Paper 3110-2019

The SAS® Hash Object: Well Beyond Table Lookup

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

ABSTRACT

Most applications of the SAS® hash object focus on its table lookup facility, mainly to combine data. While it is indeed a great lookup tool, its functionality is much broader: As a data structure, the hash object comes equipped with all common table operations, which means that it can be also used for data aggregation, splitting, unduplication, and so on. And, since these operations are executed in memory in constant time, they tend to outperform other SAS alternatives. Furthermore, the run-time nature of the hash object, its autonomous I/O facility, and its ability to point to other objects make it possible to write dynamic and easily parameterized programs without hard coding. In this paper, we outline the functionality of the hash object well beyond table lookup as a compendium of the book "*Data Management Solutions Using SAS Hash Table Operations. A Business Intelligence Case Study*" recently published by SAS® Press.

INTRODUCTION

Since its inception in version 9.0, the SAS hash object has matured from its rather humble beginnings into a versatile programming tool. Thanks to the work of creative programmers who have used it in imaginative ways in a variety of industries, many new techniques have been developed based on its properties and dynamic capabilities. Their sheer diversity would make merely listing hash-based techniques, even illustrated with examples, look like a hodgepodge. Besides, it would unnecessarily duplicate work already presented in the existing SAS literature on the subject. Rather, it appears that at this point, a unified picture of what can be done using the hash object based on some classification method could be useful - both to the people yet unfamiliar with the subject and those well-versed in it. This paper is an attempt to provide such a picture by classifying hash object techniques from the standpoint of general data processing table operations the hash object can help accomplish.

SAMPLE DATA IN BRIEF

The sample data and some code snippets used in this paper are from a set of programs that will be used to create sample data for a SAS Press book that will address the broad functionality supported by the SAS hash object. The data to be generated is for a fictitious game called "*Bizarro Ball*" (aka "BB") and is conceptually similar to Baseball, with a few wrinkles. It includes information for teams, players, and games. The game data will include a row for each pitch, at bat and runner. Running the programs as directed (which takes mere seconds) outputs a number of tables in the library "*Bizarro*", rich enough to illustrate different aspects of the hash object's functionality. Also, the programs themselves use the hash object as a tool to generate the sample data and include a number of techniques discussed below.

The detailed description of the programs and their location, as well as the sample data, can be found at the end of the paper in the section "Sample Data and Programs".

HASH OBJECT PROPERTIES

The nature of the data processing tasks that can be addressed using the hash object as a tool is in turn dictated by its nature. Let us, therefore, first take a look at the hash object and its properties at a very high level.

THE HASH OBJECT IN A NUTSHELL

The hash object is a dynamic data structure controlled from the DATA step (or the DS2 procedure) environment during the execution time. It consists of the following principal elements:

- A hash table in memory for data storage and retrieval, organized to perform table operations based on accessing the table directly via its key.
- An underlying hashing algorithm which, in tandem with the specific table organization, facilitates this access quickly and efficiently, and whose run time scales as *O(1).*
- A set of tools to control the existence of the table, that is, to create and delete it.
- A set of tools to activate the table operations based on key access.
- Optionally, a number of tools used without using a key, such as the hash iterator object linked to the hash table and accessing the table sequentially.

THE HASH TABLE

The hash object table "looks" just like any table with columns and rows, but with a few marked distinctions, in particular:

- It resides completely in memory (RAM, main storage, depending on the operating system).
- Its columns are called *hash variables*, and its rows are termed *hash items*.
- At least one variable is designated as the *key portion*, and at least one variable - as the *data portion*. Together they constitute what is called the *hash entry*.
- All hash variables *must* be defined in the Program Data Vector (PDV, *[1]*) at compile time as the hash table's *host variables* (see the *caution note* below the list). It is from these variables that the hash variables inherit their attributes.
- The key portion can consist of one variable (*simple key*) or two or more variables (*composite key*). The key portion variables can be only *scalar*, i.e. numeric or character.
- The data portion can also contain more than one variable, character or numeric. However, unlike the key portion, *it can also contain variables of the type object*, in particular, other hash object instances.
- Only the data portion hash variables can update their host counterparts in the PDV.
- The table is dynamic: It grows/shrinks at run time as entries are added/removed, and the memory needed to store the entries is acquired/released accordingly.

Caution Note: It may appear that it is possible to create the *key portion only* without creating the data portion because after calling the DEFINEKEY method calling *DEFINEDATA can be omitted*. However, the stand-alone DEFINEKEY has the effect of *automatically creating the data portion with the same variables*. It may come in handy and shorten code *when this is exactly what you want*. However, it may also have a *negative impact on the memory* footprint (see *Memory Management* section).

THE HASH ALGORITHM

The hash algorithm is a specific internal mechanism, by means of which the hash table is searched based on its key. From the standpoint of using the hash object (and this paper), the gory details of the algorithm are rather unimportant, all the more that they are well described in a number of sources. What is of interest here are those properties of the algorithm which affect the speed and scalability of the hash table key-access operations, as well as some of its useful side-effects. In particular:

- Hash search is fast - at least on par with other SAS methods based on searching in memory.
- Its run time scales as *O(1)*. In layman terms, it means that an act of hash search does not depend on the number of table items: It is equally fast whether the table contains 100 or 1,000,000 of them.
- Since the response of the hash function used by the algorithm is random by its very nature, it places the items into their internal locations in the table randomly, unless directed otherwise by the value given to the argument tag ORDERED.

HASH TABLE OPERATIONS - OVERVIEW

The hash object methods support "standard" common table operations, all performed at the DATA step run time. They can be divided into two distinct groups. The operations in the larger group are based on searching for a given key first to locate an item or a set of items. The operations in the other group access

the table without the need to supply a key sequentially or en masse. Subdivided in this manner, they are listed in the table below:

Table Operation	Action Description
Key-Based	
Insert	If key not is in table, add new item with this key.
Search	Check if key is in table. Do *not* overwrite data portion host PDV variables.
Delete Items	If key is in table, remove all or some items with this key.
Retrieve	If key is in table, overwrite host PDV variables with their data portion variables.
Update	If key is in table, overwrite data portion variables with their PDV host variables.
Enumerate by Key	If key is in table, list and *retrieve* (see above) all or some items with this key.
Start iterator	If key is in table, place iterator pointer at item with this key.
Order	Force items to *enumerate* in given key order.
Not Key-Based	
Create hash object	Declare hash object, define its entries, create an instant of it.
Create iterator object	Declare iterator object and tie it to a hash object.
Delete object	Delete hash or iterator object. Deleting hash also deletes its iterator.
Delete items	Delete all hash items *en masse* without deleting object itself.
Enumerate Serially	List one or more items serially from table start, end, or given key. For each item listed, overwrite host PDV variables with data portion variables.
Output	Write data portion variables of all table items to SAS data set *en masse*.
Compare	Determine if two hash objects are equal.
Describe	Get table attributes, such as number of items and entry length.

Such a set of table operations is rich enough to perform just about any data processing task. The hash object tools calling them to action are the *statements, methods, operators, attributes,* and *argument tags*. The latter are somewhat similar to SAS options, in that they modify the actions of statements, methods, and operators.

"DYNAMIC"

We have referred to the hash object as *dynamic* (and will do it again), but it is worth adding a few words to define more precisely what it means, as many hash-based techniques are rooted in this dynamic nature either implicitly or explicitly:

- All the actions related to the hash object, including its creation and deletion, are executed at run time rather than compile time.
- The hash table can grow and shrink at run time, as memory needed for its items is acquired or ceded as the items are added or removed. Contrast that with such look-up media as an array, whose number of items and memory are pre-allocated at compile time, or a character string, whose length is predetermined at compile time as well.
- The arguments given to the argument tags, contrary to a seemingly common illusion, do not not have to be hard-coded literals. Rather, the argument tags are designed to accept general SAS expressions

of the correct data type. (The reason the literals are okay, too, is that they are also SAS expressions, albeit the simplest ones). It means that the actions of the statements and methods can be modified dynamically and conditionally based on the corresponding values of variables currently in the Program Data Vector (PDV, see *[1]*). E.g., if you code `h.find(key:A*B)`, then if at the time of the call A=3 and B=5, the method will search the table for key=15.

- The hash object's *I/O facilities work independently* from those of a DATA step from which they are called - and of course also at run time. It means, for example, that if a line of code directs the hash object to write its table's content to a SAS data file, it will handle this entire operation and *close the file* before program control is passed to the next executable line, and then the DATA step will keep running - in sharp contrast with a data file named in the DATA statement, which gets closed only after the step has stopped its execution.

CLASSIFICATION

The reason we have dwelled on the hash table operations is that the hash-derived data processing techniques can be classified based on them. Of course, it is not the only classification method - and, depending on personal preferences, maybe not the best one. An obvious problem with it is that most techniques involve a combination of more than one operation. However, usually one of them plays the dominant role, so we can structure the narrative by keeping that in mind. Any reasonable classification system is better than none.

KEY-BASED HASH TABLE OPERATIONS

We will begin with the key-based operations, followed by those not based on using a key. The emphasis will be placed not on the syntax or minute details of the hash object tools being used - they are described in the related SAS documentation - but on the data processing result to be achieved. However, in those instances, when we feel the documentation is not clear in some respect or does not illuminate the action of a certain feature amply enough, we will call attention to the fact.

INSERT OPERATION

Another term for this operation is table load. The very purpose of a hash table is to contain data that can be efficiently compared to values outside the table. Hence, SAS provides a variety of tools and options facilitating the insertion of keys and data in a variety of ways:

- from hard-coded key and data values, one at a time or in a DO loop
- from expressions given as arguments to the argument tags KEY and DATA, via successive statements or in a DO loop
- from a data set in the implied DATA step loop or explicit DO loop
- from a data set via an expression given to the DATASET argument tag, with data set options if need be
- accepting duplicate key items using the MULTIDATA argument tag
- rejecting duplicate key items and controlling the corresponding actions through the choice of a method or the DUPLICATES argument tag
- forcing the table items into an order by the key or leaving the order undefined (random)

Trivial Unduplicated Insertion

The following snippet from the program *S0100-GenerateTeams.sas* is a typical example of trivial unduplicated insertion. Here, the purpose is to load the table with a set of unique team names and add a surrogate key for each to the data portion:

```
if _n_ = 1 then do ;
   dcl hash teams() ;
   teams.defineKey  ('Team_Name') ;
```

```
   teams.defineData ('Team_SK','Team_Name') ;
   teams.defineDone () ;
end ;
infile datalines eof = lr ;
input Team_Name $16.;
Team_SK + ceil (ranuni (&seed1) * 4) ;
rc = teams.ADD() ;
```

Let us highlight a few relevant features and make some notes:

- The team names are read in-stream via the DATALINES statement.
- Since the hash TEAMS is declared without the argument tag MULTIDATA:"Y", every time the ADD method is called, it first searches the table implicitly to see if Team_Name key is already there. If so, the method ignores the entry; otherwise, the item is inserted into the table. The RC= assignment prevents the duplicate key log error message - and the step from being aborted.
- Since no ORDERED argument tag is present in the hash declaration, either, the keys are inserted into the table, thanks to the underlying hash function, *completely at random*. This is very convenient because this is what we want here and it rids us from introducing an extra random key to sort by.

Duplicate-Key Insertion

In many different situations, it may be desirable to load a table with duplicate key items. An example of it can be found in this snippet from the program *S0400-AssignPlayersToTeams.sas*:

```
dcl hash available  (DATASET:"bizarro.player_candidates", MULTIDATA:"Y") ;
available.defineKey ('Position_Code') ;
available.defineData('First_Name','Last_Name') ;
available.defineDone() ;
```

The reason why duplicate-key items need to be loaded in this case is that the program later on needs to process all the players with the same position using *keyed enumeration* - in other words, to "*harvest*" the items with the same key. This excerpt differs from the unduplicated insertion snippet above in two aspects:

- The table is loaded from an existing data set by giving the argument tag DATASET its name. Actual input occurs at run time when the DEFINEDONE method is called, and the whole table is loaded before the next executable line of code.
- The argument tag MULTIDATA:"Y" ensures that the ADD method works behind-the-scenes to insert the items without looking if the keys are already in the table.

Insertion Methods Other than ADD

The ADD method is not the only one that facilitates the insert operation: The REPLACE and REF methods may be more convenient depending on the circumstances, and here is why. With MULTIDATA:"N" (or not specified) input will be auto-unduplicated in any case but with subtle differences, depending on the method used:

- REPLACE overwrites the data portion of an item if its key is in the table. If not, it simply inserts the new item. Hence, for a number of identical input keys, it does it every time for the same input key, and data portion ends up loaded with the values for this key that come in *last*. As opposed to that, ADD results in loading the data values coming *first*. Also, unlike ADD, REPLACE is always successful by its nature and does not generate error messages if coded stand-alone, i.e. without coding RC=.
- REF always first checks if the key is already in the table. If so, no insertion is done, so the end result is the same as with ADD. The difference is that REF, just like REPLACE, is also always successful. So, in the example of trivial unduplicated insertion above, REF could be used without RC= instead of ADD.

On the other hand, coding MULTIDATA:"Y" alters the behavior of all three methods:

- ADD is always successful and merely adds an item, whether its key is already in the table or not.
- REPLACE behavior depends on the SAS version (9.4 or earlier). We will discuss it later.
- REF in this case is quite different. Because it inserts an item *only if the key is NOT in the table,* it always inserts only the first duplicate-key item (regardless of the MULTIDATA specification).

SEARCH (TRIVIAL TABLE LOOK-UP) OPERATION

Search (or *pure search* or *trivial table look-up*) is needed only to find out whether a given key is in the table - and nothing else. As already mentioned, all other key-based operations also involve search as their first stage. However, in some situations they may have undesirable side effects - in particular, they overwrite the data portion in the table or their PDV host variables if the key is found. Also, because pure search does not involve any data movement between the data portion of the table and the PDV, it executes faster.

Keeping Track of Keys

A good example of using pure search purposedly can be found in the program *S0600-GenerateSchedule.sas*. There, the hash table USED keeps track of the league and team pairs already encountered during the execution:

```
dcl hash used  (multidata:'y', ordered:'a') ;
used.defineKey ('League', 'Team_SK') ;
used.defineDone() ;
```

The pure search call **`if used.check()`** is then used to determine if a pair of (*League,Team_SK*) values is in the table without overwriting the corresponding PDV host variables, and only the pairs not yet encountered are added to the table using the call **`used.add()`**.

Note: Above, there is no call to DEFINEDATA after calling DEFINEKEY. This automatically places variables (*League,Team_SK*) *in the data portion as well*.

Sortless Stable File Unduplication

Another data processing case where we need only pure search is unduplicating a file in a single pass without sorting. For example, suppose that it has struck our fancy to read data set *bizarro.Players* and output only the records where each unique first name is encountered first. Moreover, we want the record thus retained to be in the same exact relative order as in the original file - in computer science, the latter arrangement is called "*stable*". In other words, out of the following snapshot:

Team_SK	Player_ID	First_Name	Last_Name	Position_Code
165	10124	Alan	Lee	SP
165	10133	John	Watson	SP
219	10150	Alan	Cooper	SP
219	10158	Philip	Morris	SP
219	10176	Johnny	Morris	SP
219	10189	Alan	Ramirez	SP
219	10192	Philip	Stewart	SP
219	10202	Randy	Hall	SP
158	10206	Randy	Morgan	SP

we would like to eliminate the records with Cooper, Ramirez, Stewart, and Morgan, otherwise leaving the record sequence the same. Why, it is easy:

```
data nodup ;
   if _n_ = 1 then do ;
      dcl hash h() ;
      h.definekey ('first_name') ;
      h.definedone() ;
   end ;
   set bizarro.players ;
   if h.CHECK() ne 0 ;
   h.ADD() ;
```

```
run ;
```

The logic is simple: Search H for the key and if it is not there, insert it and output the record; otherwise read the next one. All that takes is a single pass through the input data. Without using the hash object, we would have to go through the sort-dedup-sort cycle, at the cost of at least two more data passes.

File Subsetting

Subsetting a file based on the keys stored in a hash table also involves nothing more but pure table look-up. Imagine that we want to get all the batters whose first name is John from data set *bizarro.batters* (for the sake of brevity, only the records with the batting order 1-3 are shown):

Game_SK	Team_SK	Batting_Order	Player_ID
FC4C07F4D2D46E89FB24D5E3396859	317	1	16725
FC4C07F4D2D46E89FB24D5E3396859	317	2	14351
B45C70C1C159FEC5F0272022023DD68	251	1	14207
B45C70C1C159FEC5F0272022023DD68	251	2	15530
8D9E88F9D7EA840BA0A13C5AFDDA6	224	1	16267
8D9E88F9D7EA840BA0A13C5AFDDA6	224	2	15089
C26FC66B16B4890590B6D397637B664	193	1	14152
C26FC66B16B4890590B6D397637B664	193	2	16502
555D7E706F484C873A9C6DA3524C9C5	246	1	17529
555D7E706F484C873A9C6DA3524C9C5	246	2	14905

However, the player names are stored in the dimension table *bizarro.players*, a snapshot of which was shown in the subsection above. The two tables are linked by the key tuple (*Team_SK,Player_ID*). Now we can store the keys where `first_name='John'` in a hash table and, for each record in *bizarro.batters*, search it to see if the record key is there. If so, the record is written out:

```
data batter_john ;
   if _n_ = 1 then do ;
      dcl hash h(dataset:"bizarro.players(where=(first_name='John'))") ;
      h.definekey('team_sk', 'player_id') ;
      h.definedone() ;
   end ;
   set bizarro.batters ;
   if h.CHECK() = 0 ;
run ;
```

Of course, this is the same you would get from an SQL join (or subquery, or MERGE). Compared to SQL, the advantage of using the hash object is that if desired, data processing can continue in the same step without writing interim output. Compared to MERGE, the hash object rids us from sorting either input file. In addition, the hash solution is likely to run faster overall than either SQL or MERGE.

RETRIEVE OPERATION

From the standpoint of using the hash object, this operation means the following: (a) search the hash table for a key and (b) if the key is found, overwrite the PDV host variables with the values of their hash table counterparts.

Plain Join

The ability of the hash object to improve performance joining files more efficiently than other SAS methods was its first claim to fame. The logical coding scheme for joining is not much different from subsetting. The difference is that to join files, the retrieve operation, rather than the search operation, is used. That is, when a key is found in the table, the data portion variables, loaded there from one of the files, update the PDV host variables.

As an illustration, let us modify the subsetting task described above by adding the requirement that the player's last name and position code be added to the output. To do that, we need to store variables *Last_name* and *Position_code* in the data portion of H, which in turn requires placing these variables in the PDV beforehand at compile time as host variables. Below, it is done using IF 0 THEN SET at _N_=1,

thus letting the DATA step compiler read the descriptor or file *bizarro.players* and place the needed variables in the PDV without reading its data.

```
data batter_john_lname_poscode ;
  if _n_ = 1 then do ;
     IF 0 THEN SET bizarro.players (keep = last_name position_code) ;
     dcl hash h(dataset:"bizarro.players(where=(first_name='John'))") ;
     h.definekey('team_sk', 'player_id') ;
     h.DEFINEDATA('LAST_NAME', 'POSITION_CODE') ;
     h.definedone() ;
  end ;
  set bizarro.batters ;
  if h.FIND() = 0 ;
run ;
```

Here the *retrieve* operation is implemented by calling the FIND method, which - unlike CHECK - is designed to update the host variables with their data portion variables upon finding the key from the current record in *bizarro.batters*.

Multiple Look-up Tables

Naturally, this join scheme is not limited to retrieving data just from one hash table. For example, if above, in addition to getting the last names of all batters named John, we also want to add their team names and leagues to the output, we can load *bizarro.teams* into another hash and look it up during the same pass:

```
data batter_john_more_info ;
  if _n_ = 1 then do ;
     IF 0 THEN
        SET bizarro.players (keep = last_name position_code)
            bizarro.teams   (keep = team_sk team_name league)
     ;
     dcl hash H  (dataset:"bizarro.players(where=(first_name='John'))") ;
     h.definekey ('team_sk', 'player_id') ;
     h.DEFINEDATA ('LAST_NAME', 'POSITION_CODE') ;
     h.definedone() ;
     dcl hash T  (dataset:"bizarro.teams") ;
     t.definekey ('team_sk') ;
     t.definedata ('TEAM_NAME', 'LEAGUE') ;
     t.definedone() ;
  end ;
  set bizarro.batters ;
  if h.FIND() = 0 ;
  call missing (team_name, league) ;
  _IORC_ = t.FIND() ;
run ;
```

The host variables for the new hash T *must* be prepared at compile time, which is done by letting the compiler read the descriptor - but not the data - of data set *bizarro.teams* via IF 0 THEN SET maneuver. The actual data from this data set are read by the hash object T all by itself.

As a small side note, using *_IORC_* as a return code variable is nifty because there is no other use for it in the step, it is automatically dropped - and is also kind of aptly named.

Cascaded (Dimensional) Join

Above, we retrieved the data from two different hash tables H and T using two different keys coming from the same file *bizarro.batters*, read sequentially. A somewhat different situation arises in data warehousing star schemas when one dimension contains a key for yet another. For example, in our sample data, data set *bizarro.position_dim* serves as a dimension for *bizarro.players*, holding the information on the players' positions by key variable *Position_code*:

	Position_Code	Position	Count
1	SP	Starting Pitcher	5
2	RP	Relief Pitcher	5
3	C	Catcher	3
4	IF	Infielder	4
5	OF	Outfielder	3
6	UT	Utility	2

Suppose that we want to get all batters named John from *bizarro.batters* and also get their actual positions. In order to tie them to *bizarro.batters*, we need to (1) join it with *bizarro.players* first to retrieve *Position_code* as data from H (as done above) and then (2) use it as the key into another hash table P loaded from *bizarro.position_dim*.

```
data batter_john_positions ;
  if _n_ = 1 then do ;
     IF 0 THEN
        SET bizarro.players (keep = last_name position_code)
            bizarro.positions_dim (keep = position count)
     ;
     dcl hash H  (dataset:"bizarro.players(where=(first_name='John'))") ;
     h.definekey ('team_sk', 'player_id') ;
     h.DEFINEDATA ('POSITION_CODE') ;
     h.definedone() ;
     dcl hash P  (dataset:"bizarro.positions_dim") ;
     p.definekey ('POSITION_CODE') ;
     p.definedata ('POSITION', 'COUNT') ;
     p.definedone() ;
  end ;
  set bizarro.batters ;
  if h.FIND() = 0 ;
  call missing (position, count) ;
  _iorc_ = p.FIND() ;
run ;
```

Needless to say, such cascaded retrieval is not limited to two-link dimension chains and can be extended in the same vein to drill down into a chain of any practical length.

Many-to-Many, Full, Outer Joins

Note that in the schemes shown in this section thus far, the items inserted into the hash tables are automatically unduplicated while being loaded. Therefore, the types or join that can implemented in this manner are one-to-one or many-to-one equijoin, left join, or right join. Implementing other types of joins, such as many-to-many, full, and outer, is also possible by letting the hash tables contain duplicate-key items and using the *keyed enumeration operation* against the tables. We will dwell on this operation later on in this paper. However, as far as the other join types are concerned, their details are already described elsewhere in the SAS literature (see, e.g., *[2]*) that it makes no sense to duplicate the effort. So, we are merely pointing the reader to the reference.

Retrieve Operation at Large

While joins are a good example of using the retrieve operation, it is universal in the sense that it is equally applicable in any programmatic situation that dictates pulling data values from a hash table into the PDV for a given key. Rather elaborate instances can be found of using it under circumstances that have nothing to do with joins- for example, in the programs generating the sample data for this paper.

Also, the retrieve operation is one critical part of using the hash object for data aggregation. This is because accumulating aggregates in a hash table necessarily involves two-way data traffic between the data portion hash variables and their PDV host counterparts. We will see how and why later on.

DELETE ITEMS OPERATION

As far as the hash object is concerned, this operation means the following: (a) search the hash table for a key, and (b) if the key is found, remove the corresponding item or items from the table. The operation elicits no data traffic between the data portion of the table and the PDV host variables.

The REMOVE hash method used to achieve the goal deletes all items with a given key from the table, no matter whether the items with this key are unique or not. Optionally, the REMOVEDUP method, working in tandem with keyed enumeration, can delete specific items from a same-key group of items depending on the programming logic and/or values of the data portion variables.

Exceptive Join / Subsetting

A fairly typical situation in which the *delete item* operation is of value is when we want to use a hash table for a join or subsetting *but* exclude some keys beforehand based on another piece of information. For example, suppose that we need to get all information from *bizarro.players* for each matching record from *bizarro.pitchers* by (*Team_SK, Player_ID*). However, we have a third file, *Exceptions*, listing the team and player pairs we are *not interested* in for some reason. A few records from it may look like this:

Team_SK	Player_ID
158	15913
171	12059
193	14175
281	14599
339	17166
342	16461
344	16206

To achieve the goal, we can load the whole of *bizarro.players* into the hash table, remove the exception key pairs from the table in place, and then proceed with the match:

```
data non_except_pitchers ;
  dcl hash P (dataset: "bizarro.players") ;
  p.definekey ("team_sk", "player_id") ;
  p.definedata ("first_name", "last_name", "position_code") ;
  p.definedone() ;
  do until (last) ;
     set exceptions end = last ;
     _iorc_ = p.REMOVE() ;
  end ;
  last = 0 ;
  do until (last) ;
     set bizarro.pitchers end = last ;
     if p.find() = 0 then output ;
  end ;
  stop ;
  set bizarro.players ; *get host variables;
run ;
```

Of course, the exception items could be handled differently. For instance, SQL or other SAS tools could be used to create an intermediate file, with the unneeded key records eliminated. However, doing it right in the already loaded hash table eliminates the need for the interim I/O and handles everything in a single step.

Grow and Shrink Data Structures

The ability of the hash object to add and delete items at run time and acquire and release requisite memory accordingly lends itself to the implementation of dynamic data structures absent in the DATA step before the advent of the hash object. One of such structures is a *stack*, which is a list of items operating on the principle LIFO (last in - first out). It encompasses two actions:

1. *Push*: An item is added onto the top of the stack.
2. *Pop*: The item pushed in last returns its value to the PDV and is taken off the stack.

Of course, it can be implemented using an array. The problem is, when you do not know how many items will have to be pushed, you do not know how to size the array. Not so with the hash object - it does not have to be sized beforehand. Below, numeric items 111, 222, 333 are first *pushed* onto the stack in a loop, then the next loop *pops* them one at a time:

```
data _null_ ;
  dcl hash S  () ;
  S.definekey ("seq") ;
  S.definedata("item") ;
  S.definedone() ;
  call missing (seq, item) ;
  put ">>> Pushed on stack: " @ ;
  do item = 111, 222, 333 ;
     S.ADD (key:S.num_items + 1, data:item) ;
     put item @ ;
  end ;
  put "<<< Popped off stack: " @ ;
  do until (S.num_items = 0) ;
     S.FIND(key:S.num_items) ;
     S.REMOVE(key:S.num_items) ;
     put item @ ;
  end ;
run ;
```

The message printed in the SAS log is:

```
>>> Pushed on stack: 111 222 333 <<< Popped off stack: 333 222 111
```

The *delete item* operation actuated by the REMOVE method is critical here. Note how handily the hash operator NUM_ITEMS comes in thanks to the fact that the value it returns automatically grows and shrinks according to the current table size. Also note how the option of giving a method a key via an expression assigned to the argument tag makes code more compact - and also makes it superfluous to use extra variables outside the hash object to keep track of the key variable SEQ. Finally, due to the nature of the algorithm, all method calls are successful and can be used stand-alone (with no RC= assignment).

More information about using the hash object to implement stacks and queues can be found in [*12*].

UPDATE OPERATION

From the standpoint of using the hash object, this operation means the following: (a) search the hash table for a key, and (b) if the key is found, overwrite the data portion hash variables with the values of their PDV host counterparts.

One obvious application of this principle is the ability to update a hash table in place. The other is less obvious but much more important, since it gives the hash object the ability to aggregate data. Let us consider them one at a time.

Updating a Hash Table in Place

Suppose it has turned out that some players' last names in *bizarro.players* have been entered incorrectly, and before joining the file with *bizarro.pitchers* (see the previous subsection), they need to be corrected from an *errata sheet* file called *players_errata*, whose partial snapshot looks like this:

Team_SK	Player_ID	Last_Name	Last_Name_In_Error
342	11206	Clarke	Clark
165	10108	Grey	Gray
161	10060	Moure	Moore
339	10909	Riveera	Rivera

One way of doing this is to (1) create an interim file with the names updated from the errata sheet file, and then (2) use this new file in the hash join. However, the hash update operation makes (1) unnecessary:

```
data pitchers_join_noerrata ;
   dcl hash P (dataset: "bizarro.players") ;
   p.definekey ("team_sk", "player_id") ;
   p.definedata ("first_name", "last_name", "position_code") ;
   p.definedone() ;
   do until (last) ;
      set players_errata (drop = last_name_in_error) end = last ;
      _iorc_ = p.REPLACE() ;
   end ;
   last = 0 ;
   do until (last) ;
      set bizarro.pitchers end = last ;
      if p.find() = 0 then output ;
   end ;
   stop ;
   set bizarro.players ; *get host variables;
run ;
```

Using the REPLACE hash method to enact the update operation enables us to load *bizarro.players* into its hash table first and update it in place without the need of an extra step and/or interim output.

Another practical application of the hash update operation in the same vein is updating star schema dimensions loaded into hash tables and then using them in the same process to update the fact table.

Data Aggregation

While SAS is home to a wide variety of data aggregation methods, the hash object stands out as a vehicle which under certain data processing scenarios can ride where others cannot and, on the average, deliver more payload faster. The tandem of the update and retrieve operations is what makes it tick.

The goal is to aggregate the unsorted input data with multiple per key into the data portion of a single hash item per key. As a result, different hash data variables will have contained different aggregates, all obtained in a single pass through the input data. At that, the aggregation is not limited to additive types, such as straight sums or counts, but can include non-additives - like the number of unique values ("count distinct") or, if it should strike one's fancy, a product or something weird else.

As a representative example, let us take sample data set *bizarro.atbats* and, for each batter, compute RBIs (number of runs) and Games (games played by each batter - an example of "count distinct"). This is a severe reduction of a complete, "baseball-realistic" sample program *SampleMetrics* packaged with the paper:

```
data _null_ ;
   dcl hash H  (ordered:"a") ;
   H.defineKey ("Batter_ID") ;
   H.defineData("Batter_ID","Games","RBIs") ;
   H.defineDone() ;
   dcl hash U  () ;
   U.defineKey ("Batter_ID","Game_SK") ;
   U.defineDone() ;
   do until (lr) ;
      set bizarro.atbats end = lr ;
      rc = H.FIND() ;
      Games + (U.add() = 0) ;
      RBIs  + Runs ;
      H.REPLACE() ;
   end ;
   H.OUTPUT(dataset:"Stats") ;
   stop ;
run ;
```

Here is what the program does, in a nutshell:

- The "main" hash table H is used to store the aggregates by key *Batter_ID*.

- An extra table U is used to keep track of unique occurrences of (*Batter_ID, Game_SK*) pairs to compute metric *Games*, which is the *number of distinct games* each batter has played.
- Every time FIND is called, it overwrites PDV host variables RBIs and Games with their hash values accumulated thus far. Then the current record values are added to them.
- REPLACE sticks the new aggregate back into the table, overwriting previous values there.
- At end, table H with the final aggregates is written to data set STATS using the OUTPUT method.

This example is given here in a reduced form because its goal is just to demonstrate the basic principle of using a hash table for data aggregation - and show why the update operation in tandem with the retrieve operation is crucial for the task. It also presents a typical case of using *two-way data traffic - from the PDV to the hash and back* - to accomplish it. In many other data aggregation respects we have only scratched the surface of the subject. Its rather elaborate treatment, replete with a discussion of the advantages and shortcomings of using the hash object for data aggregation, can be found in [*3*].

Also, we will use another, expanded version of this example to support the discussion of the keyless enumeration operation later on in the paper.

ENUMERATE BY KEY OPERATION

Before SAS version 9.2, the hash object had been able to store only items with unique keys. In order to handle duplicate key items, programmers had to resort to a number of kludges, including the adding of another, unique, key to the key portion and using extra hash tables to track its values. The introduction of the argument tag MULTIDATA:"Y" made it possible to store items with duplicate keys directly in the table, and the addition of the associated hash methods - to access and process them. Together, they enable a programmer to use a key to point to a same-key group of items and to step through all the items in the group to *retrieve*, *update*, or *delete* items by choice. In other words, it became possible to *enumerate* them and perform the basic table operations described above on selected items with the same key value.

Keyed versus Keyless Enumeration

Enumerating hash entries *with a given key* and using corresponding methods to do it is the operation of *enumerating by key* or *keyed enumeration*. On the other hand, there exist scenarios when some or all items in the table need to be accessed sequentially starting from one of its endpoints. Such *keyless enumeration* is not supported by the methods supplied with the hash object and requires a *hash iterator object* to be linked in. We will discuss the latter and its methods in the corresponding section below.

Traversing Duplicate-Key Items

If a hash table is set with the unique-key mode and a given key is in the table, the retrieve operation done by the FIND method points to the item with that key, then grabs the data portion values and overwrites the host variables with them. If the table contains multiple items for the same key, the FIND method does the same, *except* that now it points only to the *logically first item* in the same-key group. To perform the same operation on the rest of the items in the group, the FIND_NEXT method must be used. Every time it is called, it steps to the next item. When the method returns a non-zero code, there are no more items to retrieve. Let us illustrate it with a simple example.

Here is an image of a tiny subset from *bizarro.players* for only two teams, *Titans* (player_id=103) and *Bears* (147), and outfielders (*Position_code*="OF"):

Position_Code	Team_SK	Player_ID	First_Name	Last_Name
OF	147	16040	Patrick	Hill
OF	147	16044	Jason	Lee
OF	147	16053	Jesse	Brown
OF	147	16056	Sean	Rodriguez
OF	103	16453	Gerald	Phillips
OF	103	16455	Ryan	Smith
OF	103	16456	Arthur	Jones

Suppose that *the whole data set* (for whatever reason) is loaded into a hash table P as follows:

```
dcl hash P  (dataset:"bizarro.players", multidata:"Y") ;
P.defineKey ("position_code") ;
P.defineData("position_code","player_id","team_sk") ;
P.definedone() ;
```

The table is keyed by *Position_code* - clearly, a duplicate key. Now imagine that, for the teams *Titans* and *Bears*, we need to find all outfielders (*Position_code*="OF"), list all the respective hash items, and write them out as records in data set *Outfielders*. One way to do it is to use the combination of the FIND method to point to the first duplicate-key item and then use the FIND_NEXT method to harvest the rest:

```
data outfieders (drop = rc) ;
  dcl hash P  (dataset:"bizarro.players", multidata:"Y") ;
  P.defineKey ("position_code") ;
  P.defineData("position_code","player_id","team_sk") ;
  P.definedone() ;

  position_code = "OF" ;

  do rc = p.FIND() by 0 while (rc = 0) ;
     if team_sk in (103, 147) then output ;
     rc = p.FIND_NEXT() ;
  end ;
  stop ;
  set bizarro.players ;
run ;
```

Above, FIND accepts the key value from the PDV host variable Position_Code, retrieves the data portion values into their host variable, and sets RC=0. Then FIND_NEXT is called repeatedly in the DO loop, which causes program control to step down the table for the same key one item at a time and retrieve the data portion values for each into the PDV. When there are no more items to retrieve, a non-zero value is returned to RC, and the loop stops iterating.

Note that the same thing could be done more elegantly if the table P were keyed by the *composite key* (*Position_Code, Team_SK*) instead, i.e. if the key portion were defined as:

```
P.defineKey ("position_code", "team_sk") ;
```

In this case, we would not have to harvest the outfielders for *all* teams and filter them on the way out. Instead, we would code:

```
do team_sk = 103, 147 ;
   do rc = p.FIND() by 0 while (rc = 0) ;
      output ;
      rc = p.FIND_NEXT() ;
   end ;
end ;
```

So, FIND would first point to the first item with the key ("OF",103), and FIND_NEXT would harvest the rest of the items for this key pair only. Then the process would be repeated for ("OF",147).

Better Keyed Enumeration in 9.4

However, in Version 9.4, there is a better way. Returning to the case where the table is keyed by *Position_code* only, we can use the new method DO_OVER, specifically designed for the purpose, and modify the loop above in the following manner:

```
do while ( p.DO_OVER(KEY:"OF") = 0 ) ;
   if team_sk in (103, 147) then output ;
end ;
```

The first call to DO_OVER places the pointer at the first item in the group with *Position_code*="OF", and the subsequent calls cause it to step through the group one item at a time, performing the retrieve operation. When the method returns a non-zero value for the WHILE condition, the loop terminates.

Though the general scheme remains the same, DO_OVER makes code simpler and cleaner - and also makes it unnecessary to use variable RC. Also note that using the argument tag KEY is handier than making the method with no argument tag accept the key from the PDV. First, it allows to get rid of the statement assigning "OF" to *Position_Code*. Second, it leaves the pre-loop value of *Position_code* intact, which in many programming situations can be quite valuable by adding a degree of flexibility.

One-to-Many, Many-to-Many Joins

The first use of keyed enumeration that comes to mind is the ability to perform one-to-many and many-to-many file matches when the file loaded into the hash table must contain duplicate-key items. We already mentioned it above while discussing the joins. Again, the subtopic - along with a much more detailed discussion of keyed enumeration in general - is amply illuminated elsewhere [*2*], so please follow the reference. In the upcoming book, there will be a chapter dedicated to the subject.

Quasi-Cartesian On-the-Fly Processing

There exist scenarios where, before performing a data processing task, a table needs to be enriched via bringing a secondary key and data associated with it into play by first matching with another table on the primary key. If the relationship between the two by the primary key is not one-to-one, it may result in a huge interim table, as the number of interim records for each primary key value will be the product of those in the two inputs. Depending on the product, in some real-life situations the result set can be too voluminous to store or handle, though the actual result of the data processing collapse is not sizable.

A typical scenario of this kind, described in detail in [*3*], can arise during a data aggregation process. In such cases, the keyed enumeration operation using a hash table comes to rescue. This is because aggregating by the composite (primary key, secondary key) and accounting for the linked data can be done inside the enumerating loop on the fly without first storing the quasi-Cartesian "explosion" somewhere and then reading it. Thus, the entire aggregation process occurs in memory and eliminates the two extra I/O cycles. It can spell the difference between being able to perform the task reasonably fast and not being able to perform it at all because of the resource overload.

Using Keyed Enumeration for Selective Update and Delete Item Operations

The REPLACE and REMOVE methods (discussed in the sections devoted to the *update* and *delete Item* operations) are designed to act at once on the entire group of items with the same key. For example, suppose that in the example with table P above (keyed by *Position_code*), we code:

```
rc = REMOVE(key:"OF") ;
```

Such action will result in *all* of the outfielder items deleted from the table. In the same vein, suppose that we code:

```
rc = REPLACE(key:"OF", data:"OF", data:0, data:0) ;
```

In this case, the Player_ID and Team_SK values will be zeroed out in every outfielder-related hash item.

Note of caution: The REPLACE method for a table declared with MULTIDATA:"Y" acts this way *only in Version 9.4 and up.* This is more logical than in the earlier versions where, instead of updating all the items in the same-key group with new values, REPLACE adds a single item with the same key and new data values to the group.

In other words, REMOVE and REPLACE delete and update same-key hash items *en masse*. However, more often than not, this is not what we want. Rather, it is more utile to be able to update or delete a *specific item* (or *items*) within the same-key group *based on a condition*. And this is where the enumeration by key operation, coupled with the methods REMOVE*DUP* and REPLACE*DUP* comes in very handy. The core idea is that enumeration enables program control to visit every item in a same-key group separately. While it dwells on the item selected according to some criterion, either it can be deleted or its data portion - updated. Here are a few simple examples based on the same table P as above.

In the group with *Position_code*="OF", delete the items for team *Bears* (team_sk=147):

```
do while ( p.DO_OVER(KEY:"OF") = 0 ) ;
```

```
      if team_sk in (147) then rc = p.REMOVEDUP() ;
   end ;
```

In the infielder and utility player groups, if Player_ID ends in 7, replace it with 77777:

```
   do _q = 1, 2 ;
      do while ( p.DO_OVER(KEY:_scan ("IF UT", _q)) = 0 ) ;
         if mod (player_id, 10) ne 7 then continue ;
         player_id = 77777 ;
         p.REPLACEDUP() ;
      end ;
   end ;
```

Note how above, the tag argument KEY accepts *not a literal* but an *expression*. Hopefully, the idea is clear, and variations are limited only by the nature of the task at hand and imagination. In addition, the keyed enumeration facility is also packed with auxiliary methods, such as HAS_NEXT, HAS_PREV, RESET_DUP, which can be useful in more elaborate programming scenarios.

More Keyed Enumeration Examples

More examples of putting keyed enumeration to good use can be found in the programs that generate sample data for this paper. They are easy to spot by looking for MULTIDATA argument tag and/or for REMOVEDUP and REPLACEDUP key words.

START ITERATION OPERATION

Even though it is a key-based operation, it logically belongs to the discussion of sequential enumeration (the hash iterator), so we will defer it till then.

ORDER OPERATION

Just as an SQL result set can be forced into a specific order by using ORDER BY, a hash table items can be ordered by the table key as well. But before we get to the discussion of its utility, a few words are in order (pun intended) on what the *order* actually means in the hash object context.

Logical versus Physical Order

The *physical order* in which the hash keys and their items are actually stored in a hash table behind-the-scenes is dictated by the data structures and the algorithm underlying the hash object. From the standpoint of using the hash object in a SAS program, the physical order is of no consequence.

What does matter is the order in which the hash keys and items appear in the PDV as they are accessed and retrieved, which is the *logical order*. The only way to see it programmatically is to add the key to the data portion and then either: (a) enumerate the table or (b) write its content to a SAS data file using the OUTPUT method.

Ordering a Hash Table by Key

With the hash object, it does not get any simpler: If the ORDERED argument tag (an option supplied with the DECLARE HASH statement or the _NEW_ HASH operator) is valued as "A", the logical order will be ascending. If the value is "D", it will be descending. (Other values resulting in the same ordering are listed in the documentation.) And if the value is "N" - or if the argument tag is omitted altogether - the order will be undefined. In this case, it will be actually random, and, as we have already discussed above, it can be put to good use.

In this section, we will overview the advantages of having the table forced into ascending order (or, for that matter, descending, which is logically similar). But before doing it, a note of caution (perhaps, somewhat disconcerting): If the key is *composite*, you *cannot have some of its components ordered ascending and some - descending*. A composite key, just like for the purposes of table access, is *treated as a whole*, i.e. as if its components were concatenated.

Utility of Hash Table Order

For its primary operations - keyed data storage and retrieval - the logical order of a hash table is immaterial. However, under many scenarios having the table forced into an order is exceptionally useful. Let us list some applications:

- A programmer using a hash table may need to logically rely on a certain key order. For instance, if the table is ordered, finding the lowest/highest key is a simple matter of grabbing the first/last item. Organizing a stack, as demonstrated earlier, is a good example of utilizing an existing order. Another example in this category: an ordered table makes it very simple to sort an array or a number of parallel arrays.
- An ordered table that has been processed - for example, with a number of updates and/or removals done - can be enumerated or output in order. A sorted output table is invariably easier to eyeball, and the output from an ordered table is already sorted intrinsically. A typical example of such utility is using an ordered table for data aggregation. Although the order is not essential for the aggregation algorithm per se, having the aggregated output automatically sorted is a welcome free bonus.
- Ordered hash tables are an invaluable resource-saving tool for processing of so-called *partially-sorted* data. For instance, picture a large file with insurance claims already sorted by its member ID. Now imagine that for each member by-group, two aggregate metrics must be calculated. However, to obtain one metric, the file needs to be sorted, within its member ID, by claim number, and to obtain the other - by service date. Needless to say, re-sorting a billion-record file twice and then merging the results by the member ID is undesirable, to put it mildly. But, armed with ordered hash tables, one does not need to. This is because in the same pass through each member ID by-group, two hash tables, one ordered by claim number and the other - by service date, can be populated with the necessary variables in the same pass through the by-group. Then each can be processed separately to compute the respective metrics and emptied out before the next by-group (see the discussion of the CLEAR method below). *Voila*.

Needless to say, this list merely scratches the surface. Again, more examples of utilizing this valuable feature can be found in the programs generating the sample data (look for the key word ORDERED).

KEYLESS HASH TABLE OPERATIONS

This is a group of hash table operations which, *from the standpoint of a SAS programmer* using the hash object, do not need a given key to work. They either access the table sequentially or perform an *en masse* action involving all the hash keys and items.

It must be noted that at times, the distinction between key-based and not key-based can be somewhat murky. For example, the REPLACEDUP method, formally speaking, does not need a key. But since it cannot operate without another key-based method called first to locate the group of items for it, it is still logically a key-based operation. Also, it is quite possible that to an underlying software developer, everything in the hash object is key-based. However, it is the SAS user viewpoint that matters here.

CREATE OBJECT AND DELETE OBJECT OPERATIONS

These operations are rather trivial and well described in the documentation. Certain caveats and user-developed alternative techniques to make such actions as, say, parameter type matching or defining multiple key and data portion variables less error-prone and more automated, are amply covered in the SAS literature [*4*]. Here we will only make a couple of notes:

- In all examples given in the paper, the DECLARE statement, rather than the _NEW_ operator is used to create an object. This is because only a single instance of it is needed. Under other scenarios, specifically the need to store other hash object instances in the table (colloquially termed "hash-of-hashes"), the _NEW_ operator is the proper tool.
- All things hash are run-time calls. Hence, if you let program control revisit the group of statements declaring, defining, and instantiating a hash object, it will be *automatically deleted* and then *recreated*. Moreover, if DATASET argument tag is present, the file specified in it will be reread anew to reload

the table. This is why such a group of statements is either encumbered by the condition IF _N_=1 or the STOP statement is used to prevent program control from revisiting the group.

DELETE ITEMS OPERATION

This is a very useful operation because it makes it possible to empty a hash table in one fell swoop without the need of removing its keys and their items one at a time or deleting the table itself. The operation is performed by calling the CLEAR method. The method is typically called for in the situations where a hash table is used to process one by-group at a time and needs to be emptied out before the next by-group starts populating it - which is why it was mentioned above in relation to partially-sorted data processing.

Emptying the table in this manner is two-pronged: (a) it "reinitializes" the table without deleting it and incurring the overhead of recreating it, and (b) it releases the part of RAM occupied by its keys and data, which in many applications (specifically, in data aggregation scenarios) can be extremely critical.

The sample data program *S0600-GenerateSchedule.sas* has a good example of using the CLEAR method to prepare table USED for the next loop iteration. Let us consider another example more in vein with by-processing mentioned above.

File Splitting Example

Historically, it is the first user-written program where a hash table is used to process one by-group at a time. It is also interesting because it fittingly illustrates the dynamic nature of the hash object.

Hence, suppose that we want to split data set *bizarro.teams* pre-sorted by variable *League* into as many separate data sets as there are leagues. We also want to name the output data sets after the pattern *League_#*, where # is the league value itself. (Of course, as there are only 2 leagues in the file, it can be done conventionally by listing *League_1* and *League_2* in the DATA statement and using simple IF logic. Let us pretend, however, that we do not know ahead of time how many *League* values are in the file, nor what those values are.) Moreover, we want the output file for every league to be sorted by the team name - and do all that work in a single pass through the input file. Here is code:

```
data _null_ ;
   if _n_ = 1 then do ;
      dcl hash H (multidata:"Y", ordered: "A") ;
      H.definekey ("team_name") ;
      H.definedata ("team_sk", "team_name") ;
      H.definedone () ;
   end ;
   do until (last.league) ;
      set bizarro.teams ;
      by league ;
      H.add() ;
   end ;
   H.output (dataset: catx ("_", "League", league)) ;
   H.CLEAR() ;
run ;
```

And this is what happens here:

Before the first input record is read, hash table H is created.

- The *DoW-loop*, `do until (last.league)`, reads the first *League* by-group and stores the values from each record the hash table H. (*See the note on the DoW-loop below.*)
- After the group has been processed, the OUTPUT method writes the content of H to a file, whose name is based on the current value of *League*, and closes it. Note that it happens at run time.
- Table H is emptied out by calling the CLEAR method and thus prepared to be loaded from the next by-group. The RAM portion occupied by the table is released.
- The process repeats for the next by-group, and so on.

The structure of this program, replete with using the DoW-loop to segregate the actions within each by-group from those before and after, is quite typical for any arrangement where a hash table is used to process one by-group at a time, such as data aggregation or partially-sorted data processing. For the information on the DoW-loop, please see [*5, 6, 9, 10, 11*].

ENUMERATE SERIALLY OPERATION

Keyed enumeration discussed above is based on pointing to the same-key group of hash items *with a given key* and then traversing this group one item at a time. However, in many practical situations we may want to traverse the *whole table* or part of it and retrieve the data portion values for each item we encounter *regardless of the keys* encountered in the process. In other words, we need to perform the operation of *keyless enumeration*.

This operation is supported by an additional object linked to the hash object and called *the hash iterator*. The iterator is supplied with its own access methods, making it possible to do a number of things:

- Call the FIRST (or NEXT if called first) method to get at the *logically first* item in the table.
- Call the LAST (or PREV if called first) method to get at the *logically last* item in the table.
- Call the NEXT method to get to the item *following* the item on which the iterator currently dwells.
- Call the PREV method to get to the item *preceding* the item on which the iterator currently dwells.
- Call the SETCUR method to get to the item with a specific key value. NEXT and PREV can then be called to step forward or backward from thence.

It is important to realize a few basic facts about the way the hash iterator works:

- *Any* method listed above retrieves the hash data, i.e. overwrites the PDV host variables with the values of their hash counterparts. It is equally important to keep in mind that as any retrieval method, these methods retrieve *only* the data portion values. Therefore, if it is desirable to retrieve the key as well, it must be defined in the data portion in addition to the key portion.
- Every iterator method issues a return code and fails - i.e. returns a non-zero code - only if the item it looks for is not in the table. So, all of them fail if the table is empty. Otherwise, FIRST and LAST are always successful, and NEXT and PREV fail only if they currently dwell on the last and first items, respectfully.
- The item on which the iterator currently dwells cannot be removed from the table. As a corollary, if the iterator dwells on any hash item, the CLEAR method will always fails since it aims to remove all the items, and the REMOVE method will fail unless the iterator is not moved away from the item REMOVE is trying to delete.
- The most efficient method to move the iterator out of the table is to call FIRST followed by PREV or LAST followed by NEXT. That clears the way for the CLEAR method to work.

Iterator Mechanics Example

For a change, let us illustrate the iterator mechanics by loading the hash table H below from an array. Note that DEFINEKEY in the absence of DEFINEDATA automatically adds the key to the data portion.

```
data _null_ ;
   array a [9] (5 4 1 9 6 8 3 7 2) ;
   dcl hash h   (ordered: "A") ;
   h.definekey  ("item") ;
   h.definedone () ;
   retain item . ;
   do i = 1 to dim(a) ;
      h.add(key:a[i], data:a[i]) ;
   end ;
   dcl hiter HI ("H") ;
   rc = hi.first() ; put "FIRST: " item= rc= ;
   rc = hi.next()  ; put "NEXT : " item= rc= ;
```

```
      rc = hi.last()  ; put "LAST : " item= rc= ;
      rc = hi.prev()  ; put "PREV : " item= rc= ;
    * get iterator out of table ;
      rc = hi.first() ; put "FIRST: " item= rc= ;
      rc = hi.prev()  ; put "PREV : " item= rc= ;
      put / "List items forward : " @ ;
      do while (hi.next() = 0) ;
         put item @ ;
      end ;
      put / "List items backward: " @ ;
      do while (hi.prev() = 0) ;
         put item @ ;
      end ;
   run ;
```

The program prints the following in the SAS log:

```
FIRST: item=1 rc=0
NEXT : item=2 rc=0
LAST : item=9 rc=0
PREV : item=8 rc=0
FIRST: item=1 rc=0
PREV : item=1 rc=160038

List items forward : 1 2 3 4 5 6 7 8 9
List items backward: 9 8 7 6 5 4 3 2 1
```

Since the table is ordered ascending and enumerated in its *logical* order, the items are listed in ascending key order. Note that the second call to PREV failed, for at the time the iterator was dwelling on the first item as a result of the preceding call to FIRST. However, the failure was useful because it moved the iterator out of the table and made it possible for the initial NEXT call in the second loop to start at the first item. The second loop did not need any such artifices to work because after the first loop ended, the iterator was moved out of the table automatically by the final call to NEXT, whose very failure terminated the first loop.

Array Sorting

It should be fairly obvious from the example above that an ordered hash table coupled with an iterator can be used to sort an array - in fact, a number of *parallel* arrays - simply and efficiently. All that remains to be done is to iterate through the table and put the items back into the array, now in order. We are leaving this to the readers geeky enough to entertain this curious exercise. If necessary, a full treatment of the subject can be found in [*4*].

Start Iterator Operation

This operation is supported by the SETCUR iterator method. It presents a certain classification dilemma because, while it is a key-based method, it works only in tandem with the hash iterator, which by its nature is keyless - that is why we have relegated the discussion of the operation to this section. At any rate, for a number of data processing tasks involving the hash iterator, SETCUR comes in very handy because it makes it possible to start iterating not from the beginning or end of the table but from the item with a given key value somewhere in the middle.

For instance, suppose that in the table H in the example above, we want to locate the two items abutting item 5 on its either side. Clearly, keyed enumeration is of no use here. But with the aid of SETCUR, it is easy:

```
      rc = hi.setcur (key:5) ; rc = hi.next() ;
      rc = hi.setcur (key:5) ; rc = hi.prev() ;
```

In the same manner, one can, if need be, explore a wider vicinity around any given key item by coupling SETCUR with more than one successive call to NEXT and PREV. An example of solving a "minimax" problem (stemming from a real-world SAS-L question) using such an approach can be found in [*4*].

Keyless Enumeration Applications

The examples given above are good for explaining the mechanics of the keyless enumeration operation and even possess a certain degree of utility - however peculiar it might be. In our experience, however, most hash iterator applications fall in a few categories below.

File Post-Processing

Typically, it means that after a hash table has been populated with some information obtained from reading a file, the data it now contains has to be enriched, or output, or both. To illustrate, let us recall the data aggregation example presented earlier - with a few twists and nifty tricks. Suppose that the stats we need to calculate now are *AB* (at bats) , *H* (hits), *TB* (total bases), *BA* (batting average), and *SLG* (slugging). (The calculation of "count distinct" *Games* is left out for the sake of brevity.) *BA* and *SLG* can be computed only after the rest of the variables needed to do it have been already accumulated. It means that we have to process the aggregated hash table after the last file record has been read - in other words, to *post-process* it. Let us look at the program first:

```
%let stats = AB H TB BA SLG ;
data metrics (keep = Batter_ID &stats) ;
   dcl hash B  (ordered:"a") ;
   B.defineKey ("Batter_ID") ;
   B.defineData("Batter_ID") ;
   do i = 1 to countw ("&stats") ;
      B.defineData(scan("&stats", i)) ;
   end ;
   B.defineDone() ;
   do until (LR) ;
      set bizarro.atbats end = LR ;
      rc = B.FIND() ;
      AB = sum (AB, Is_An_AB) ;
      H  = sum (H,  Is_A_Hit) ;
      TB = sum (TB, Bases) ;
      B.REPLACE() ;
   end ;
   dcl hiter BI ("B") ;
   do while (BI.NEXT() = 0) ;
      BA  = divide (H,  AB);
      SLG = divide (TB, AB);
      OUTPUT ;
   end ;
   stop ;
   format BA SLG 5.3 ;
run ;
```

And now let us look at how it all works:

- As before, the summary stats AB, H, TB are accumulated for each distinct Batter_ID in hash table B during the pass through the input file.
- After it is over (LR=1), we need to walk through every item in table B to compute BA and SLG, so we define iterator BI linked to table B.
- NEXT is first to be called, so it begins at the *logically first* hash item, and then repeated calls to NEXT in the loop cause the iterator to step to the next item one at a time, every time retrieving the aggregated *hash* values of H, TB, and AB into their PDV host counterparts.
- For each item on which the iterator dwells, BA and SLG are calculated from H, TB, and AB.
- The current PDV values are written as a record to output data set *Metrics*. Note that using the iterator makes it unnecessary to call the OUTPUT method, as it visits every hash item anyway.

Another twist added to the program is the repeated calls to the DEFINEDATA method in a loop. They define the data portion variables one at a time from an expression `scan("&stats",i)` given to the

method as a positional argument. Doing so replaces the necessity to list the variable names as hard-coded quoted literals and comes in extremely handy when the list is long. The ability to do it this way can be looked at though the prism of the *dynamic nature* of the hash object in general.

By-Group Post-Processing

This is the case when one or more hash tables are loaded from the data in a by-group and then scanned by the iterator to determine the by-group's collective characteristics. If a table is used to aggregate data, the iterator can be used just to add its aggregated content to the output. In other cases, it can be used to obtain metrics pertaining to each by-group as a whole - for example, to find out if a medical procedure has been performed on at least two distinct dates or more than once on the same date, etc. (a kind of situation typical of calculating HEDIS measures).

As an example of using keyless enumeration for by-group post-processing, suppose that we want to know, based on the data set *work.Metrics* created in the example above, which 3 players in each team have the highest and lowest batting average. Let us assume that data set *bizarro.players* has already been sorted into data set *work.Players* by team, i.e. variable *Team_SK*. To get what we want, we will first link *work.Players* to *work.Metrics* via *Player_ID=Batter_ID* by loading *work.Metrics* into a hash table, and then proceed as follows:

```
data BA_minimax3 (keep = team_sk player_id BA) ;
  if _n_ = 1 then do ;
     dcl hash T (ordered: "A", multidata: "Y") ;
     T.definekey ("BA") ;
     T.definedata ("BA", "Player_ID") ;
     T.definedone() ;
     dcl hiter TI ("T") ;
     if 0 then set metrics (keep = batter_id BA) ;
     dcl hash M (dataset:"Metrics") ;
     M.definekey ("Batter_ID") ;
     M.definedata ("BA") ;
     M.definedone () ;
  end ;
  do until (last.team_sk) ;
     set players ;
     by team_sk ;
     if M.find (key:Player_ID) = 0 then T.add() ;
  end ;
  do q = 1 by 1 while (TI.next() = 0) ;
     if q <= 3 or q + 3 > T.num_items then output ;
  end ;
  T.clear() ;
run ;
```

The logic here follows the typical by-group post-processing pattern:

- The DoW-loop processes one team at a time. For each record, table M is searched. *BA* and *Player_ID* are added to table T if *Player_ID=Batter_ID* match is found.
- When the DoW-loop is done with the current team (*last.team_sk=1*), table T contains all BA and *Player_ID* items for the team. Note that table T is *both keyed* and *ordered* by *BA*.
- Because *T is ordered by BA*, all we need to do is iterate through T and get the 3 first and 3 last items, which is what the iterator loop does.
- Hash table T is emptied out, which prepares it for the next team by-group, and program control is passed back to the DoW-loop to process it.

Hash Tables Cross-Matching

In some situations, it is necessary to cross-match one or more hash tables. In this case, one table is scanned by the iterator one item at a time, matching the other tables via key-based operations. Note that most of the time, such need also arises in the course of file or by-group post-processing.

Using Keyless Enumeration as an Aid to Keyed Enumeration

Sometimes, when duplicate-key items need to be harvested from a hash table A for a number of distinct key values, the logic of the program dictates that they be stored somewhere first. The most natural container for such list is another hash table B since it does not require (unlike arrays, say) to know the number of these distinct key values ahead of time. Thus, table B can be used as a key reference for traversing table A one key group at a time. Table B is scanned using an iterator, and for each key extracted from it, table B is traversed by using, for example, the DO_OVER method. A good example of this kind of application is presented in sample program *S0600-GenerateSchedule.sas*.

OUTPUT OPERATION

The hash object is supplied with its own I/O facility, enabling it to read and write SAS data files at run time. This is done by giving a file name (in general, an expression resolving to a valid file name) to the argument tag DATASET. Input is done using the DECLARE statement or _NEW_ operator, and output - by calling the OUTPUT method. Since we have already seen many examples of using them in this paper, there is no need to dedicate separate examples to it here. However, a few important points need to be stressed:

- The hash object I/O operations are strictly run-time. For example, when the OUTPUT method is called, program control dwells on it and is not passed to the next executable statement before the output operation is finished and the output file is closed.
- Only the data portion hash values are written out. So, if a key has to be added to the output, it must be defined as a data portion variable as well. In this sense, OUTPUT acts exactly like any other *retrieval* operation.

OTHER TOPICS

HANDLING NON-SCALAR DATA TYPES

The data portion of a hash table is normally composed of numeric and character variables, collectively termed *scalar* data types. It is well-known, and most of the time, they are the only ones that are needed. However, it is less known that the data portion can also house variables of the type object. In particular, different instances of another hash object can be stored there - and handled.

Such a capability has engendered a number of interesting uses, the first of which is the possibility to split a file dynamically similarly to the *Example of Splitting a File* subsection above but without the need to have the input file sorted. Other curious applications can be found in [*4, 6, 7*].

Here we will only illustrate the basic concept by a simple example:

```
data _null_ ;
  dcl hash X () ;
  x.definekey ("n") ;
  x.definedata ("h", "n") ;
  x.definedone () ;
  dcl hash H ;
  do n = 1 to 3 ;
     h = _new_ hash () ;
     h.definekey ("_n_") ;
     h.definedone() ;
     do _n_ = 1 to ceil (ranuni(1) * 10) ;
        h.add() ;
     end ;
     x.add() ;
  end ;
  dcl hiter XI ("x") ;
  do while (xi.next() = 0) ;
     h.output (dataset: catx ("_", "hash", n)) ;
  end ;
```

```
run ;
```

Above, table X, keyed by variable N, is used to store 3 i*nstances* of the hash object H. A new instance is created every time when, for the pre-declared object H, the _NEW_ operator is invoked, and for each, its own hash table, keyed by _N_, is defined - and populated. Then iterator XI linked to table X is used to scroll through X and, for each of the 3 items it visits, to output its own hash table to a separate SAS data set. Note that the data portion of X contains two variables: One, N, is scalar, while the other, H, is of the type hash object. (*Please run this program and observe the results in the SAS log and output.*)

MEMORY MANAGEMENT

The SAS hash object exists and operates totally in memory. Herein lies its strength because it makes it fast and dynamic. However, herein also lies its weakness, as the size of physical computer memory is a limited thing despite constantly getting bigger and cheaper. Therefore, the size of a hash table ought to be treated with respect. It depends on two things: The *hash entry length* and the *number of items*.

The hash entry length is not merely the sum of the lengths of the key and data portion variables. It also depends on the operating system, memory addressing (32-bit or 64-bit), and the data types of the variables. The relationships of this kind are rather complex. However, as a rule of thumb, it is still safe to assume that the more variables the table has and the longer they are, the longer the hash entry will be. Therefore, it is bad practice to overload the hash entry with variables that are not really needed. It may work fine while the data are relatively small, but bloated hash entries scale poorly and can bite really hard when the table size exceeds a certain threshold. (As a curiosity, the shortest hash entry under any circumstances is one with a single numeric key variable and a single numeric data variable.)

There exists a number of techniques to reduce the *hash entry length*, among them:

- Avoid defining key variables in the data portion if the program does not need to retrieve them.
- To the point above, if all you want to do is pure search (i.e. call the CHECK method) for a long composite key, *do not call DEFINEKEY stand-alone* - i.e. not followed by DEFINEDATA. Not only it *would not make the data portion not exist* (it always exists!) but instead cause *all key portion variables to be auto-added to the data portion* and may overload memory without reason. Rather, call DEFINEDATA with a single numeric dummy variable.
- Using the MD5 function signature of a long composite key instead of the key itself. The beauty of this method lies in the fact that it reduces the key length of any composite key, no matter how long, to mere 16 bytes. The MD5 method works especially well under all file-matching and data aggregation scenarios. An example of such subterfuge can be found in sample program *S0600-GenerateSchedule*. Also, it is given a rather thorough treatment in [*3*].
- Offloading the data portion to disk. It means that instead of storing all the data variables in the entry, they are left on disk - typically, in a SAS data set - and only their RID (record identifier) is stored in the hash table alongside the key. Then when these variables are needed in the PDV, they can be accessed via the SET statement with the POINT=RID option. A comprehensive discussion of the method is given in [*13*].

The *number of hash items* is a bit tougher thing to address - after all, you put them all in the hash table with reason, right? However, the fact is that in some situations it can be reduced rather dramatically, for example:

- If a hash table is loaded from a file sorted by the leading component of the composite key you intend to store in the table, give it a pause. Most likely, you can leave it out and do by-processing by this key instead. This way, the table needs to house only enough data to accommodate the largest by-group, as before each next by-group it is emptied out. The MiniMax_B3 example above gives a good idea how this can be done.
- Input for data aggregation or file matching can be split by using the MD5 function and processed in chunks within which no keys overlap. The method is rather involved to be treated here, but it is described in gory details in [*3*].

In the upcoming book, a separate chapter will be dedicated to memory management.

SAMPLE DATA AND PROGRAMS

The sample data and code snippets used in this paper are from a set of programs that will be used to create sample data for a SAS Press book that will address the broad functionality that can be supported by the SAS hash object. The programs included here generate the sample data needed to illustrate a broad range of common functions needed in managing and reporting on data. An incomplete list includes:

- Table Lookup Techniques – the most well know and the problem most commonly addressed using the SAS hash object.
- Data loading and management techniques – accessing transactional data to create, for example, star schema fact and dimension tables.
- Basic Data Aggregation Techniques – either from a structured data warehouse or directly from the transactional data.
- Advanced Data Aggregation Techniques – including those multiple levels of aggregation as well as non-hierarchical aggregation.
- Memory Management Techniques – approaches to minimize the memory management required while still fully leveraging the SAS hash object.

The data to be generated is for a fictitious game called *Bizarro Ball*. Bizarro Ball (aka *BB*) is conceptually similar to Baseball, with a few wrinkles. Our generated sample data will include data for teams, players, and games. The game data will include a row for each pitch, at bat and runner. This provides us with a rich set of data and also provides the opportunity to highlight both well known, and less known/used, capabilities of the SAS hash object.

The following programs will be available from the sasCommunity.org page mentioned in the *Introduction* above and included below:

- *autoexec.sas* – an autoexec that sets up the environment.
- *GenerateDataDriver.sas* – a simple driver program that calls the following programs in order to generate the data.
- *S0100-GenerateTeams.sas* – reads in a list of available team names and randomly selects 16 team names for two leagues by leveraging the inherent features of the SAS hash object.
- *S0200-GeneratePositionsDimensionTable.sas* – a simple SAS program that simply loads data on the distribution of how many players to assign to each team (e.g., how many infielders, outfielders, starting pitchers, etc.).
- *S0300-GeneratePlayerCandidates.sas* – reads in a list of the most common 100 first names and 100 last names and then uses the SAS hash object to create the 10,000 rows corresponding to the Cartesian product.
- *S0400-AssignPlayersToTeams.sas* – uses the SAS hash object to select 50 names for each team as well as assign a their role (e.g., what position they play) on the team.
- *S0500-GenerateMatchUpCombinations.sas*– uses the SAS has object to create a filtered/subset Cartesian product of all the possible matchups within a given league (i.e., each team playing each other team up to a fixed number of times)
- *S0600-GenerateSchedule.sas* – subset the combinations so that each team plays twice each data against a randomly selected opponent for each game in a given day.
- *generatelineups.sas* – a macro that for each team and for each day, assigns a fixed number of batters and pitchers to each game played that day so that a given player only plays in one game.
- *S0800-GeneratePitchDistribution.sas* – a simple SAS program that creates the distribution for the results of each pitch (e.g., what percent are balls, called strikes, swinging strikes, singles, etc.)

- *generatepitchandpadata.sas* – a macro that generates the At Bat, Pitch and Runner data that is use in many of the examples.

Running these programs produces a reasonably rich set of data:

- 32 teams (2 leagues, each with 16 teams)
- 1,600 Players (2 leagues * 16 teams * 50 players)
- 992 games over 31 days (16 morning games and 16 afternoon games for each of 31 days)
- 27,776 batter x game combinations (each team has 14 batters in each of the 62 games)
- 17,856 pitcher x game combinations (each team has 9 pitchers, who each pitch one inning, in each of the 62 games)
- 98,769 plate appearances across all the teams/games. This number can be varied based on the s random number seeds used to determine the results of each pitch. Note that this data includes details on what bases a given runner occupies at the completion of each plate appearance.
- 301,506 pitches for those 98,769 plate appeances.
- 23,901 runs scored

The summary statistics calculation is a subset of the data in *Sample Metrics AllWays* or the *Sample Metrics* program. A zip file with the programs described above can be found here:

http://www.sascommunity.org/wiki/Beyond_Table_Look-up:_The_Versatile_SAS_Hash_Object

CONCLUSION

The SAS hash object, through the efforts of SAS R&D and user contributions, has developed into a flexible and versatile programming tool. The variety of its existing and potential applications is hard to grasp merely by reading the SAS documentation, for though it is systematic, it views the hash object through the prism of the tools it is packaged with. On the other hand, studying user-written examples is useful from the "how to do this" angle but leaves the learner, especially a novice, without a system and invites thoughtless code cannibalization.

Yet in order to use the hash object conscientiously and creatively, one needs to digest its basic principles and the way it communicates with its programming environment. In this paper, an attempt has been made to systemize the hash object's practical uses against the framework of the hash table operations. To which extent this approach has succeeded or not is for the reader to decide.

REFERENCES

1. Henderson, Donald J., Merry G. Rabb, Jeffrey A. Polzin. 1991. The SAS System Supervisor - A Version 6 Update. *Proceedings of the 1991 SUGI Conference*. New Orleans, LA.
2. Dorfman, Paul M. 2016. Using the SAS® Hash Object with Duplicate Key Entries. *Proceedings of the SAS Global 2010 Conference*. Las Vegas, NV.
3. Dorfman, Paul M., Henderson, Don. 2015. Data Aggregation Using the SAS® Hash Object. *Proceedings of the SAS Global 2015 Conference*. Dallas, TX.
4. Dorfman, Paul M. The SAS® Hash Object in Action. 2014. *Proceedings of the WUSS 2014 Conference*. San Jose, CA.
5. Dorfman, Paul M., Vyverman, Koen. 2009. The DOW-Loop Unrolled. *Proceedings of the SAS Global 2009 Conference*. National Harbor, MD.
6. Loren, Judy, DeVenezia, Richard A. 2011. Building Provider Panels: An Application for the Hash of Hashes. *Proceedings of the SAS Global 2011 Conference*. Las Vegas, NV.

7. Hinson, Joseph. 2013. The Hash-of-Hashes as a "Russian Doll" Structure: An Example with XML Creation. *Proceedings of the SAS Global 2011 Conference*. San Francisco, CA.
8. Bian, Haikou, Maddox, David. 2014. More Hash: Some Unusual Uses of the SAS® Hash Object. *Proceedings of the SESUG 2014 Conference*. Myrtle Beach, SC.
9. Li, Arthur. 2014. Understanding and Applying the Logic of the DOW-Loop. *Proceedings of the PharmaSUG 2014 Conference.* San Diego, CA.
10. Allen, Richard Read. 2010. Practical Uses of the DOW Loop. *Proceedings of the WUSS 2010 Conference*. San Diego, CA.
11. Foty, Fuad J. 2012. The DOW-loop: a Smarter Approach to your Existing Code. *Proceedings of the SAS Global 2012 Conference.* Orlando, FL.
12. Hoyle, Larry. 2009. Implementing Stack and Queue Data Structures with SAS® Hash Objects. *Proceedings of the SAS Global 2009 Conference*. National Harbor, MD.
13. Dorfman, Paul M., Shajenko, Lessia S. 2011. Hash+Point=Key. *Proceedings of the NESUG 2011 Conference*. Portland, MN.

ACKNOWLEDGMENTS

Thanks to Pete Lund for the invitation to present this paper. The authors would also like to recognize the developers in SAS R&D who have made the SAS hash object happen in the first place and continue to improve it and enrich it with features. Also many thanks to all SAS users who, through their creativity, have invented new and, at times, unintended, uses for the SAS hash object. Paul Dorfman would like to thank his co-author Don Henderson for having given him many opportunities to use the hash object in real-life commercial environments, sometimes beyond the intended limits.

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) 260-6509
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