 5. A few observations about this technique are due:

- The more distinct key values are involved in this repetitive process, the more RAM we save - provided that the values are distributed more or less uniformly throughout the input data set.
- At the same time, the more such values to rely upon we have, the more passes through the input file we have to make. However, this negative effect is mitigated by using the WHERE clause, so the filtered-out observations are never moved from the buffer for processing. The great advantage we gain is that is definitely doable, and a few passes with the WHERE clause are certainly less taxing that sorting the whole thing, if it is even possible. (A variant of this method, more general than the simple case at hand, will be presented later on.)
- Needless to say, it is not necessary to write repetitive code to do this: It can be done instead by writing a simple macro or using other methods of dynamic code generation (e.g. PUT-%INCLUDE). Above, it was done explicitly to convey an idea how the assembled code may look like.

MULTI-SCAN WHERE CLAUSE SPLIT: REAL LIFE

A simple example presented above is good to illustrate the concept. Data aggregation life, however, is almost never as simple as the luck of having a whole key with a few, uniformly distributed values. On the other hand, the applicability of the technique is not limited to such a case. To understand why, let us observe that our only really necessary and sufficient criteria for being able to aggregate separate chunks of input independently and save hash memory are:

1. No two chunks must share the same key
2. All chunks should be of roughly equal size
3. The number of the chunks should not be too high

Let us consider them separately:

1. Two composite keys cannot be identical if any one of their components is different between them. Therefore, any partial key is good if it can be used to satisfy criteria (2) and (3). Moreover, if two keys - simple or composite - differ even in a single same-position byte, they are different. Hence, the required key separation can be achieved based on any byte (or bytes), as long as its values lend themselves to meeting criteria (2) and (3).
2. This criterion is met when the values of the component (see criterion 1), on which the chunks are segregated, are distributed more or less uniformly in the input file. Oftentimes, business knowledge can help. For example, a character or digit in a customer identifier (e.g. a bank account, phone number, patient ID, etc.) may be limited to several values specifically intended to be evenly distributed amid the customer base. If so, the WHERE clause can be based on it.
3. The number of chunks into which input is "split" is a matter of reasonable balance between the desirable RAM load reduction and the number of passes through the input file for each value of the component, on which segregation is based, fed into the WHERE clause. To wit, if the number of chunks is 1000, say, it will sure reduce hash objects' RAM usage accordingly, but it will lead to 1000 passes and a mountain of generated code and at the same time may result in memory underutilization and poorer performance. (One exception to this line of thought is the case when the component is an *indexed* partial key, as it would make each pass blisteringly quick, and the enormous SAS log generated by the process may be a price easy to pay.) On the other hand, if there are 10 chunks only and the system can comfortably handle enough RAM for each, there is no reason to opt for more, and the aggregation process will be more efficient.

Consider for example, the Point-of-Sale use case referenced above. The authors had determined that the last character of the consumer ID was roughly equally split between 0 thru 9. Thus, a macro loop could be used to read approximately 10% of the input data at a time, thus cutting the size of the hash tables by a factor of 10. And if more reduction is needed, the last two digits could be used to get a 100 fold reduction. The use of a WHERE clause to read the subsets introduced very little additional overhead, thus making the RAM savings worthwhile.

MULTI-SCAN WHERE CLAUSE SPLIT: REAL LIFE MD5 PLUS

It is all good and well, but what if no component can be found in the aggregation key that would satisfy the criteria listed above? Fortunately, not only the situation can be remedied but it is possible to both ensure the approximate equality of the chunks and control their number at will. The MD5 function, again, is to rescue, but at a different angle. In order to understand its potential, let us run this test:

```
data MD5_TEST ;
  do KEY = 1 to 999999 ;
     md5_byte1 = put (md5 (put (KEY, z6.)), $hex2.) ;
     output ;
  end ;
run ;
proc freq data = MD5_TEST noprint order = freq ;
  tables md5_byte1 / out = MD5_BYTE1_FREQ ;
run ;
```

Essentially, it creates 999,999 distinct non-random character keys from "000001" to "999999" and looks at the distribution of the first byte of the MD5 signature. If you look at the content of data set MD5_BYTE1_FREQ, you will see that the first byte of the MD5 response is shared almost equally by all the different keys: Its frequency percent falls in the narrow range of 0.373 to 0.405, with the minimum of 3,734 keys (MD5 byte "B8"x) to the maximum of 4,049 keys (MD5 byte "0B"x). The picture is practically

the same, whether we pick the first, second, or any other byte of the MD5 response, which should be no surprise since MD5 in itself is a *hash function* designed to distribute disparate arguments randomly. Note that the first byte of the MD5 response distributes the keys almost equally among all 256 characters available in the collating sequence. It means that if we should leave it to its own devices and base the process of segregation input into chunks based on the first byte of the MD5 response, we would get up to 256 chunks, depending on the input keys. If it is too many and we would like fewer, we could base the separation criteria on several ranges of MD5_BYTE1 rank. For example, if we wanted 4 chunks, we could merely feed the ranks of 0-63, 64-127, 128-191, 192-255 into 4 consecutive WHERE clauses. If we wanted more chunks, we could base segregation on the first *two* bytes of the MD5 signature, to the avail of 65,536 chunks - which is quite unlikely to be necessary under any circumstances. On the other hand, if we wanted a more uniform chunk distribution, we could still base segregation on the two first bytes instead of one; but, as with the first byte only, simply select a desired number of ranges by dividing 65,536 equally into whatever number of ranges we would want. This concept can be extrapolated to making use of the first 3 bytes (16,777,216 possible values), 4 bytes (4,294,967,296), and so forth. In fact, we could pick up those bytes from anywhere inside the MD5 signature, and they do not have to be consecutive, either. Note that the endpoints of the rank boundaries can be safely hard coded into the WHERE clauses since they are not input-driven.

Let us illustrate the technique by aggregating data set MULTIKEY based on the first byte of the composite key MD5 signature:

Example 9

```
%Macro MD5 (n_chunks) ;
   %do x = 0 %to %eval (&n_chunks - 1) ;
     %let lo = %eval (64 * &x) ;
     %let hi = %eval (64 * (&x + 1) - 1) ;
     h.clear() ;
     u.clear() ;
     do end = 0 by 0 until (end) ;
        set MD5VIEW end = end ;
        WHERE MD5_BYTE1 between &lo and &hi ;
        if h.find() ne 0 then call missing (SUM, UNQ) ;
        SUM = sum (SUM, VAR) ;
        if u.check() ne 0 then do ;
           UNQ = sum (UNQ, 1) ;
           u.add() ;
        end ;
        h.replace() ;
     end ;
     do while (i.next() = 0) ;
        output ;
     end ;
   %end ;
%mEnd ;

data md5view / view = md5view ;
   set MULTIKEY ;
   MD5_BYTE1 = rank (char (md5 (cats (of KN:, of KC:)), 1)) ;
run ;

data hash_agg_multikey_chunks (keep = KN: KC: SUM UNQ) ;
```

```
   if _n_ = 1 then do ;
     dcl hash H () ;
     dcl hiter I ("H") ;
     h.definekey  ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3") ;
     h.definedata ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3", "SUM", "UNQ") ;
     h.definedone () ;
     dcl hash U () ;
     u.definekey ("KN1", "KN2", "KN3", "KC1", "KC2", "KC3", "VAR") ;
     u.definedata ("_N_") ;
     u.definedone () ;
   end ;
   %MD5 (4)
run ;
```

Looking at the SAS log, we see that it reports:

```
NOTE: There were 272153 observations read from the data set WORK.MD5VIEW.
      WHERE (MD5_BYTE1>=0 and MD5_BYTE1<=63);
NOTE: There were 274724 observations read from the data set WORK.MD5VIEW.
      WHERE (MD5_BYTE1>=64 and MD5_BYTE1<=127);
NOTE: There were 266420 observations read from the data set WORK.MD5VIEW.
      WHERE (MD5_BYTE1>=128 and MD5_BYTE1<=191);
NOTE: There were 277070 observations read from the data set WORK.MD5VIEW.
      WHERE (MD5_BYTE1>=192 and MD5_BYTE1<=255);
```

In spite of the fact that there is nothing special about the selection of the keys in MULTIKEY, the input records have been grouped into almost equal chunks, none sharing any keys with the others. Also, memory utilization is reported to be mere 15,698 kilobytes, down from 54,227 kilobytes with the sledge-hammer attack. This is an almost fourfold reduction, even though no provision has been made to shorten the hash entries via another incarnation of MD5 discussed earlier. Had it been combined with the file split method, memory usage would have been reduced further by another half.

CONCLUSION

The SAS hash object is a powerful, quick, and flexible in-memory table look-up tool that lends itself to data aggregation in situations where other tools in the SAS arsenal may be inefficient or incapable of doing the job, in particular with massive unordered input data and high cardinality of grouping keys. That said, learning how to manage the hash object's memory usage is a must.

ACKNOWLEDGEMENTS

The authors would like to thank everyone on the SAS Base R&D team who have contributed to the development of the wonderful thing known as the SAS hash object; among others, to Paul Kent, Robert Ray, Jason Secosky, and Al Kulik. They would also like to thank each other for long-time intellectual camaraderie, without which the techniques presented in this paper could have never been developed and tested in the unforgiving world of real data - and indeed the paper itself would have never happened.

RECOMMENDED READING

Anything in the SAS literature regarding the SAS hash object and the DoW-loop.

CONTACT INFORMATION

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

Paul M. Dorfman
Independent Consultant, Proprietor
4437 Summer Walk Ct
Jacksonville, FL 32258
Phone: (904) 226-0743
Email: sashole@gmail.com

Don Henderson
Henderson Consulting Services, LLC
3470 Olney-Laytonsville Road, Suite 199
Olney, MD 20832
Work phone: (301) 570-5530
Fax: (301) 576-3781
Email: Don.Henderson@hcsbi.com
Web: http://www.hcsbi.com
Blog: http://hcsbi.blogspot.com
http://www.sascommunity.org/wiki/User:Donh
http://www.sascommunity.org/wiki/Presentations:Donh_Papers_and_Presentations