Paper 94

Using the SAS® Hash Object with Duplicate Key Entries

Paul M. Dorfman, Independent Consultant, Jacksonville, FL

ABSTRACT

By default, the SAS hash object permits only entries whose keys, defined in its key portion, are unique. While in certain programming applications this is a rather utile feature, there also others, where being able to insert and manipulate entries with duplicate keys is imperative. Such an ability, facilitated in SAS since Version 9.2, was a welcome development: It vastly expanded the functionality of the hash object and eliminated the necessity to work around the distinct-key limitation using custom code. However, nothing comes without a price; and the ability of the hash object to store duplicate key entries is no exception. In particular, additional hash object methods had to be - and were - developed to handle specific entries sharing the same key. The extra price is that using these methods are surely not quite as straightforward as the simple corresponding operations on distinct-key tables, and the documentation alone is a rather poor help for making them work in practice. Rather extensive experimentation and investigative coding is necessary to make that happen. This paper is a result of such endeavor, and hopefully, it will save those who delve into it a good deal of time and frustration.

INTRODUCTION

For SAS Version 9.2 and higher, the SAS hash object was given a new, heretofore absent, ability to store and manipulate entries with duplicate keys. Before that, any time the programming task demanded the presence of non-unique key entries in a hash table, it was necessary to resort to clever workarounds, for example, by adding another, unique component to the key portion and using another hash table to track the entries - otherwise, an attempt to add an entry with a key already in the table would result in an abend and an error message in the SAS log. Now that SAS had added the MUTLIDATA: "Y" argument-tag to the object constructor, such workarounds became superfluous. However, with the new functionality came a need to answer a few new questions before a table with non-distinct key entries could be successfully exploited in a program, for example:

- How to harvest all entries with the same key from the hash table?
- How to remove a specific entry within the same key-group based on a condition or, if need be, all entries with the same key?
- How to update the data portion information for a specific entry within the same key-group or, for that matter, all entries with the same key?

To answer the questions from this by no means exhaustive list, SAS had offered new hash object methods and corresponding descriptions in the documentation augmented by usage examples. As I have subsequently discovered the hard way, neither is always necessary and sufficient to gain proper understanding of the rules by which the operations facilitated by the methods are governed. So, I betook myself to programmatic experiments aimed at refining those rules for my own benefit - and, hopefully, for that of others, too. This paper is a compendium of what I've been able to distill the only way I know how to learn about the things of this nature.

DUPLICATE-KEY ENTRY HASH TABLE LOAD

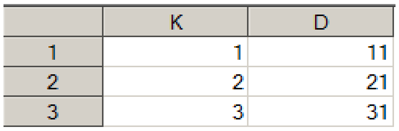
Fortunately, there is nothing fancy about loading a hash table with duplicate key entries. However, let us first see what happens if we attempt to load duplicate key entries into the hash object *without telling it specifically to accept them*:

```
data multi ;
   input K D ;
   cards ;
1 11
2 21
2 22
3 31
3 32
3 33
;
run ;

data _null_ ;
   if 0 then set multi (keep = K D) ;
   dcl hash h   (dataset: "multi", ordered: "A") ;
   h.definekey  ("K") ;
   h.definedata ("D") ;
   h.definedone () ;
   h.output (dataset: "H") ;
   stop  ;
run ;
```

The SAS log tells us:

```
NOTE: There were 6 observations read from the data set WORK.MULTI.
NOTE: The data set WORK.H has 3 observations and 2 variables.
```

and, looking at the output data set H reflecting the hash table content, we see the following picture:



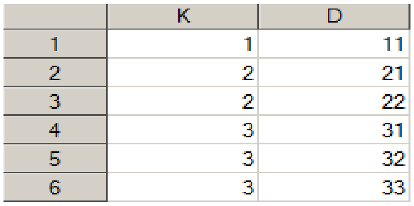
	K	D
1	1	11
2	2	21
3	3	31

Obviously, without telling SAS *specifically to accept multiple key entries*, only the first entry for each key is stored, so we need to address that. This is done by using the MULTIDATA: "Y" hash parameter tag in the hash object declaration:

```
data _null_ ;
   if 0 then set multi (keep = K D) ;
   dcl hash h   (dataset: "multi", ordered: "A", multidata: "Y") ;
   h.definekey  ("K") ;
   h.definedata ("D") ;
```

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

Now if we rerun the last step, the SAS log tells us that data set H now contains 6 records, and the hash table content looks as expected:



	K	D
1	1	11
2	2	21
3	2	22
4	3	31
5	3	32
6	3	33

Needless to say, using the tag parameter DATASET: to load the entries is not the only way. They can be also loaded explicitly one entry at a time by calling the ADD method repeatedly or in a DO loop. For example:

```
data _null_ ;
   dcl hash h    (ordered: "A", multidata: "Y") ;
   h.definekey  ("K") ;
   h.definedata ("K", "D") ;
   h.definedone () ;
   do until (end) ;
      set multi end = end ;
      h.add() ;
   end ;
   h.output (dataset: "H") ;
run ;
```

Jumping a bit ahead, let us note that instead of the ADD method, the REPLACE method can be also used to get the same result. This is quite logical, as with MULTIDATA: "Y", the data portion item for the same key is not overwritten. Instead, the entry with the same key is merely added to the table. Now that we know how to load same-key entries into the hash object, we can proceed to learning how to traverse them or, to use another term, harvest them.

TRAVERSING (HARVESTING) SAME-KEY ENTRIES

Suppose we have key K with a specific key value and want to do the following:

1. If the value of K is not in table H, do nothing.
2. Otherwise, if it is in table H, print all corresponding hash entries in the SAS log.

Intuition tells to try FIND method repeatedly in hope that each time it is called, it will get the next entry while the return code is zero. However, it is not how it works in reality. Let's try it, say for K=3:

```
data _null_ ;
   dcl hash h    (dataset: "multi", ordered: "A", multidata: "Y") ;
   h.definekey  ("K") ;
   h.definedata ("K", "D") ;
   h.definedone () ;
   k = 3 ;
   do i = 1 to 3 while (h.find() = 0) ;
      put k= d= ;
   end ;
   stop  ;
   set multi ;
run ;
--------
k=3 d=31
k=3 d=31
k=3 d=31
```

Above, the SAS log content caused by the PUT statement is shown below the dash line. Evidently, every time FIND method is called, it returns only the first entry in the K=3 group.
Obviously, the FIND method alone is not enough, and something else is needed. That something else is the additional method named FIND_NEXT, which SAS have designed specifically for this purpose to work in tandem with FIND using the following scheme:

1. Call the FIND method.
2. If the key is not in the table (the call returns a non-zero code), there is nothing to harvest, so exit the algorithm.
3. Otherwise, calling FIND will set program control at the first logical hash table entry (from now on we will interchangeably say "hash pointer" or just "pointer" by analogy with a file record pointer and use the respective verbs). Also, the FIND call will overwrite the PDV host variables values with the values from the respective variables from the data portion of the hash table. So, print the variable values in the log and output the record.
4. Call the FIND_NEXT method.
5. If the FIND_NEXT call return code is zero, it means that there exists another entry with the same key value, and program control is now pointing at this entry. Go to step #4.
6. Otherwise if the return code is not zero, there are no more entries with the same key, and the pointer has moved past the last entry with this key value. Stop.

In the SAS language, it can be expressed much more tersely merely by replacing the DO-loop in the example above with:

```
   rc = h.find() ;
   do while (rc = 0) ;
      put k= d= ;
      rc = h.find_next() ;
   end ;
--------
k=3 d=31
k=3 d=32
k=3 d=33
```

Another, even more concise entry-harvesting form, can be devised by using the FROM-BY index properties of the DO-loop:

```
   do _iorc_ = h.find() by 0 while (_iorc_ = 0) ;
      put k= d= ;
      _iorc_ = h.find_next() ;
   end ;
```

Above, using H.FIND() as the FROM expression initializes the loop index _IORC_ with its return code. If the latter is not zero, no key value the FIND method accepts is found in the table, and so the loop (because of the WHILE clause) will not iterate even once. Hence, nothing germane will appear in the log. Otherwise, if H.FIND() returns zero, the loop will enter its first iteration, print the values of K and D, and call H.FIND_NEXT(). If this call returns _IORC_=0, the loop will enter its next iteration. However, if the BY expression were not given 0, it would stop right there. The BY expression keeps the loop in the iterative mode, but because it is valued as 0 it cannot modify _IORC_ in any other way but via the H.FIND_NEXT() call - which is exactly what is needed. Note that in this code excerpt, automatic numeric variable _IORC_ is used as a return code container instead of RC - solely because, unlike RC, it is auto-dropped, and so there is one fewer little thing to worry about.

ONE-TO-MANY AND MANY-TO-MANY HASH JOINS

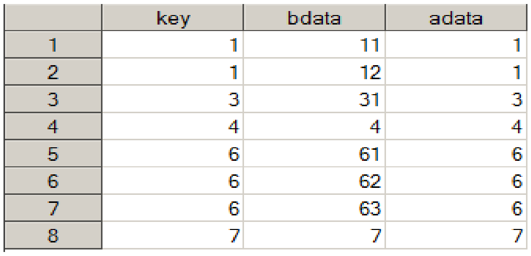
Before the argument tag MULTIDATA:"Y" functionality was developed, only unique key entries could be stored in a hash table. Thus, matching two files with one-to-many or many-to-many key relationships using the hash object had been problematic and involved custom code and an additional hash table to disambiguate between the same-key entries. Now with MUTIDATA:"Y" and the ability of the FIND_NEXT method to harvest duplicate-key entries it had become simple. Let's consider an example. Below, the hash object is used to perform equivalents of one-to-many and many-to-many joins.

1. One-to-Many, Many-to-Many Equijoins

```
data a ;
input key adata ;
cards ;
1  1
2  2
3  3
4  4
5  5
6  6
7  7
;
run ;

data b ;
input key bdata ;
cards ;
1  11
1  12
3  31
4  4
6  61
6  62
6  63
7  7
;
run ;

data c ;
   if _n_ = 1 then do ;
```

```
      if 0 then set b ;
      dcl hash   b (dataset: "b", multidata: "y") ;
      b.definekey  ("key") ;
      b.definedata ("bdata") ;
      b.definedone () ;
   end ;

   set a ;

   do _iorc_ = b.find() by 0 while (_iorc_ = 0) ;
      output ;
      _iorc_ = b.find_next() ;
   end ;
run ;
```

Result (File C content):



	key	bdata	adata
1	1	11	1
2	1	12	1
3	3	31	3
4	4	4	4
5	6	61	6
6	6	62	6
7	6	63	6
8	7	7	7

- First, file B is loaded into hash table B using the argument tag DATASET with all the same-key entries thanks to the MULTIDATA:"Y" specification.
- Parameter type matching (placing the host variables for hash B into the PDV) is done by reading the descriptor of data set B on line 3.
- Then, for each record read from file A, all entries, whose KEY match the KEY value in the record being currently read, are harvested, and each of them contributes a record to the output file C.
- Note that it's not critical that file A in the sample data is unique by KEY. This code would still work correctly if the keys in file A were also duplicate.
- Because the step above outputs a record only if there is a match, the non-matching KEY=2 and KEY=5 from file A are absent from the output - exactly what is expected from an *equijoin*.

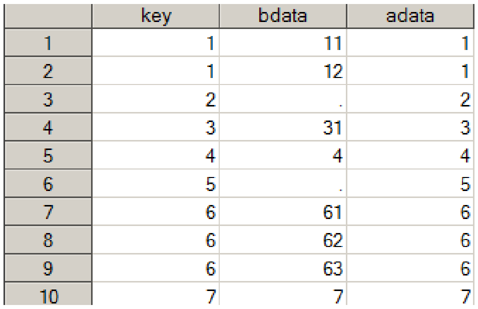
2. One-to-Many, Many-to-Many Left Joins

If we want a left join instead, (a) every KEY from file A must be present in the output, however (b) the output records whose input KEY has no match in file B must have the BDATA values missing. To achieve that, the step above should be tweaked a bit to meet these two requirements:

```
data c ;
   if _n_ = 1 then do ;
      if 0 then set b ;
      dcl hash   b (dataset: "b", multidata: "y") ;
      b.definekey  ("key") ;
      b.definedata ("bdata") ;
```

```
    b.definedone () ;
   end ;

   set a ;

   _iorc_ = b.find() ;

   if _iorc_ ne 0 then call missing (bdata) ;

   output ;

   do while (b.find_next() = 0) ;
      output ;
   end ;
run ;
```

Result (File C content):



	key	bdata	adata
1	1	11	1
2	1	12	1
3	2	.	2
4	3	31	3
5	4	4	4
6	5	.	5
7	6	61	6
8	6	62	6
9	6	63	6
10	7	7	7

- Now, we first call B.FIND(). If there is not match in B, the satellite variable from B, BDATA, is set to the standard missing value, and the PDV content is written out. In this case, when program control enters the DO loop, the condition B.FIND_NEXT()=0 is untrue, hence (because of WHILE) program control exits the loop without hitting the OUTPUT statement and moves to the SET statement instead to read the next record from A (or hits the empty buffer and terminates the step if there is nothing more to read).
- Otherwise, if there is a match by KEY, calling B.FIND() places the hash pointer at the first entry with this KEY, overwrites host variable BDATA from its value from this hash entry, and outputs the record.
- If the KEY is not duplicate, the call to B.FIND_NEXT() returns a non-zero code, and the DO loop terminates exactly as above.
- If the KEY is duplicate in B, the DO loop starts iterating and executes the OUTPUT statement for all remaining entries with the same KEY.

Note that is the line with CALL MISSING were omitted, it would result in incorrect output because BDATA is auto-retained because of the SET B statement on line 3. (If parameter type matching were done using the LENGTH statement, it would not be the case, and CALL MISSING would not be necessary.)

3. One-to-Many, Many-to-Many Full Joins

What if we need to perform a full hash join? In other words, let's alter data set B a little, so that it will have some records in file B with no match in file A by KEY. Below, it is done by inserting two fractional keys 1.5 and 6.5 in the file:

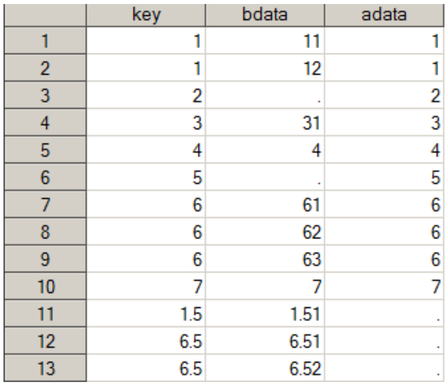
```
data b ;
input key bdata ;

cards ;
1   11
1   12
1.5 1.51
3   31
4   4
6   61
6   62
6   63
6.5 6.51
6.5 6.52
7  7
;
run ;
```

In output file C, we now need: (a) all records from A with BDATA missing where there is no match in B and (b) all records from B with ADATA missing where there is no match in A. To achieve that, we can devise the following scheme:

1. Do the same we have done for the left join and output all requisite left-join records.
2. While passing through file A, remember all *unique* values of KEY we encounter in a separate hash table X. Hence, MUTIDATA:"Y" is *not* needed for this extra table.
3. After the left join is complete, go through hash B (already containing the data from file B) one entry at a time. To do so, we will need to declare a hash iterator for table B. Below, it is named IB.
4. Call X.CHECK(). If KEY in the entry, at which the iterator IB is pointing in table B, has a match in table X, we are not interested: Those records are already in the output. Otherwise, output the record to augment the left join output with KEY values from file B not found in file A and the corresponding BDATA values.

```
data c ;
   if 0 then set B ;
   dcl hash  B  (dataset: "B", multidata: "Y") ;
   dcl hiter IB ("B") ;
   b.definekey  ("key") ;
   b.definedata ("key", "bdata") ;
   b.definedone () ;

   dcl hash X   () ;
   x.definekey  ("key") ;
   x.definedone () ;

   do until (eof) ;
      set A end = eof ;
      x.replace() ;
      if b.find() ne 0 then call missing (bdata) ;
      output ;
      do while (b.find_next() = 0) ;
         output ;
      end ;
   end ;
   call missing (adata) ;
   do while (ib.next() = 0) ;
      if x.check() ne 0 then output ;
   end ;
   stop ;
run ;
```

Result (file C content):



	key	bdata	adata
1	1	11	1
2	1	12	1
3	2	.	2
4	3	31	3
5	4	4	4
6	5	.	5
7	6	61	6
8	6	62	6
9	6	63	6
10	7	7	7
11	1.5	1.51	.
12	6.5	6.51	.
13	6.5	6.52	.

Full-join Notes:

- If we should happen to have more satellite variables (like ADATA and BDATA) on one or both sides, listing them in CALL MISSING can become fuzzy. It is much simpler to replace it with CALL MISSING (OF _ALL_) to automatically cover the whole territory. The reason it was not done above is purely pedagogical: it demonstrates which particular side variable we intend to "nullify".
- The hash iterator does not care whether the table is created with MULTIDATA:"Y" or not; it works exactly the same way in both cases.

THE CURIOSITY OF THE *FIND_PREV* METHOD (AND MORE)

In addition to the FIND_NEXT method, there is also a method named FIND_PREV. It's obvious from its name that, within the group of entries with the same key, the method is designed to move the hash pointer one entry up from the entry being pointed at currently. At this time, I consider the method more curious than functional, and here is why.
One would think that the method would be useful to traverse/harvest same-key hash entries backwards versus the FIND_NEXT method. There indeed would be a certain degree of utile functionality in it - for example, if the hash table were ordered ascending, we might envision a situation where it would be beneficial to be able to traverse/harvest its entries in reverse, i.e. descending. But in this case, we would also need a method that would place the hash pointer on the *last* same-key entry, just like the FIND method places it on the first. However, no such method exists - yet. Therefore, in order to harvest the same-key entry group backwards, we first need to set the hash pointer at the last entry, and the only way to do so it is to travel there using FIND_NEXT. This is the reason I have not found any significant practical utility for FIND_PREV yet. However, since my readers have repeatedly proven to be smarter than me, I thought I would still give an example of its usage using the scenario described above. The sample data set used below is snatched from the SAS documentation, except that here, it is ordered intrinsically for easier visual consumption:

```
data h ;
input key hdata ;
cards ;
1   5
1  10
1  15
```

```
2   9
2  11
2  16
3  20
3 100

4   6
5   5
5  99
;
run ;
```

Now suppose that we have loaded this data set into a hash table H and want to list the hash entry group keyed by KEY=2 backwards. To do that, we need to (a) call H.FIND() to point at the first entry, (b) use H.FIND_NEXT() to get to the end of the group, (c) *do something* to make the hash pointer to *actually stop at the last entry* since merely repeating H.FIND_NEXT() calls would get the pointer past the end, and (d) traverse backwards by repeatedly calling H.FIND_PREV(). Let us write code first and discuss the "do something" part in (c) *a posteriori*.

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;

   key = 2 ;
   do rc = h.find() by 0 while (rc = 0) ;
      h.has_next (result: have_more) ;
      if have_more = 0 then leave ;
      rc = h.find_next() ;
   end ;

   put key= hdata= ;
   do while (h.find_prev() = 0) ;
      put key= hdata= ;
   end ;
run ;
--------------
key=2 hdata=16
key=2 hdata=11
key=2 hdata=9
```

Let us make a few extra notes:

- An alert reader has doubtless noticed that in addition to FIND and FIND_NEXT, there is a call to yet another method, *HAS_NEXT*. This is the *"something"* mentioned in the item (c) above. The method gives us the ability to predict whether the entry, on which the hash pointer presently dwells, is the last in the same-key block or there is another entry with the same key ahead of it. You supply a variable of your choice - above, it is variable *have_more* - and H.HAS_NEXT(RESULT: HAVE_MORE) call sets HAVE_MORE=0 if this is the last entry and to a non-zero value otherwise.

- Because of the conditional LEAVE statement, the loop does not call H.FIND_NEXT() another time when the pointer has moved to the last same-key entry. In its absence, H.FIND_NEXT() would kick the hash pointer past the last KEY=2 entry group, and the ensuing H_FIND_PREV() call would have no effect by stopping the second DO loop without executing its body even once.
- While at that, let us mention that there exist yet another hash method, *HASH_PREV*. It is completely analogous to HAS_NEXT and works exactly the same way, except it helps determine whether or not the hash pointer dwells at the *first* same-key entry by looking backwards.
- Logically, code offered above works correctly regardless of whether a key group has only one entry or more. If you would like to verify it, try it with KEY=4 (which is unique) instead.
- Neither HAS_NEXT, nor HAS_PREV modifies any hash host variables in the PDV. In this sense, they are just as "neutral" as the CHECK method.

REMOVING ENTRIES FROM SAME-KEY ENTRY BLOCKS

If there is a need to remove entries from a hash block with duplicate key values, the task fundamentally falls into two unequal categories: (a) remove the entire block and (b) remove only certain entries selectively based on specific conditions. The inequality comes from the fact that (a) is a piece of cake, while (b) requires the kind of understanding of hash pointer mechanics we have already degusted above.

1. Kill'em All (a Piece of Cake)

Let us continue using the same sample data set H we have created earlier. Now suppose that we need to remove the entire block of hash entries with KEY in (1, 3, 5). It turns out that for this purpose, the old good "pre-multidata" method REMOVE works just fine. You give it a key, and if it is not in the table, nothing happens, except that the method returns a non-zero code. Otherwise if the key is in the table, the method removes the hash entries with this key all at once:

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;

   key = 1 ;
   rc = h.remove() ;
   rc = h.remove (key: 3) ;
   k = 5 ;
   rc = h.remove(key: k) ;

 * Use hash iterator IH to show content of table H ;
   dcl hiter ih ("H") ;
   do while (ih.next() = 0) ;
      put key= hdata= ;
   end ;
run ;
--------------
key=2 hdata=9
key=2 hdata=11
```

```
key=2 hdata=16
key=4 hdata=6
```

In this step, just for the purpose of demonstration, the three different keys were used with H.REMOVE() call in slightly different fashions. First, the method can merely accept the key value currently in the PDV (shown with KEY=1). Second, it can accept *an expression* given to the parameter tag KEY: if the expression is of the correct data type. The onus of making sure the type is correct and that the expression resolves in the requisite key value is on the programmer. The expressions used in this example are simple: A literal (shown with KEY=3) and a variable (KEY=5). However, in reality it can be any valid expression (including, if need be, functions, operators, etc.) as long as it is of the correct data type and resolves in the value you need.

2. Removing Same-Key Hash Entries Selectively

Ah, now that is different. If we want to remove an entry *based on a condition* we need to find this entry - if it exists, of course, and then remove it. The REMOVE method is ill-suited for the purpose since, as we already know, it removes all entries with a given key at once. So, an additional method, aptly named *REMOVEDUP*, has been designed for selective/conditional removal.

1. First, let us consider a simple case when all we want is to remove the entries with KEY in (1, 2, 5) where HDATA>9. In order to do that, we need, for each key in question, do the following.
2. Place the hash pointer on the first key entry.
3. Traverse the same-key block from the beginning to the end.
4. While at that, remove each entry where HDATA>9.

Translating it into the SAS language:

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y", ordered: "A") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;

   do key = 1, 2, 5 ;
      do find_rc = h.find() by 0 while (find_rc = 0) ;
         if hdata > 9 then remove_rc = h.removedup() ;
         find_rc = h.find_next() ;
      end ;
   end ;

   dcl hiter ih ("H") ;
   do while (ih.next() = 0) ;
      put key= hdata= ;
   end ;
run ;
```

```
---------------
key=1 hdata=5
key=1 hdata=15
key=2 hdata=9
key=2 hdata=16
key=3 hdata=20
key=3 hdata=100
key=4 hdata=6
key=5 hdata=5
```

Now let us have more fun and try doing this:

- If a key has only one entry (unique key), leave the entry alone
- Else if a key has more than one entry, remove the last one

Adding pun to fun, having this kind of fun is also funny because this task, used as an example for the REMOVEDUP method in the SAS documentation, is actually coded incorrectly - and results in incorrect output. Doubters can run code provided in this link:

http://support.sas.com/documentation/cdl/en/lecompobjref/63327/HTML/default/viewer.htm#n0ok7sej0uwin8n13ti53el0inta.htm

and compare output with the program's intended goal. Thus, let us rectify it now:

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y", ordered: "A") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;

   do key = 1 to 5 ;
      find_rc = h.find() ;
      do while (h.find_next() = 0) ;
         h.has_next(result: have_more) ;
         if not have_more then h.removedup() ;
      end ;
   end ;

   dcl hiter i ("h") ;
   do while (i.next() = 0) ;
      put (key hdata) (=) ;
   end ;
run ;
--------------
key=1 hdata=5
key=1 hdata=10
key=2 hdata=9
key=2 hdata=11
key=3 hdata=20
key=4 hdata=6
key=5 hdata=5
```

Notes:

- The call to H.FIND() is taken out of the DO loop specifications intentionally. Doing so ensures that if the key is unique, the loop will not iterate even once because H.FIND_NEXT() will return a non-zero code, leaving the entry intact.
- If the key is not unique, the loop will traverse its same-key entry block past it end, removing the last entry when the IF NOT HAVE_MORE condition evaluates true.

REPLACING THE DATA PORTION WITHIN SAME-KEY ENTIRES

In the SAS hash object's bailiwick, "replace" usually means overwriting a data portion variable in a hash table for a specific key entry with a different value. The REPLACE works perfectly well for tables with unique key entries. The method accepts a key value, and if it is found in the table, the corresponding data portion variable(s) in the entry is/are replaced with either the values currently residing in the PDV host variables or the values to which expressions supplied to the argument tag DATA resolve. However, with MULTIDATA:"Y", we need to decide which same-key entry's data portion variable(s) to replace based on certain conditions. In principle, the mechanics needed to identify an entry or entries in question is basically the same as with the REMOVEDUP method, so there is no need to reinvent the wheel.

1. Using the REPLACEDUP method

Suppose, for example, that, emulating the first REMOVEDUP example, we want to replace HDATA in the entries keyed by keyed by KEY in (1, 2, 5) where HDATA>9 with HDATA=HDATA*1001 (there is a method behind the madness of choosing 1001 - it makes the replacement values more starkly glaring). Just as with REMOVEDUP, we:

1. Place the hash pointer at the first entry in the key block using the FIND method.
2. Traverse the key block using the FIND_NEXT method.
3. If in any given entry, HDATA>9, multiply the value just dumped into PDV host variable HDATA by the method call.
4. Call REPLACEDUP to overwrite the HDATA value in this hash entry with the HDATA value times 1001.

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y", ordered: "A") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;

   do key = 1, 2, 5 ;
      do find_rc = h.find() by 0 while (find_rc = 0) ;
         if hdata > 9 then do ;
            hdata = hdata * 1001 ;
            replace_rc = h.replacedup() ;
         end ;
         find_rc = h.find_next() ;
      end ;
   end ;

   dcl hiter ih ("H") ;
   do while (ih.next() = 0) ;
      put key= hdata= ;
   end ;
run ;
-----------------
key=1 hdata=5
key=1 hdata=10010
key=1 hdata=15015
key=2 hdata=9
key=2 hdata=11011
key=2 hdata=16016
key=3 hdata=20
key=3 hdata=100
key=4 hdata=6
key=5 hdata=5
key=5 hdata=99099
```

Now, let us do roughly the same but, just as in the second REMOVEDUP example, replace *the data in the last entry* only within those blocks that *have duplicate-key entries*:

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y", ordered: "A") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;
   do key = 1 to 5 ;
      find_rc = h.find() ;
      do while (h.find_next() = 0) ;
         h.has_next(result: have_more) ;
         if not have_more then h.replacedup(data: key, data: hdata*1001) ;
      end ;
   end ;

   dcl hiter i ("h") ;
   do while (i.next() = 0) ;
      put (key hdata) (=) ;
   end ;
run ;
key=1 hdata=5
key=1 hdata=10
key=1 hdata=15015
key=2 hdata=9
key=2 hdata=11
key=2 hdata=16016
key=3 hdata=20
key=3 hdata=100100
key=4 hdata=6
key=5 hdata=5
key=5 hdata=99099
```

Notes:

- H.REPLACEDUP(DATA: KEY, DATA: HDATA*1001) call merits elaboration.
- It could be done with assigning HDATA=HDATA*1001 and then calling H.REPLACEDUP() without argument tags.
- It was done as it was done with the purpose of calling attention to the variegation the method is capable of.
- The only argument tag that can be used with the REPLACEDUP method is DATA; the key portion is assumed. The reason KEY had to be specified in addition to HDATA is because the table was defined with KEY in its data portion. The latter was only necessary to make KEY PDV-dumpable - and hence printable in the log - by the I.NEXT() call.

2. What about the REPLACE method?

Knowing how the REMOVE method works on same-key hash entries - that is, by removing the whole entry block with the matching key - it would be reasonable to speculate that REPLACE would work in a similar way and replace a data portion variable with the current value of its PDV host variable in all key-matching entries. However, *it is NOT* how it works. Instead:

- If the key is found in the table, REPLACE merely adds another entry to the corresponding key block.
- Otherwise, it just adds another entry to the table - both with the new key and data.

Let us check it out:

```
data _null_ ;
   if 0 then set H ;
   dcl hash  H  (dataset: "H", multidata: "Y", ordered: "A") ;
   h.definekey  ("key") ;
   h.definedata ("key", "hdata") ;
   h.definedone () ;

   do key = 1, 3, 7 ;
      hdata = key * 1001 ;
      rc = h.replace() ;
   end ;

   dcl hiter i ("h") ;
   do while (i.next() = 0) ;
      put (key hdata) (=) ;
   end ;
   stop ;
run ;
----------------
key=1 hdata=5
key=1 hdata=10
key=1 hdata=15
key=1 hdata=1001
key=2 hdata=9
key=2 hdata=11
key=2 hdata=16
key=3 hdata=20
key=3 hdata=100
key=3 hdata=3003
key=4 hdata=6
key=5 hdata=5
key=5 hdata=99
key=7 hdata=7007
```

Note that if you examine the RC values in the DO KEY loop, it turns out that RC=0, regardless of whether there is a key match or not. In other words, the method is always successful, and in this situation it is completely identical to the ADD method.

CONCLUSION

The ability to store more than one entry per key in a hash table adds a lot of functionality to the SAS hash object. Compared to the tables with unique key entries only, it makes hash equivalents of things like many-to-many joins much easier to program. At the same time, using table with duplicate keys requires learning new SAS-supplied methods necessary to traverse same-key entry blocks, harvest and remove entries from them, and replace data portion values. Although the narrative and examples in this paper pretty much only scratch the surface of what can be done using the ability of the hash object to operate with duplicate key entries, it should provide some initial insight into the way they work.

CONTACT INFORMATION

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