HASH + POINT = KEY

Paul M. Dorfman, Independent SAS Consultant, Jacksonville, FL
Lessia S. Shajenko, Senior Quantitative Analyst, VP, Bank of America, Boston, MA

ABSTRACT

One of the most trivial applications of the SAS® hash object (as well as one of its most often cited claims to fame) is its application as a fast-searched look-up table for joining two SAS files. Typically, a smaller "look-up" file, complete with its keys and satellites, is loaded into a memory-resident hash table, and then a larger "driver" file is read sequentially. For each record read from the driver, the hash table is searched, and some action is taken depending on whether or not the key is found in the look-up table. This scheme typically yields join times well ahead of other methods, such as sort and merge, SQL join, SAS index or SAS format searches. However, the inherent limitation of the hash object lies in its RAM-resident nature, especially when the non-key hash variables are numerous and/or long. In this paper, the problem of such hash memory overload is abated by resorting to a hybrid-indexing scheme, whereby only the keys and file pointers are stored in the hash, whilst the satellites are fetched from the disk-resident look-up file via the POINT= SET statement option. Two other, conceptually analogous, but more programmatically sophisticated, uses of the technique are proposed and elaborated upon.

INTRODUCTION

The SAS hash object, introduced first in Version 9.0, had by the end of Version 9.2 become a very mature and feature-rich DATA step programming tool. The object, coupled with its sibling the hash iterator, allows for kind of run-time, dynamic SAS programming unimaginable before its advent.

Besides being flexible, the hash object returns very quick performance against all the standard table storage and retrieval operations: Check (discover if a key is in the table), Find (search for a key in the table and, if found, retrieve the associated data), Add (insert a key and the associated data into the table), Replace (check if a key is in the table and, if yes, overwrite the associated data), and Remove (find if a key is in the table and, if yes, delete the key and the associated data). All these operations are executed by the hash object in O(1) time. In layman terms, the latter simply means that the time needed to perform any of these operations *does not depend on the number of entries* stored in the object. (Contrast that to a table accessed sequentially, where all such operations except the Add operation take time proportional to the number of entries in the table.)

The hash object owes its nimble nature to 2 main factors:

1. The speed of the underlying hybrid algorithm ('true' hashing coupled with AVL-tree bucket storage)
2. The fact that the hash object stores and manipulates its keys and data entirely in memory (RAM)

However, the strength of the object's memory-resident nature is also its vulnerability , for although RAM has in the last decade become immensely more voluminous and less expensive, it is still quite limited compared to disk storage, especially on multi-user machines, where large volume/file data processing is mostly concentrated. Therefore, it is imperative that the total byte volume of the keys and data stored in a hash object must stay below the RAM threshold available for the SAS job.

In many situations, however, the total size of hash keys and data is too large to fit in RAM. At the same time, the 'standard' recipe for the hash object usage – as described in the SAS documentation – invariably prescribes to define and load all the data (i.e. non-key) fields into the object, if they are to exchange values with their PDV host fields. The data

portion of the object often contains satellite information, such as business descriptive fields, which tends to occupy much larger storage than the key portion. Placing this satellite information alongside the keys in a hash table, particularly with a large number of entries, can very well cause the object to run out of memory and fail.

The remedy usually suggested to alleviate this kind of problem is to ditch the hash object as a programming tool altogether in favor of a method based on disk-, rather than on memory-resident storage - for example, an indexed SAS file, sort-and-merge, or SQL. In this paper, we suggest that in many situations of this nature it is not really necessary to throw the baby out with the bathwater.

First, we shall present a concept of how to keep the baby. Next, we shall illustrate it by hopefully comprehensible annotated programmatic examples.

HASH + POINT = KEY: THE CONCEPT

Instead of placing all the keys and satellites from the 'look-up' SAS file in the hash data portion we can load only the keys and an integer pointer to the record on disk (routinely termed Record Identifier, or RID), where the satellites are stored to begin with. Thereafter at a later time, whenever the satellites corresponding to a key need to be brought into the PDV, we will be able to retrieve them from disk using the direct-access SET statement with the POINT= option.

Since RID, being a SAS numeric field, occupies only 8 bytes per hash entry, placing it into the object instead of the satellites can potentially save a good deal of memory. Suppose, for instance, that we have a SAS look-up file with 10,000,000 unique numeric (thus 8-byte long) keys and two 200-byte long character satellite variables. If we store both the keys and the satellite data in the object, they will together occupy 3.8 Gb of RAM. However, if we store only the keys and the RID pointer, they will take up only 0.15 Gb, i.e. approximately 26 times less memory space. Nowadays, using 150 Mb for hash table storage will not present any problem even for a netbook, whilst a request for 3.8 Gb may present a challenge, especially in a multi-user and/or strictly administered computing environment, such as a mainframe.

LEFT-JOIN VIA HASH LOOK-UP

As pointed out in the abstract, this is the most trivial, as well as perhaps most well known and frequently undertaken task for the hash object's application. At the advent of the construct in Version 9.0, many long-time SAS programmers were impressed by the ability to match two or more files by a common key without the need to sort any of the files explicitly (as with match-merge) or implicitly (as with SQL).

To illustrate the problem, let us first concoct some test data:

```
data DRIVER (keep = KEY     )
     LOOKUP (keep = KEY SAT:)
   ;
   do KEY = 5, 1, 4, 3, 2 ;
      do _N_ = 1 to ceil (ranuni (1) * 3) ;
         SAT1 = put (ranuni (1), hex16.) ;
         SAT2 = put (ranuni (1), hex16.) ;
         output DRIVER ;
         if KEY in (1, 3, 5) then output LOOKUP ;
      end ;
   end ;
run ;
```

If printed out, file DRIVER would look as follows:

```
KEY
---
 5
 1
 4
 4
 3
 3
 2
```

And file LOOKUP would look thus:

```
KEY          SAT1                SAT2
-----------------------------------------
 5     3FEF0AF77D7E15EF    3FD996B8AFB32D71
 1     3FED7DC4B07AFB89    3FEF0451EEBE08A4
 3     3FEB4F0195F69E03    3FB1330228226604
 3     3FEE9FF07F3D3FE1    3FD30539D5260A74
```

What we need to do is to 'left-join' file LOOKUP to file DRIVER, so that the output file would look like this:

```
KEY          SAT1                SAT2

 5     3FEF0AF77D7E15EF    3FD996B8AFB32D71
 1     3FED7DC4B07AFB89    3FEF0451EEBE08A4
 4
 4
 3     3FEB4F0195F69E03    3FB1330228226604
 3     3FEE9FF07F3D3FE1    3FD30539D5260A74
 3     3FEB4F0195F69E03    3FB1330228226604
 3     3FEE9FF07F3D3FE1    3FD30539D5260A74
 2
```

Since the KEY values 2 and 4 are not present in LOOKUP, the corresponding SAT variables in the result are expected to be missing, as shown above. Also note that the entries with KEY=3 in file LOOKUP are duplicate with different values of the SAT variables, and so the code is expected to gather both values for each of the rows in file DRIVER where KEY=3. Therefore, 2*2=4 rows with KEY=3 are expected in the result.

The 'standard' way to accomplish the task using the hash object can be represented as follows:

```
data LEFT_JOIN ;
   dcl hash H   (hashexp: 16, multidata: 'Y') ; /*declare hash, allow for duplicate keys   */
   H.definekey  ('KEY') ;                       /*key table with KEY variable             */
   H.definedata ('SAT1', 'SAT2') ;              /*store satellites in data portion        */
   H.definedone () ;                            /*check validity and instantiate object   */

   do until (l_eof) ;                           /*use explicit DO to process LOOKUP file  */
      set LOOKUP end = l_eof ;                  /*read next record from LOOKUP            */
      h.add() ;                                 /*insert current row from file into hash  */
   end ;                                        /* at end of loop, hash has been loaded   */

   do until (d_eof) ;                           /*use explicit DO to process DRIVER file  */
      set DRIVER end = d_eof ;                  /*read next record from DRIVER            */

      _iorc_ = h.find() ;                       /*find if KEY from DRIVER is in hash table */
                                                /*if yes update PDV SAT fields from hash  */
      if _iorc_ ne 0 then do ;                  /*if not then return code is not zero, so */
         call missing (of SAT:) ;               /*fill PDV SAT host fields with nulls     */
```

```
         output ;                                /*and write record to output file          */
      end ;
      else do while (_iorc_ = 0) ;               /*else key is found launch a DO while loop */
         output ;                                /*output record with hash values of SAT    */
         _iorc_ = h.find_next() ;                /*if keys are duplicate, harvest next row  */
      end ;                                      /*if no more dup keys exit DO loop         */
   end ;

   stop ;
run ;
```

Note that instead of loading the hash table by explicitly reading file LOOKUP, we could have simply given the file name to the DATASET: argument tag in the DCL instruction. It would have made the code more parsimonious but would not have changed anything principally.

Now imagine that a real-world file LOOKUP is so large that memory shortage would prevent the hash table from being loaded with the SAT variables alongside KEY, yet we still want to use the hash object for KEY look-up! The workaround, as noted above, is to leave the SAT variables in their original place on disk and instead, load a file record identifier variable RID into the data portion of the hash table H:

```
data LEFT_JOIN ;
   dcl hash H   (hashexp: 16, multidata: 'Y') ; /*declare hash, allow for duplicate keys   */
   H.definekey  ('KEY') ;                       /*define KEY as hash object key           */
   H.definedata ('RID') ;                       /*define RID as hash object data field    */
   H.definedone () ;                            /*check validity and instantiate object   */


   if 0 then set LOOKUP ;                       /*Tell compiler what SAT1, SAT2 fields are */

   do RID = 1 by 1 until (l_eof) ;              /*add 1 to RID for each next LOOKUP record */
      set LOOKUP (keep = KEY) end = l_eof ;     /*read next record from LOOKUP - just KEY! */
      h.add() ;                                 /*add KEY and RID to hash object           */
   end ;                                        /*end loop - hash table has been loaded    */

   do until (d_eof) ;                           /*use explicit DO to process DRIVER file   */
      set DRIVER end = d_eof ;                  /*read next record from DRIVER file        */

      _iorc_ = h.find() ;                       /*if KEY from DRIVER is in hash _iorc_=0   */
                                                /*and RID from hash updates its PDV host   */
      if _iorc_ ne 0 then do ;                  /*if KEY from DRIVER is not in hash        */
         call missing (of SAT:) ;               /*null out SAT fields in PDV               */
         output ;                               /*and output record                        */
      end ;
      else do while (_iorc_ = 0) ;              /*if KEY found in hash launch DO loop      */
         set LOOKUP point = RID ;               /*use RID from hash to read row with obs#  */
         output ;                               /*equal to RID from LOOKUP and output      */
         _iorc_ = h.find_next() ;               /*harvest next dup KEY entry from hash     */
      end ;                                     /*if no more dupes exit DO loop on top     */
   end ;

   stop ;                                       /*stop DATA step execution                 */
run ;
```

Of course, reading a record with the observation number equal to RID directly from disk using the POINT= option involves some I/O overhead compared to overwriting the SAT host PDV fields directly from the hash object memory. However, the overhead is first somewhat offset by the fact that while we are loading the table, we do not have to copy the SAT variables from disk in the first place – instead we are just keeping KEY. (Note that because of that, we use another SET LOOKUP statement for SAT1 and SAT2 parameter type matching, but because it is not executed at the run

time due to IF 0 condition, it does not affect performance.) Second, the work of copying presumably long satellites from the hash object to PDV still does not amount to absolute zero.

All in all, experiments with really voluminous files and long satellites (see, e.g. [10]) – as opposed to the smallish snippets used here for the purpose of demonstration - reveal that even with additional I/O incurred by the direct SET POINT= statement, the hash method still outperforms other left-joining method by significant margins.

INDIRECT FILE SORTING EMULATOR

Let us contrive a data set somewhat similar to file LOOKUP used above:

```
data UNSORTED (keep = KEY SAT1 SAT2) ;
   do KEY = 9, 0, 8, 1, 7, 2, 6, 3, 5, 4 ;
      do _N_ = 1 to ceil (ranuni (1) * 3) ;
         SAT1 = put (KEY + ranuni (1), hex16.) ;
         SAT2 = put (_N_ + ranuni (1), hex16.) ;
         output ;
      end ;
   end ;
run ;
```

This is how this file would look like if printed:

```
KEY          SAT1                SAT2
-----------------------------------------
 9      4023F0AF77D7E15F    3FF665AE2BECCB5C
 0      3FED7DC4B07AFB89    3FFF8228F75F0452
 8      40211039E5C22074    3FF0CBF4D22197EA
 8      4020221502D4442A    40068DF6E50D1BEE
 1      3FFDA780CAFB4F02    3FF1133022822660
 1      3FFF4FF83F9E9FF0    400260A73AA4C14E
 7      401EC27CE9C584FA    3FFFA0D4313F41A8
 2      400581822A1B0304    3FF69AAE48AD355C
 6      4019261E78724C3D    3FF79CD53F0F39AA
 6      401B6144441EC288    40051381618A2703
 3      400CA92080D95241    3FF6083F2BAC107E
 3      400DD3AF39BBA75E    40040D8626B81B0C
 5      4017B76B3F0F6ED6    3FF96F3FADB2DE80
 5      4015305B2F9260B6    400320DADAF641B6
 5      4015E3C4794BC789    400D6FAB320ADF56
 4      4010AA85D199550C    3FFDF00BAD7BE018
```

What we need to do is sort this file by KEY. Let us assume that real-world files, to which the task at hand is applicable, are not only 'longer' (i.e. contain large numbers of records) but, mainly, are also much 'wider' – in that the total length of the satellite variables makes it aspect ratio look 'wider' than 'longer'. Typically, whenever there is a need to sort such files, it yields well to methods based on *indirect sorting*. Simply put, it means that the file keys are first sorted with the associated observation numbers (file pointers, RIDs) as a satellite variable. Then the thus permuted RIDs are used to retrieve all records from the file, now in the sorted key order.

Essentially, this is what happens when an unsorted SAS data file is read with a BY statement using a SAS index created on the key. The index (stored on disk) contains the sorted keys and corresponding RID values, so to read a record, the index first locates the next key value in the ordered key sequence and then uses the corresponding RID value to grab its record from the file.

What we are going to do below is very similar, except that we shall create our own 'index' in the form of an ordered hash table - entirely in memory. And because we are going to store only RID values in its data portion along with KEY in the key portion, the table's memory footprint should be quite compact even for a relatively long file.

```
data SORTED ;
   dcl hash  H  (ordered: 'A', multidata: 'Y') ; /*declare ordered hash, allow for dup keys */
   dcl hiter I  ('H') ;                          /*need hash iterator to scroll thru table  */
   H.definekey  ('KEY') ;                        /*store KEY in the key portion of hash     */
   H.definedata ('KEY', 'RID') ;                 /*store both KEY and RID in data portion   */
   H.definedone () ;                             /*check if valid and instantiate           */

   if 0 then set UNSORTED ;                      /*need empty read to compile SAT1 and SAT2 */

   do RID = 1 by 1 until (eof) ;                 /*use RID to enumerate file records        */
      set UNSORTED (keep = KEY) end = eof ;      /*read record from UNSORTED, keep KEY only */
      h.add() ;                                  /*add next KEY and RID to hash             */
   end ;                                         /*at this point hash H is loaded           */

   do _iorc_ = I.first() by 0                    /*point iterator to lowest KEY in hash     */
                    while (_iorc_ = 0) ;         /*loop while next() still gets entries     */
      set UNSORTED point = RID ;                 /*use RID from hash to read disk record    */
      output ;                                   /*output record                            */
      _iorc_ = I.next() ;                        /*point iterator to next hash entry        */
   end ;                                         /*_iorc_ ne 0 means all entries listed     */

   stop ;                                        /*stop DATA setp execution                 */
run ;
```

As as result, file SORTED is in ascending order by KEY, and the locations of the SAT variables are correspondingly permuted. If printed, the output file would look thusly:

KEY	SAT1	SAT2
0	3FED7DC4B07AFB89	3FFF8228F75F0452
1	3FFDA780CAFB4F02	3FF1133022822660
1	3FFF4FF83F9E9FF0	400260A73AA4C14E
2	400581822A1B0304	3FF69AAE48AD355C
3	400CA92080D95241	3FF6083F2BAC107E
3	400DD3AF39BBA75E	40040D8626B81B0C
4	4010AA85D199550C	3FFDF00BAD7BE018
5	4017B76B3F0F6ED6	3FF96F3FADB2DE80
5	4015305B2F9260B6	400320DADAF641B6
5	4015E3C4794BC789	400D6FAB320ADF56
6	4019261E78724C3D	3FF79CD53F0F39AA
6	401B6144441EC288	40051381618A2703
7	401EC27CE9C584FA	3FFFA0D4313F41A8
8	40211039E5C22074	3FF0CBF4D22197EA
8	4020221502D4442A	40068DF6E50D1BEE
9	4023F0AF77D7E15F	3FF665AE2BECCB5C

If we needed to order the file into descending order, we would only need to change 'A' to 'D' in the DCL HASH statement.

But what if we did not want to physically write file SORTED to disk, yet still wanted to be able to access UNSORTED *as if there were an index on KEY* – so that it could be used as input with a BY statement? Why, it is easy! All we need to do is to rewrite the DATA statement as follows:

```
data VSORTED / view = VSORTED ;
```

Thus, instead of creating a physical data set, we are creating a view. Now we can read the view with the BY statement as desired. Here is the SAS log excerpt, the irrelevant log noise edited out:

```
NOTE: DATA STEP view saved on file WORK.VSORTED.
NOTE: A stored DATA STEP view cannot run under a different operating system.

84  data _null_ ;
85     set vsorted ;
86     by KEY ;
87  run ;

NOTE: View WORK.VSORTED.VIEW used (Total process time):
      real time           0.14 seconds
      user cpu time       0.00 seconds

NOTE: There were 16 observations read from the data set WORK.UNSORTED.
NOTE: There were 16 observations read from the data set WORK.VSORTED.
```

Since the step with BY statement does not stop and issues no errors, its input must be in order. Incidentally, using SET with BY in this fashion is the simplest way to check if an input file in physical sort order.

GETTING "TOP N" OBSERVATIONS FOR EACH VALUE OF A KEY

Suppose that, as it often happens in many industries, we have a "transaction" file comprising 3 variables: (1) transaction identifier KEY, (2) transaction amount QTY, (3) additional transaction (satellite) information SAT1 and SAT2. In the real world, satellites may be more numerous and contain, for example, such items as vendor business identifiers and/or descriptors. In large financial and telecommunication industries, transaction files may typically contain billions of records and appear in the form of unsorted input.

Now imagine that an analyst needs to extract N (for example, 5) records with the N highest values of QTY for each value of the transaction identifier KEY. To visualize the picture, let us, as usual, slap together some test data set:

```
data TRAN ;
   do KEY = 1 to 3 ;
      do _N_ = 1 to ceil (ranuni (1) * 8) ;
         QTY = ceil (ranuni (1) * 100) ;
         SAT1 = KEY * 1E6 + QTY ;
         SAT2 = ceil (ranuni (1) * 1E10) ;
         output ;
      end ;
   end ;
run ;

proc sort
   data = TRAN
   out  = TRAN_RND
   ;
   by SAT2 ;
run ;
```

Thus, the data set is intentionally randomly disordered to emulate the "real world" situation. However, if the file were sorted in the descending order by KEY, it would look as follows:

KEY	QTY	SAT1	SAT2
3	**85**	**3000085**	**6345241185**
3	**60**	**3000060**	**5825815265**
3	**51**	**3000051**	**9312135941**
3	**38**	**3000038**	**7283615600**
3	**29**	**3000029**	**4757893051**
2	**98**	**2000098**	**2265075186**
2	**96**	**2000096**	**2971939642**
2	**86**	**2000086**	**671845768**
2	**82**	**2000082**	**5238705215**
2	**69**	**2000069**	**4127638664**
2	55	2000055	5316917229
2	28	2000028	6899296310
2	5	2000005	665665517
1	**98**	**1000098**	**3998243061**
1	**26**	**1000026**	**9216025779**

Suppose that our fictional analyst wants 5 records with top 5 values of QTY for each value of the KEY. That would mean that from the file visualized above as sorted, the observations shown in **boldface** and red and green color would have to be obtained. Indeed, all the group KEY=3 has is 5 records, so all of them go to the output. The group KEY=2 contains more than 5 records, so only the ones with top 5 values of QTY are needed. Finally, the group KEY=1 contains fewer than 5 records, so again the entire group should go to the output.

Note that if, for some KEY value, there were more than one identical value of QTY, both should be given consideration because they may carry different additional info. If any of them tie for the Nth (in this case, 5th) place within a given KEY value, business rules would dictate which one is picked – but we shall not be concerned with such matters here.

An alert reader would have already raised a hand and asked what the problem is. Indeed, just sort the file by KEY as shown above, select N first records from each BY-group, and end the story, right? Sure – conceptually. Remember, though, that for a typical large transaction file that may consist of billions of records and long satellites, such as business names, the sorting is much easier said than done. Of course, no matter what algorithm is employed to tackle the problem, *the nature of the task requires reading the entire file at least once*. It is an arduous enough process for a large file; however, sorting requires about 4 times the disk space of the original file and is an order of magnitude more resource-intensive than the mere sequential read.

It is also important to keep in mind that real files may have hundreds and thousands identical KEY values (corresponding, for example, to transactions by the same vendor), yet we need only a small number of records per key. Therefore, writing out the entire file in the sorted order amounts to a substantial waste of computer space and time in addition to the resources expended by the sorting procedure per se.

Instead, proposed below is a SAS algorithm based on – you have guessed it! – a combination of an ordered hash object and direct-access SET statement. Here is 'annotated' pseudo-code:

1. Set the desired value of "top" N – for example, 5 - to a macro variable N.
2. Create a hash table H with entries ordered descending by [KEY, QTY, RID]. Including RID in the composite key ensures that the table can accommodate duplicate values of [KEY, QTY] without resorting to MULTIDATA:"Y". It will become clear later on why this extremely convenient feature cannot be used here instead of using RID as a discriminator.
3. Create a hash iterator I for the table H, so that the hash entries could be traversed serially – if necessary, starting with a certain [KEY, QTY, RID] tuple key value.

4. Create another hash table K, keyed only by KEY, in order to track how many entries for a given value of KEY table H currently contains - using the data portion variable LAST.N.
5. Insert a dummy entry with all missing values in table H via the ADD() method. (Although not mandatory, the entry serves as a "low-values" sentinel at the bottom of the table and helps greatly simplify the algorithm by avoiding unnecessary conditional statements.).
6. Read a record from TRAN_RND, keeping KEY and QTY only – we need no satellites at this point, for they would only create an unnecessary I/O drag.
7. Hit table K using the FIND() method with the current value of KEY (one just read from the file). If it is not found in the table, set LAST.N=0. Otherwise, FIND() will overwrite LAST.N in the PDV with the value it has just extracted from table K.
8. *Unconditionally,* insert an entry into table H. Because the table is ordered descending, the entry, for the current value of KEY, will automatically bubble up to its proper place by QTY rank.
9. If, for the current value of KEY, LAST.N = &N, the entry just inserted in table H (again, important: for the current KEY value!) is already (N+1)th. It has bubbled up to its proper rank among the (N+1) entries for this value of KEY. However, we need only &N entries for this value of KEY, hence the bottom-ranked entry for this value of KEY needs to be removed. To do so, use the SETCUR() iterator method to locate the entry we have just inserted in H. It will be now somewhere among the entries with the current KEY value. Then, beginning from this entry, use the NEXT() iterator method to step down table H one entry at a time until having bumped into an entry whose KEY value is *lower* than the KEY value we started the iterator with. It means that *the previous entry* the iterator hit is the bottom-ranked entry for the current KEY value, so use the REMOVE() method *to remove this previous entry from H*. (The process requires some simple variable reassignment mechanics to track which composite key values are "previous", because the NEXT() method overwrites their PDV values with those from the hash object.).
10. Otherwise, add 1 to LAST.N and insert it, for the current value of KEY, into table K. This way, we know - before any record is read sequentially from TRAN_RND - how many entries table H has got for any given KEY value encountered thus far.
11. If there are more records in TRAN_RND, go to step #6.
12. Otherwise, the file has been processed and table H contains, in its data portion, the values of RID pointing exactly to, *and only exactly to*, those records in TRAN_RND that we need. Therefore, use iterator I to step sequentially through table H. Every time the NEXT() method is called, another RID value from the table is written to its host PDV variable. Use this RID value with SET POINT=RID statement to extract SAT1 and SAT2 directly from file TRAN_RND. Now use the KEEP= option to read only the satellite variables, because they are the only ones we need from TRAN_RND at this point: KEY and QTY have been already written into their PDV host counterparts from table H by the iterator along with RID. For each iteration - except the one pointing to the dummy entry (for which RID is either missing or zero) - write the current PDV content to the output file. When the bottom sentinel entry is encountered (which will happen inevitably by design), just stop the loop.

And here is the SAS code:

```
%let N = 5 ;                                  /*set N to needed numnber of top N records  */

data TRAN_TOP_&N ;
   dcl hash  H  (ordered: 'D') ;              /*declare hash H as ordered descending      */
   dcl hiter I  ('H') ;                       /*declare iterator I for H                  */
   H.definekey  ('KEY', 'QTY', 'RID') ;       /*add RID to keys as discriminator          */
   H.definedata ('KEY', 'QTY', 'RID') ;       /*need to also add keys to data portion     */
   H.definedone () ;                          /*check and instantiate H                   */

   dcl hash K   () ;                          /*need hash K to track LAST.N for each KEY  */
   K.definekey  ('KEY') ;                     /*and so K is keyed by KEY only             */
```

```
   K.definedata ('LAST.N') ;                    /*LAST.N is # of entries for each KEY value */
   K.definedone () ;                            /*check and instantiate                     */

   H.add() ;                                    /*add dummy-sentinel all nulls entry into H */

   do _N_ = 1 by 1 until (eof) ;                /*process TRAN_RND using explicit DO-loop   */
      set TRAN_RND (keep = KEY QTY) end = eof ; /*read next in-record, keep keys only here  */

      if K.find() ne 0 then LAST.N = 0 ;        /*if KEY not in K, H has no entries for it  */
                                                /*else LAST.N for this KEY written to PDV   */
      RID = _N_ ;                               /*set RID to number of record just read     */

      H.add() ;                                 /*insert entry into H - no matter what      */

      if LAST.N = &N then do ;                  /*check if for this KEY, H has N entries?   */
         LAST.KEY = KEY ;                       /*remember current KEY value (from file)    */

         do _iorc_ = I.setcur() by 0            /*point iterator I to entry just inserted   */
                   while (KEY = LAST.KEY) ;     /*loop while next KEY same as previous      */
            LAST.QTY = QTY ;                    /*remember previous iterator entry QTY      */
            LAST.RID = RID ;                    /*remember previous iterator enrty RID      */
            I.next() ;                          /*dump KEY,QTY,RID from next entry into PDV */
         end ;
                                                /*now iterator is entry with next lower KEY */
         H.remove ( key: LAST.KEY               /*so need to remove previous iterator enrty */
                  ,key: LAST.QTY                /*i.e. last entry with KEY read from file   */
                  ,key: LAST.RID ) ;
      end ;
      else do ;                                 /*if # of H-entries < top N for current KEY */
         LAST.N ++ 1 ;                          /*One was just ADDed, so add 1 to LAST.N    */
         K.replace() ;                          /*and record this for this KEY in hash K    */
      end ;
   end ;                                        /*no more records? get out of the DO-loop   */

   do _iorc_ = I.first() by 0 while (RID) ;     /*if RID not 0/null get first entry from H  */
      set TRAN_RND (keep = SAT:) point = RID ;  /*get SAT1 and SAT2 straight from TRAN_RND  */
      output ;                                  /*write record to output file               */
      I.next() ;                                /*if RID not 0/null get next entry from H   */
   end ;

   stop ;                                       /*done - stop everything                    */
run ;
```

In the SAS listing form, data set TRAN_TOP_5 now looks just as expected:

KEY	QTY	SAT1	SAT2
3	85	3000085	6345241185
3	60	3000060	5825815265
3	51	3000051	9312135941
3	38	3000038	7283615600
3	29	3000029	4757893051
2	98	2000098	2265075186
2	96	2000096	2971939642
2	86	2000086	671845768
2	82	2000082	5238705215
2	69	2000069	4127638664
1	98	1000098	3998243061
1	26	1000026	9216025779

There are still a few notes of explanation we owe to the reader.

WHY NOT MULTIDATA: 'Y'?

First, we have promised to make it clear why the tag argument MULTIDATA: 'Y' is not used in this context to allow for duplicate entries by [KEY, QTY]. The reason is that not only it works, but it works too perfectly: Not only duplicate key entries are inserted into the table when ADD() is called, *all entries for the same key are also removed from the table when REMOVE() is called*. However, in this case, we want REMOVE() to kill only the (N+1)th, bottom-ranked entry for the current KEY value. If the Nth entry has the same rank (i.e. the same values of KEY and QTY) we want to leave it in the table, so, unfortunately, we could not use a multi-data hash object for the purpose. Yet if the specifications required to obtain, for each KEY value, only records *ranked differently*, the multi-data hash would work perfectly.

FIRST.X, LAST.Y AND NO BY - WHAT IS THAT ?!

Second, an attentive reader must have noticed that variables LAST.N, LAST.RID, LAST.KEY, LAST.QTY are used throughout the code, even though not a single BY statement is found anywhere in the step. One reason why we are using such variables in a peculiar manner for strictly utilitarian, auxiliary purposes is that in the context, these names are kind of apt. The other reason is... just because we can! A little experimental DATA step below may help clarify why it is possible without any adverse effects:

```
data TEST ;
   FIRST.A = 1 ;
   LAST.A  = 2 ;
   FIRST.B = 3 ;
   LAST.B  = 4 ;
   put _all_ ;
run ;
------------------------------------------------------------
FIRST.A=1 LAST.A=2 FIRST.B=3 LAST.B=4 _ERROR_=0 _N_=1
NOTE: The data set WORK.TEST has 1 observations and 0 variables.
```

The SAS log tells the story: Variables in the form of FIRST.X or LAST.X are NOT written to the output data set, so they can be used as utility variables without us worrying about dropping them. Unlike automatic variables _N_, _ERROR_, and _IORC_, these variables are not automatically retained. Further experimentation - no SAS documentation exists on the subject - shows that these variables are not written to output data sets unless they are *explicitly retained* – i.e. named in the RETAIN statement, or listed as elements of an initialized (valued) array, or come from a SAS data set. However, if they are retained, they *will* be written to the output named in the DATA statement, unless they are dropped by the DROP statement (not in the data set DROP option, which for these variables does not work). In this case, their names will appear in the output data set just as they are named, complete with the separating period. It seems to run counter to the SAS variable names conventions, which do not allow a period in variable names... but facts are facts. Unbelievers can try this:

```
data TEST ;
   retain FIRST.A 'FIRST.A' LAST.B 1 ;
run ;

proc contents data = TEST ;
run ;
```

and see if His Majesty Experiment dost speak the truth.

Variables of this species possess a number of other peculiar features, easily capable of filling another SAS paper, so we shall leave the digressing subject at this point. (Incidentally, such usage is an example of what David Cassell calls a 'dorfmanism', i.e. SAS stuff used in an unusual or unexpected fashion. Perhaps not surprisingly, different people exhibit

very different, most of the time, diametrical, attitudes towards 'dorfmanisms' – some get infatuated with them, while others vehemently, indeed sometimes quite pugnaciously, oppose.)

WHAT IF THE INPUT FILE IS NOT A SAS DATA SET?

It is, of course, a very good question. What if the input file is a non-SAS data base table, stored, for example, in DB2, Oracle, or Teradata? If this is the case, the SET statement with POINT= options cannot be used, so does it invalidate the approach described above? In the physical/literal sense – i.e. that the SET statement with POINT= option cannot be used – sure! However, it does not invalidate the concept itself at all. Indeed, if TRAN_RND is, for instance, a Teradata table, it is invariably indexed, so instead of RID we can store the index columns in the hash object, and then use the SET statement with KEY= option to retrieve the rows pre-identified in the hash using the data base index. With data bases, where the concept of row ID is implemented (Oracle, for example), it is even simpler and closer to the SAS implementation.

CONCLUSION

Combining the SAS hash object with the time-tested direct access to SAS data sets via the POINT= option lends itself to a number of programming techniques, which can preserve performance advantages offered by the hash object by offloading a major part of its RAM footprint to the original disk-resident file.

ACKNOWLEDGEMENTS

The authors owe their gratitude to Edward Heaton for his kind invitation to present this material at NESUG 2011. Also, they would like to thank Karsten M. Self - the person, in whom one of the authors (P.D.) first confided the concept comprising the subject of this paper during a vigorous intellectual correspondence on hashing in SAS the day before Christmas 1998. His enthusiastic support aptly expressed in the email subject line 'Hash Rocks, Dude!' impelled his interlocutor to redouble his SAS hashing efforts. Finally, it is difficult to imagine how the SAS hashing concepts, which started as an obscure SAS-L post filled with hairy SAS code back in December of 1998, could make its way into the SAS mainstream without their recognition by Ian Whitlock and ardent support by his authority as the acknowledged SAS Master of the Universe.

REFERENCES

[1] D.E. Knuth, The Art of Computer Programming, 2.
[2] D.E. Knuth, The Art of Computer Programming, 3.
[3] R. Sedgewick, Algorithms in C, 1-4.
[4] T.A. Standish. Data Structures, Algorithms and Software Principles in C.
[5] P.M. Dorfman. Table Lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing. Proceedings of SUGI 26, Long Beach, CA, 2001.
[6] M.A. Raithel. Tuning SAS Applications in the MVS Environment, Cary,NC:SAS Institute Inc.,1995.
[7] P.M. Dorfman, K.Vyverman. Data Step Hash Objects as Programming Tools. Proceedings of SUGI 30, Philadelphia, PA, 2004.
[8] P.M. Dorfman. From Sequential Search to Key-Indexing. Proceedings of SESUG'99, Mobile, AL, 1999.
[9] M.A. Raithel. The Complete Guide to Creating and Using Indexes, Cary,NC:SAS Institute Inc.,2005.
[10] P.M. Dorfman, L.S. Shajenko. Crafting Your Own Index: Why, When, How. Proceedings of SUGI 31, San Francisco, CA, 2005.
[11] R.Ray and J.Secosky. Better Hashing in SAS 9.2. http://support.sas.com/rnd/base/datastep/dot/better-hashing-sas92.pdf

AUTHOR CONTACT INFORMATION
Paul M. Dorfman
4437 Summer Walk Court
Jacksonville, FL 32258
(904) 260-6509
paul_dorfman@bellsouth.net

Lessia S. Shajenko
12 Mary Shepherd Rd
Littleton, MA 01460-2261
(617) 489-8037
Lessia.S.Shajenko@bankofamerica.com