Why Hash?

Glen Becker, USAA

Abstract: What can I do with the new Hash object in SAS 9? Instead of focusing on "How to use this new technology", this paper answers "Why would I want to?" It presents the "Big Picture" of the hash object, to help you plan whether an application can benefit from using the hash object, and how to plan to use it. It also introduces the %Hash macro that makes it easy for you to define hash objects.

Body:

The new SAS hash object makes it possible for SAS to do things previously not possible:

- Merge Without Sorting:
  - Try one key, then another.
  - Merge with 2+ files with different keys
- Fast, flexible table-lookup.
- Summarize in a DATA step on-the-fly.
- Store data like an array, but:
  - Can have character keys and composite keys
  - No need to predict size in advance ← A BIG advantage!
  - Can return multiple variables at once
  - Can automatically sort data on-the-fly
  - Can easily read a SAS dataset into a hash object or write a SAS dataset from one.

How does it do this? You define be one or more variables, which can be numeric or character, as the key, and other variables as data variables. The hash object converts your key value(s) into a number that represents a hash "bucket", and stores the data there. A bucket is actually a collection of key-data pairs stored in memory.

You can add, replace, remove or retrieve data from a hash bucket using the key. Because SAS can directly locate your data without having to search for it, storing and retrieving data with a hash is far faster than using other techniques.

You can also "run" the hash from beginning to end to process all items in the hash. You can request that the hash return the items in key-variable order. Lastly, you can easily copy data between SAS datasets and hash objects.

Sounds great! Are there any drawbacks? Yes, there are some limitations:

1. The hash object can only be created and used in a DATA step program.
   PROC SQL cannot use it, except SAS 9.2 is pretty good at "hash joins".
   A macro cannot directly create and use a hash object, but creative use of macro variables provides some of the same benefit.
2. The entire hash object must fit in memory. This limits its maximum size because:

   (a) memory has limits (SAS, OS, or physical memory). Specifically, 32-bit SAS on windows limits you to about 1GB, but Linux gives you 3GB. 64-bit SAS lifts these restrictions.

   (b) copying data between a SAS dataset and a hash object takes time. You would not want to read a gigabyte of data into a hash object to do a table-lookup if your DATA step only processes a few hundred observations. An Index would be a better way of handling this.
3. SAS 9.1 does not allow duplicate entries for a key; SAS 9.2 does.
4. SAS 9.1 cannot begin reading a hash sequentially after a lookup; SAS 9.2 does.
5. The SAS statements and method calls to create a hash object are "klunky", but somewhat repetitive.

6. The SAS 9.1 documentation (built-in and web) hides the hash information:

   SAS Products → Base SAS → SAS Language Dictionary → Appendixes
   → DATA Step Object Attributes and Methods.

   SAS 9.2 promotes this to a separate section of SAS 9.1 Reference: Dictionary.
   http://support.sas.com/documentation/cdl/en/lrdict/62618/HTML/default/titlepage.htm

How to Create a Hash Object. You must do these things in this order:

1. Declare the Hash Object with a DECLARE statement, and initialize it with an initialization call. (The SAS documentation lists these as two different steps, but in practice there is no reason to use the more complex _NEW_ syntax.)

```
IF _N_ = 1 THEN DO;        /* You must call the hash definition
functions only once! */
        DECLARE HASH SUM(HASHEXP:16);   /* 16 means 2**16 or 65536
hash buckets */
        etc . . .
END;
```

   As the comments above imply, it is your responsibility to call the initialization functions only once. If you forget this, SAS will create a new hash object with every observation, and you will quickly fill up memory!

   I included the HASHEXP:16 because in practice very few hashes are so small that you want the default of 8, which provides 2**8 or 256 hash buckets. HASHEXP:16 is the largest that SAS 9.1 allows, but SAS 9.2 allows 20, or about a million hash buckets. SAS 9.1 tolerates HASHEXP:20, but behaves as if you had specified 16. 16 hash buckets causes the hash to use about 500KB of memory overhead, not counting inserted items.

   The number of buckets does **not** limit the number of items you can store in a hash. But, a large number of buckets improves performance, because there are fewer items in each bucket through which SAS must search to find your item.

   The initialization call also lets you specify that the hash is to preserve a sorted order, or to pre-load the hash from a SAS dataset.

2. Define the hash key variables:

```
SUM.DEFINEKEY ('NAME');          /* Variable name is a quoted string
*/
```

   Tell the hash object what variable(s) you want to use as a key. If you have more than one, you can add more parameters, or call the DEFINEKEY() method more than once. But, you cannot use SAS variable lists!

   DEFINEKEY() takes the name of the variable, not the variable itself. This means that the SAS compiler does not necessarily know that NAME is a SAS variable. This is OK, because the hash keys are usually already defined in your data.

3. Define the hash data variables:

```
SUM.DEFINEDATA('NAME', 'TOTAL');
```

   Tell the hash object what variable(s) you want the hash to return when use the FIND() method, or any of the iterator methods. Only data variables are written to SAS datasets by the OUTPUT() method, or return by one of the hash iterator methods. If you want the hash keys included, you must define them as data variables, also.

Unlike keys, data variables often are not already defined to the SAS compiler, so you must separately define them, typically with a LENGTH statement. Also, the SAS compiler does not know that the FIND() method will modify these variables, so if you have no other code in your program changes them, the SAS compiler does complain that they are uninitialized. To avoid this, you can include a statement like: `IF 0 THEN TOTAL=.;`

4. Tell the hash object you have finished the definition:
   `SUM.DEFINEDONE();`

This is not exactly trivial, but at least is fairly constant from program to program. You can even write a macro to make this process easier.

Uses of Hash Objects: On-the-fly summarization

This example processes a SAS dataset named RAWFILE, and writes a dataset named DETAILS. The example could have also read raw data with an INPUT statement. The difference here is that at the same time it processes the data, this also summarizes it by a character variable named NAME, adding up VALUE as TOTAL. It could have kept counters or other statistics.

The advantage of this over a separate DATA step and a PROC SUMMARY is that the program does not have to pass the data twice. If DETAILS is a very large SAS dataset, it seems a shame to re-read it just to get a simple summary.

```
DATA DETAILS;
 SET RAWFILE END=END;
 IF _N_ = 1 THEN DO;        /* You must call the hash definition
functions only once! */
    DECLARE HASH SUM(HASHEXP:16);     /* 16 means 2**16 or 65536 hash
buckets */
    SUM.DEFINEKEY ('NAME');          /* Variable name is a quoted
string */
    SUM.DEFINEDATA('NAME', 'TOTAL');    /* NAME here to be written out
*/
    SUM.DEFINEDONE();
END;

 /* Other processing that may change VALUE or NAME */

 IF ~SUM.FIND() THEN TOTAL+VALUE; /* Non-zero indicates entry found */
                    ELSE TOTAL=VALUE; /* Zero indicates not found */
 SUM.REPLACE();    /* Replace() works OK whether entry exists or not */

 /* At the end, write the totals */
 IF END THEN SUM.OUTPUT(DATASET: 'TOTALS');
RUN;
```

Uses of Hash Objects: Merge without sorting.

To use the DATA step MERGE statement, all input datasets must be sorted or indexed by the key variables. This has several drawbacks:

1. Sorting takes time, and can waste disk space.
2. MERGE with an index is slow, not counting the time to build the index.
3. A single MERGE must have a common set of BY variables. You can partially get around this with DATA step views, but not if the merge keys are completely different.
4. Using SET with KEY to merge unsorted data is hard to write, and is not fast.

Hash object gets around these limitations. You simply place one or more datasets into hash objects, then use the has FIND() method to do the actual lookups. No sorting required. Here is a simple example:

```
DATA BIGFILE_PLUS;   /* Assume it is sorted by THISVAR */
 SET BIGFILE;
 IF _N_ = 1 THEN DO;
    DECLARE HASH OTHER(DATASET:'OTHER', HASHEXP:16);
    OTHER.DEFINEKEY ('OTHERVAR');
    OTHER.DEFINEDATA('OTHER_CHAR', 'OTHER_NUM');
    OTHER.DEFINEDONE();
    /* Must manually define variables in OTHER file */
       LENGTH OTHER_CHAR $12;
       IF 0 THEN CALL MISSING(OF OTHER_CHAR OTHER_NUM);  /* Avoid
uninit */
 END;
 /* Actually "Merge" */
 IF OTHER.FIND() THEN; /* Ignore FIND() return code */
RUN;
```

This example defines a hash object named OTHER whose key is OTHERVAR, and merges in the variables OTHER_CHAR and OTHER_NUM. It pre-loads its contents from the existing dataset OTHER, assuming that that each observation has a unique value of OTHERVAR. Then, for each observation read from BIGFILE, it uses the value of OTHERVAR to find an entry in OTHER. If one is there, then all of the other variables from OTHERFILE are copied to the current variables, otherwise they are left as-is. The return code from OTHER.FIND() is ignored in this example. This seems complicated, but the only difficult part is the hash-definition part, and they are all pretty much the same.

But, if you needed to know whether the item was found, like the IN= dataset option of a MERGE, you could test or capture the return code from OTHER.FIND(). Zero indicates success, non-zero failure. (This is counter-intuitive, but consistent with the C++ object-oriented philosophy on which the hash object is based.) Thus, you would test the result of a FIND() this way: (~ means NOT)

```
  IF ~OTHER.FIND() THEN action-after-sucess; /* zero RC = OK */
  ELSE action-after-failure;
```

Why test the "not" condition instead of 0=OTHER.FIND() ? Answer is a perhaps over-zealous "need for speed". I ran tests, and SAS processes the "not" faster than a comparison with zero.

Uses of Hash Objects: "Running" all items in a dataset in a single pass.

Sometimes, you need to read all of the items in a dataset, but you cannot process them sequentially like SAS normally does. You may need to "Run" them several times. The conventional way to do that is to have multiple DATA steps to read and reread the data. But, sometimes that introduces lots of new complexity.

Hash objects provide a simple way around that. At the same time you create the hash object, you can create a Hash-Iterator, or HITER object. All it takes is one extra statement:

```
    DECLARE HITER OTHERS('OTHER');   /* OTHERS iterates object in OTHER
*/
```

Then, when you want to iterate all of the objects in the OTHER hash:

```
 DO WHILE(~OTHERS.NEXT());   /* WHILE return code is zero */
     Any statements using OTHER_CHAR, OTHER_NUM;
 END;
```

The only unintuitive part of this is that the hash key is not automatically returned when you run the NEXT() method unless the key variable was also defined as a data variable. Most hashes with iterators include the key for that reason. Also, if you want the items returned in key order, you must include the parameter ORDERED: 'YES' in the original hash definition.

The ability to "run" a hash is really nice, because you can do it as often as you wish in a DATA step. Each time the NEXT() method runs past the end of the has, it resets itself, so the next time it runs, it returns the first element again. But what if you may not always run the hash all the way to the end? This idiom works if the starting point of the iterator is unknown:

```
IF ~OTHERS.FIRST() THEN DO UNTIL(OTHERS.NEXT());  /* 1st non-zero
return ends loop */
   IF whatever… THEN LEAVE;
END;
```

Notice that it calls the FIRST() method, which must return a zero return code to run the DO loop. This way, if the hash is empty, the loop does not run. Because the first entry is already obtained, the loop must wait until after the iteration to call the NEXT() method. The loop ends with the NEXT() method returns a non-zero indicating failure.

Uses of Hash Objects: Table Look-up vs. using a Format.

A table lookup is fairly much like a merge, but hash objects make great table lookup tools. For a table lookup, what are the differences between hash objects and formats?

1. As table-lookup tools, hash objects are faster, because hashing is faster than the binary search used by formats.
2. Hash object can return more than one result, a format can return only one (but you can have multiple formats with the same keys).
3. A hash object can have a multi-variable key; a format can take only a single variable. To do multi-variable keys with formats, you have to concatenate the pieces together yourself. That gets awkward if any part of the key is numeric.
4. **Important**: You can modify a hash object in-flight, you cannot modify a format except by rebuilding it in a separate PROC FORMAT step.
5. You can "run" all items in a hash object sequentially, you cannot with a format.
6. You can insert items into a hash object in any order, and have them returned in a sorted order in the same DATA step. This is the only way to do on-the-fly sorting in a DATA step.
7. A format is (or can be) permanently stored, a hash object exists only in memory during a DATA step.
8. A format can be used anytime by any PROC, including PROC SQL and in any SAS code by `%sysfunc(putN())`, a hash object can only be in a DATA step.

Under what circumstances should you not use a hash object?

1. As above, where you may need to do a table-lookup in a PROC or in macro language.
2. A hash object has no facility to do table lookups where the key is a value within a given range. Formats do that readily.
3. A novice SAS programmer can more easily code a PUT() function than use a hash object to do table lookups.

Under what circumstances should you not use a format?

1. Where you need to insert items into the lookup table on-the-fly. You cannot modify a format inside a DATA step.
2. As above: multiple keys or multiple variables to be returned.

Other good stuff: changes in SAS 9.2.

SAS 9.2 allows larger hashes (1 million buckets vs. 65,536).

SAS 9.2 allows you to store multiple items with the same key. (In SAS 9.1, you can define an artificial sequence variable as the last variable of a key to get the same capability.)

SAS 9.2 allows you to find an item, then traverse the hash from there forwards. (SAS 9.1 would make to iterate the entire hash to find the item you want.)

SAS 9.2 allows you to CLEAN() a hash without deleting and re-defining it. This make it faster for you to reset the hash for every BY-group. SAS 9.1 hashes can be cleaned with the following idiom, assuming **HASH** has keys **KEY1** and **KEY2**, both defined as data variables, and an iterator object named **HITER**:

```
IF ~HITER.FIRST() THEN DO UNTIL(LAST._CLEAN);   /* LAST._CLEAN drops
itself */
    _KEY1 = KEY1;        /* Save keys because NEXT() method will step
on them */
    _KEY2 = KEY2;
    LAST._CLEAN = HITER.NEXT();   /* last._clean false until NEXT()
goes past last entry */
    HASH.REMOVE(KEY:_KEY1, KEY:_KEY2);  /* Remove previously-found item
*/
END;
DROP _KEY1 _KEY2;
```