SCSUG 2019

An Animated Guide©: SAS® Hashing(and other fast techniques for big data)

Russell Lavery, Contractor for Numeric Resources

ABSTRACT

Hashing is one of the fastest "table lookup" techniques, not just in SAS®, but in any programming language. Figure 1 illustrates the concept of a "table lookup" and the speed advantage of SAS V9 hashing over a SAS format table lookup (a 95% reduction in run time). If a programmer needs to select, from a large file, all subjects that are in a small file, hashing will likely save both disk space and time. In addition, Hashing capabilities have been expanded so that has tables are now programming tools. Hashing should be part of the tool kit of every programmer who deals with large files.

This paper will support some additional topics, not just table lookup, that might add to a programmer's list of tools that can be applied to big files. Thanks to Paul Dorfman, Don Henderson and Richard DeVenezia for their work on this subject.

Hash Tables can be thought of as a "Big Data Technique". Since "Big Data" is such a common term these days, other SAS based big data (read that as Fast techniques) will be reviewed.

Two things are worth an early notice. First: Hashing has been a part of PROC SQL joins for decades. The SQL optimizer examines the files being joined and decides if a hash will be the fastest join technique. If it is, PROC SQL codes a hash and does not tell you. This is not a SQL paper, so please see the appendix for more information. Secondly; the very early SAS hash users coded their own hashes using a SAS array. Because an array can only hold one piece of information, early hash use focused on table lookup and not merging-via-hashing. However, SAS hashing now has the ability to bring many variables "through the hash join". Hashing has become a competitor for "by-merging" and a more general programming technique

INTRODUCTION

Hashing was designed to allow a programmer to subset a large file, based on information in a small file (as is shown in Figure 1). While that is the designed reason for SAS adding this new feature, hashing can also replace by-merges, format lookups, IORC merges, PROC Summary and sorting. A major benefit of hashing is that it does not need sorted input files. Sorting is a disk and CPU intensive process.

Hashing has been the subject of much mathematical and programming research. Hashing gets its speed by: 1) a being a memory resident technique, 2) having a conceptually efficient methodology and 3) being very efficient programmatically. Being a memory resident technique avoids slow disk access. The logic of the hashing algorithm gives it advantages over other lookup techniques.

SAS hashing is implemented via C language Objects that are accessed through the regular SAS data step. I suggest an interested reader google "Dictionary of Hash and Hash Iterator Object Language Elements" of go to the link below.
https://support.sas.com/documentation/cdl/en/lecompobjref/69740/HTML/default/viewer.htm#p0ae2of2fa94xmn1bajgqxczla8u.htm

"Code your own" hashing was introduced to SAS by Dr. Paul Dorfman. His manually-coded techniques run in any version of SAS and are very fast. However, coding of his techniques, despite his excellent examples in SUG proceedings, has been considered difficult. Hashing, as implemented by SAS, is considered easier to use than Dr. Dorfman's original techniques, though Dr. Dorfman's original techniques should be considered when speed is critical.

Since V9.1 was released, Dr. Dorfman, Don Henderson, Richard DeVenezia and a few others, have applied the SAS hashing tool in new and creative ways. They are still leading the application of hashing in SAS. One can not do better than to study their articles. A major deliverable of this is a detailed review of several of their published examples od hashing as a teble lookup and also as a general programming technique.

Table Lookup

Table lookup is defined as the use of a key variable, or id, in one file to lookup values in another file. Often the goal is taking a subset of the large file based on values of the key variable in a small file. Use of hashing for "table lookup" (Figure 1) is a valuable programming technique because "Hashing table lookups" generally run much faster, and smaller, than other methods (like by merges). The hashing method is likely to run both quickly and with very little demand for RAM and disk space.

Best Practices for having programs run fast

As context for hashing, it might be worthwhile to review some programming practices that make your code run fast. Firstly, most often you are the expensive resource. If it takes you five extra minutes to think how to code some cool hash trick that runs 30 seconds faster, you have lost.

Everyone will recognize the code below as inefficient.

```
Data Class_age;
Set sashelp.class;
Age_mo=age*12; Run;

Data Class_age_wt;
Set Class_age;
Wt_kg=weight/2.2; Run;
```

A best practice is to minimize reading/writing the data from/to disk. While SAS has blazingly fast disk I/O, you should still avoid disk I/O if you can. Accessing data on disk is said to be 20,000 times slower than accessing information in RAM.

Avoid sorting and the sorted-by merge. SAS sorting is fast (SAS has a patent on sorting) but sorting used lots of disk and CPU resources. Select functions, and algorithms, that SAS, and the underlying hardware, can do quickly (e.g. substring and like take a lot of computing cycles … though they can be hard to avoid).

MEMORY CONSIDERATIONS

Hashing is a memory resident technique and runs only when the whole hash "object" (as it is called) and several other required methods (think of methods as subroutines used to manage the object) can be held in RAM. Hash objects, and methods, like SAS arrays are specific to one data step. The memory used by a hash object, and its methods, is automatically released when the data step finishes running. Memory allocated to the hash object can also be released manually (the hash object can be deleted) when the object is no longer needed. Manually deleting objects from memory is considered good programming practice.

Hashing is done by using a "thing" not seen before in SAS, an "object". The "'subroutine" that does the hashing is written in C and is called into existence as a SAS data step executes. There is a SAS "interface" that allows SAS programmers to call applets as part of data step execution. New applets can be developed to take advantage of this interface.

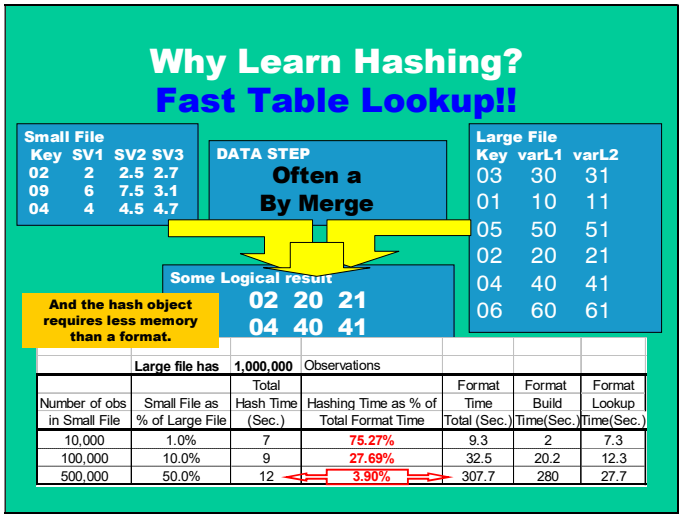


FIGURE 1

CHECK YOUR RAM ENVIRONMENT AND LOAD FILES INTO RAM

Hash tables must fit in RAM and so it is a good idea to check how much RAM you have allocated to SAS. ***By default SAS only requests 2 gig of RAM*** (you might not be using all of that RAM that you paid to have added to your laptop). Try the code below to see how much RAM you have available. If you want to allocate more RAM to SAS, and to hash tables, you can do that on startup using a command in the config file.

```
DATA _NULL_; /*PRINTS RAM AVAILABLE TO SAS*/

MEM = INPUT(GETOPTION('XMRLMEM'), 16.);
MEM_IN_K=MEM/1024;     MEM_IN_MEG=MEM /(1024*1024);    MEM_IN_GIG=MEM /(1024*1024*1024);

PUT @3 "MEM        =" @18 MEM         COMMA16.1; PUT @3 "MEM_IN_K  =" @18 MEM_IN_K   COMMA16.1;
PUT @3 "MEM_IN_MEG=" @18 MEM_IN_MEG COMMA16.1; PUT @3 "MEM_IN_GIG=" @18 MEM_IN_GIG COMMA16.1
; RUN;
```

In line with the principles above, though not a hash technique, is loading a repeatedly used data set into ram. as is shown in the code to the right. Large reductions in run time are possible.

Test1 and test2 will be written to disk as they are created but the disk activity for the reading of data, and any disk contention between simultaneous reads and writes, will be reduced.

I once created monthly reports, off the same source, for 40 nursing stations. I ran a macro 40 times against data stored on disk and wish I had known about this technique at that time.

```
LIBNAME MYDATA 'C:\CENSUS_DATA';
SASFILE MYDATA.CENSUS2000.DATA OPEN;
DATA TEST1;
   SET MYDATA.CENSUS; /*LOAD DATA INTO RAM*/
RUN;
DATA TEST2;
   SET MYDATA.CENSUS; /*USE DATA IN RAM*/
RUN;
PROC SUMMARY DATA=
   MYDATA.CENSUS PRINT;  /*USE DATA IN RAM*/
RUN;
….. MORE PROCS
SASFILE MYDATA.CENSUS CLOSE; /*FREE UP RAM*/
```

FORMAT TABLE LOOKUP: OLDER (AND SLOWER) COMEITORS OF THE HASH TABLE

Before hash tables were brought into SAS, format lookups were a fast, and common, way to do table lookup.

Table lookup is best described with an example.

Imagine your boss comes in with a list of patient IDs and says something like "these people complained about the treatment of our rehab unit. Go to the big, master table and lookup their information from the master record table."

You are looking up, in some big file, the records that are in some smaller file. That is an example of table lookup

The trick that Figure 2 illustrates is how to have SAS, programmatically, create a format statement for you. For more details google "An Animated Guide: Power Merges: The format table lookup"

The advantage of this technique is that neither of the files have to be sorted. Figure 2 shows a numeric table lookup.

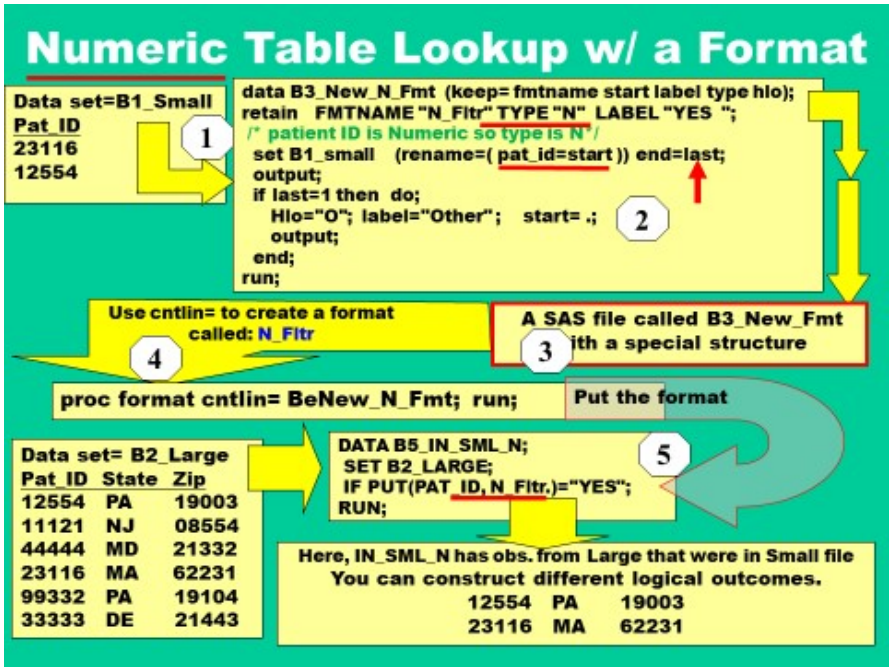


Figure 2

The techniques shown in Figures 2, 3 and 4 do not require the data be sorted as does a by merge. Avoiding sorting is very good however a by merge will let us bring many variables from one data set and many variables from another data set into the final output. The techniques in Figures 2, 3 and 4 do not let us bring as much information, from the small file, into the final output.

Figure 3 shows a character table lookup using a SAS format.

One thing to note is that a format can bring "one piece of information" through the merge.

In this case we assigned a label to be "yes" and use that in the "if statement" in step 5.

However, if the small data set had contained a column char_zip and another column (say customers in that zip) we would be able to load the 2nd column into the label of the format.

If we did that, and changed the if statement in step 5 we would be able to bring one piece of information, from the small file, through the merge and into the final data set..

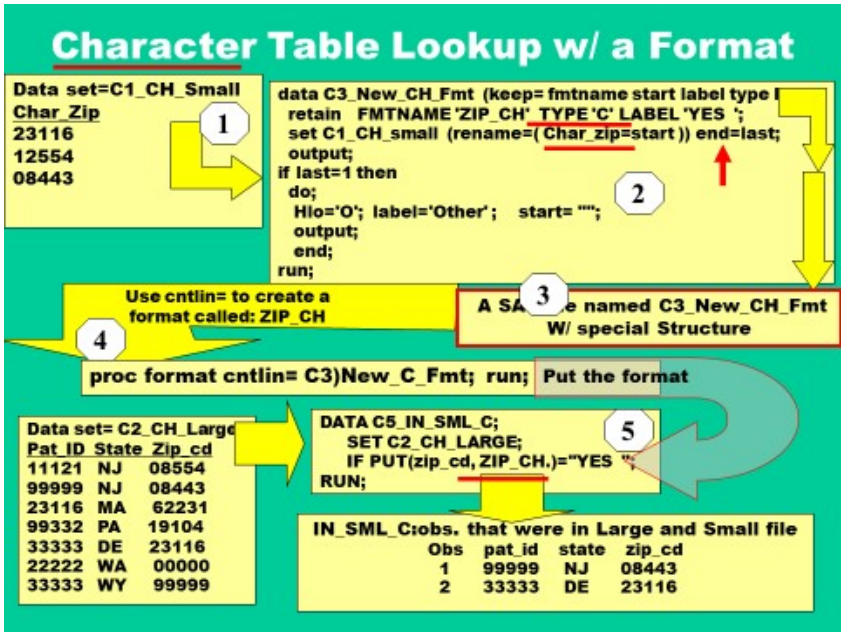


Figure 3

KEY INDEXING AS TABLE LOOKUP THE FASTEST TABLE LOOKUP[

Key indexing, the technique you can see in Figure 4, is the fastest technique for table lookup.

The business situation is the same as above; our boss has given us a small file and asked us to get matching records from the master file.

In this example we are merging on subject ID and subject ID must be numeric.

We are going to use the values of subject ID as the cell number in the array and so the array must be larger than the largest subject ID.

In the next figure you can see the small file having been loaded into the array.

We can load one column from the small file into the array, so we can bring "one variable through the lookup".

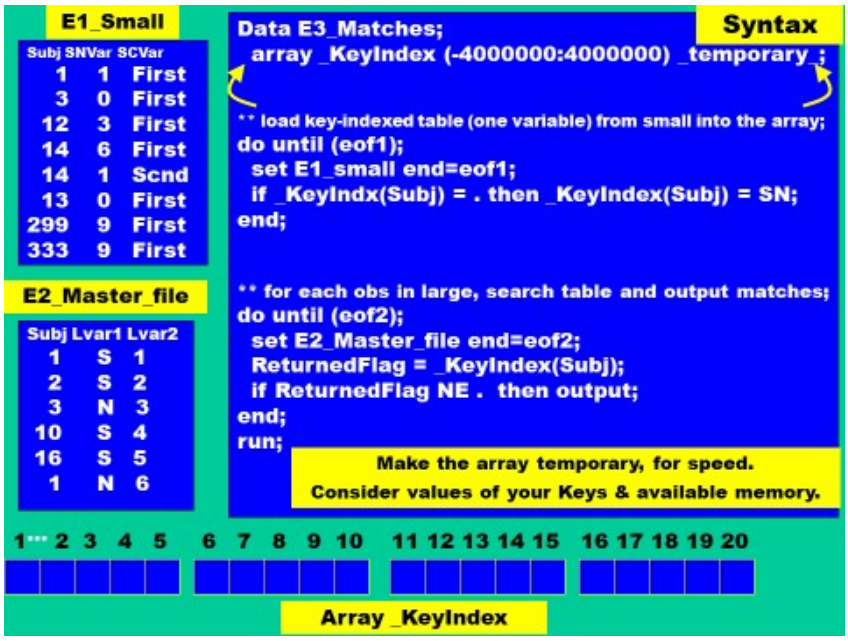


Figure 4

Here you can see the small file has been loaded into the array. Subject number 1 is loaded into cell number 1 in the array. Subject number 3 is loaded into the 3rd cell of the array.

We are using the subject ID number as *the place to put the data* for that subject. This technique is very, very fast because SAS has to do very little, behind the scenes, to make this technique work.

There is a problem if you have, when loading into the array, repeats of the subject ID. In this example 14 is repeated and the programmer must decide how to handle the situation.

In this case this programmer decided to keep the 1st encountered value. In another situation the programmer might want to keep the last encountered value – or the average of the encountered values.

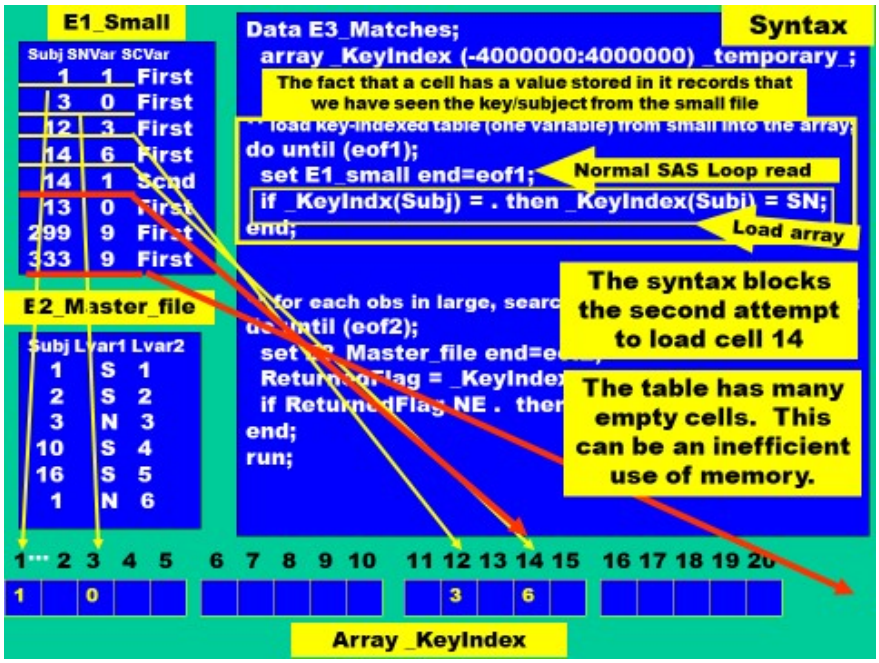


Figure 5

In Figure 6 you can see the logic for seeing whether a subj value, in the master file, was in the small file..

As was shown above, the subject ID number identifies the cell position of interest. If that cell in the array is empty, that subject ID was ***not*** in the small file. If the cell has a value in it, that subject ID was in the small file.

This is faster than hash tables but has three problems.

Firstly; the array must be allocated before the data in small can be read.

Secondly; we must be "merging" on numeric values. The merge variable is going to be the cell number and sometimes we want to merge on characters.

Finally; this is very wasteful of RAM. You can see that the array needs to be large enough to hold the biggest subject ID but most of the cells will be blank.

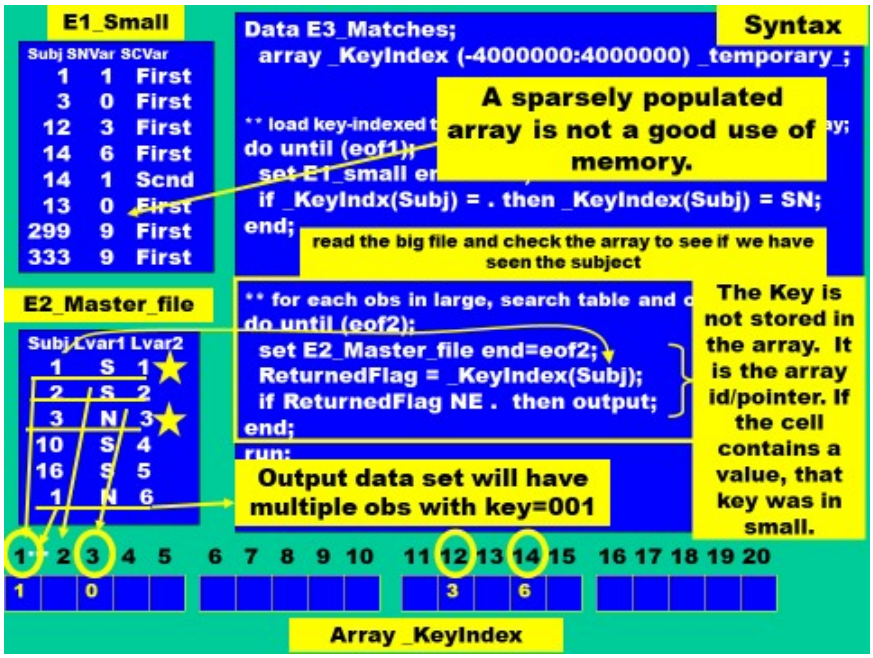


Figure 6

THE GRAPHICAL REPRESENTATION OF THE SAS SYSTEM

Figure 7 shows a graphical paradigm for thinking about the relationship of hashing vs. regular SAS subroutines and memory objects.

Figure 6 is a graphical representation of the SAS system and it shows the components of the SAS supervisor and some of the many different kinds of memory resident objects that SAS programmers must now manage.

The input stack is where SAS code is held after submission. It is a holding area for code, a pre-processing area. After code is submitted, the compiler takes control of the process and starts requesting tokens. Tokens move through this path: input stack-word scanner/token router (where they are assembled and routed) -word queue - compiler. The compiler requests tokens until it encounters a step boundary (run, PROC, quit etc).

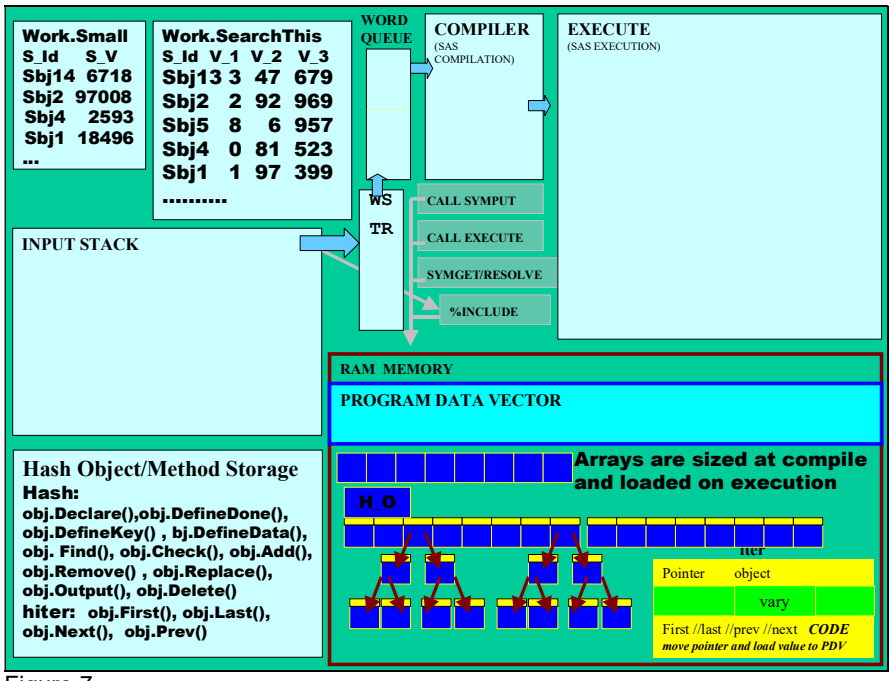


Figure 7

The Word Scanner/Token Router (WSTR) takes characters off the top of the input stack and groups the characters into tokens. Completed tokens are put on the word queue(a six token holding area). As the word queue fills up, tokens flow into, and are stored in, the compiler. When the compiler encounters a step boundary, it takes total control of the process and compiles the section of code it has in storage.

At this time, during SAS compilation, the compiler checks code syntax and *creates the program data vector* (PDV). Statements like set, and infile, trigger the compiler to create a PDV. Arrays are also created on SAS compilation. Importantly, hash objects are **not** created as code gets SAS compiled. On SAS compilation, the code required to create and manage the hash objects is recalled from storage and inserted into your compiled code. Hash objects are created later, when the included code executes.

When SAS code executes, it finally has a chance to interact with the data. On SAs execution, any hash objects, specified in the code, are created and come into existence in RAM. The box in the lower right of Figure 6 represents RAM memory and shows several of the "things" that SAS typically loads into RAM. The PDV is in RAM, as are arrays. Formats, when they are used, are loaded into RAM. Hash objects (the data storage object) and associated methods (associated subroutines to manage the hash object(s)) all must load into RAM.

THE HASH OBJECT

A hash object is shown in Figure 6 and consists of a top row and descending roots. It can be thought of as an array that grows roots for additional data storage. A hash object is assigned an initial size for the top row (the number of boxes in the top row of the hash object is set by the syntax of the submitted code) by the SAS syntax but the hash object *dynamically* "grows" root like appendages as data is fed into it. A hash object can hold an unlimited amount of data. The more data that is fed into the hash object, the longer/deeper the roots become.

When two keys "hash to the same top-level bucket" the hash object starts growing the root like structures shown below the top level (see Figure 6). Unlike two, or three, dimensional arrays these tree structures are very efficient users of memory. Hash objects "take" memory as it is needed.

The program that creates the hash object "re-balances" the object as data is loaded into the hash object. Balancing an object is maintaining a certain relationship among all the keys in a root and insures that searching down a particular root is fast. We will discuss this in more detail later.

The hash object, while memory resident like an array, is created in a unique process. The compiler uses the C-like SAS code, that the programmer types in the middle of the data step, to call pre-compiled subroutines from a "hash object/method storage library". This pre-compiled code, when it **executes**, creates the hash object and methods.

ADDRESSING INFORMATION IN THE HASH OBJECT

Accessing information in a hash object is different from accessing information in more familiar SAS storage areas (Arrays and the PDV). Previously, all information was manipulated in the PDV. Most operations (add, squaring, upcase etc.) will still be performed on the PDV. Using SAS hashing requires that programmers learn to move data between the hash object and the PDV. Providing a graphical presentation of moving data between the PDV and hash table will be a major focus of this paper.

A short review might be helpful. To use arrays, a programmer specifies a cell number that contains the desired information. To get information stored in the fourth cell of the array named sales, SAS code might say: x=sales(4). This would move a value from cell four in the array to the space on the PDV that is reserved for the variable x. Y=z; tells SAS to copy information from one part of the PDV to another (from z to y). In the past, the programmer has told SAS exactly what "array cell number", or variable, to access. Previous to the hash object, all data access was on the PDV.

Inside the hash object, "Buckets of information" can not be directly addressed via conventional SAS commands or by "bucket number". The structure of the hash object is very complex and quite dynamic. To manage/access information in a hash object, a programmer must employ things called methods (AKA helper subroutines).

To use a hash object the programmer "passes" a key value" (maybe a character value or subject_id or customer_no) to a method (a helper sub-routine) and the method does the work of figuring where in the hash object the information is stored (or should be stored). The method applies a mathematical formula (a hashing formula) to the value of the key and produces a number. That number is the location of a top level bucket on the hash object. Running the mathematical formula to calculate the "top bucket number" is called hashing.

Inventing/discovering a good hashing function is a complex process requiring a familiarity with Prime Number Theory. Poor hashing functions caused poor performance and creating a good hash function was an issue discussed in "code your own" hashing in the Dorfman articles. In SAS, a "good" hashing function/formula is automatically provided/selected by SAS and this difficult task is no longer a concern of the programmer. The automation of the hashing function is a major convenience when compared to "code your own" hashing algorithms. People who programmed hashing algorithms, in V8 SAS, had to assume responsibility for the quality of the hashing function they used.

The inputs (Keys) to the SAS automatic hashing function can be very "free form". Like keys in a SAS index, they can be numeric or character. Very importantly, they can be simple (a one variable key) or compound (a many variable key). The keys can even be a mixture of character and numeric variables. SAS hashing automatically handles all combinations of key variables. This is a major feature of SAS hashing.

AN OVERVIEW OF METHODS (OR HELPER SUBROUTINES)

Objects and methods are new words to many SAS programmers and deserve some attention. Objects are programming "things" that can *hold data and act on it*. The data is acted on by using methods (think of methods as sub-routines). I suggest an interested reader google "Dictionary of Hash and Hash Iterator Object Language Elements" of go to the link below. https://support.sas.com/documentation/cdl/en/lecompobjref/69740/HTML/default/viewer.htm#p0ae2of2fa94xmn1bajgqxczla8u.htm

It might be helpful to think of a hash object as being a smart array – an array that grows and re-balances/adjusts itself as you put data in it. Imagine how much programming effort it would take to keep track of the location of data in an array that grows and shrinks as data is added and removed. Fortunately, this "growing" array comes to the SAS programming community with a selection of pre-written "data management" sub-routines called methods.

The new methods/commands have a two-part syntax that is different from typical SAS syntax. The syntax is called "dot syntax" because of the dot that separates the two parts of the commands. The part of the command that is to the left of the dot is the name of the object to which the command is addressed. The part of the command to the right of the dot specifies the action to be taken. The two-part syntax is useful because several hashing objects can be created/used in one data step. The two part (dot) command lets programmers specify the combination of the desired object and the desired action. The two part command is a common communication technique where multiple objects/beings can "hear" the command. This syntax is not too different from the two part syntax your kids use. Examples of this are: "Mom (implied dot), can you make me a snack" or "Dad (implied dot), can I borrow the car keys". Examples of methods are:

Methods associated with the object hash:

Declare HASH name() Create an object- start looking for parameters that tell how the object will be structured.
Obj.DefineData() this list of variables, on the PDV, are data.
Obj.DefineKey() this list variables, on the PDV, are keys.
Obj.DefineDone() stop looking for parameters. You have all the information you need to create the object.

Other methods:

Obj.*ADD()* Add an observation to the hash object.
Obj.*CHECK()* Go to the hash object see if we have anything in the top bucked that this key hashes to- or in a "lower" bucket.
Obj.*CLEAR()* Removes all items from the hash object without deleting the hash object instance.
Obj.*DELETE*() Delete the hash object and free up memory.
Obj.*DO_OVER()* Traverses a list of duplicate keys in the hash object.
Obj.*EQUALS()* Determines whether two hash objects are equal.
Obj.*FIND*() The programmer provides a key and the method gets the data from the hash object that is located in the bucket that the key points to –or in the tree under that bucket. If there is no match for the key, the return code is not zero.
Obj.*FIND_NEXT()* Sets the current list item to the next item in the current key's multiple item list and sets the data for the corresponding data variables.
Obj.*FIND_PREV()* Sets the current list item to the previous item in the current key's multiple item list and sets the data for the corresponding data variables.
Obj.*FIRST*() Go to the first "value of key" in the object If this is the lowest key will depend on the sort order of the hash object.
Obj.*HAS_NEXT()* Determines whether there is a next item in the current key's multiple data item list.
Obj.*HAS_PREV()* Determines whether there is a previous item in the current key's multiple data item list.
Obj.*ITEM*_SIZE Returns the size (in bytes) of an item in a hash object.
Obj.*LAST*() Go to the last "value of key" in the object. If this is the highest key, depends on the sort order of the hash object.
Obj._*NEW_()* Creates an instance of a hash or hash iterator object
Obj.*NEXT*() Go to the next "value of key" in the object. Sort order of the object determines if the key is actually larger.
Obj.*NUM_ITEMS()* Returns the number of items in the hash object
Obj.*OUTPUT*() Copy contents of the hash object to a SAS data set--in one step.
Obj.*PREV*() Go to the previous "value of key" in the object. Sort order of the object determines if the key is actually smaller.
Obj.*REF()* Consolidates the CHECK and ADD methods into a single method call.
Obj.*REMOVE*() Go to the hash object and remove the bucket to that holds information for this key.
Obj.*REMOVEDUP()* Removes the data that is associated with the specified key's current data item from the hash object.
Obj.*REPLACE*() The programmer provides a key and some data to the replace method. The method replaces the old contents of the hash object bucket with the new data
Obj.*REPLACEDUP()* Replaces the data that is associated with the current key's current data item with new data
Obj.*RESET_DUP()* Resets the pointer to the beginning of a duplicate list of keys when you use the DO_OVER method
Obj.*SETCU()* Specifies a starting key item for iteration.
Obj.*SUM()* Retrieves the summary value for a given key from the hash table and stores the value in a DATA step variable.
Obj.*SUMDUP()* Retrieves a summary value for the current data item of the current key and puts value in a DATA step variable.

As can easily be seen, many methods (see blue above) are centered on the tasks that a programmer encounters when s/he is moving up and down a hash object.

CREATING A HASH OBJECT: THIS ILLUSTRATES MANUALLY PUTTING VARIABELS ON THE PDV

Figure 8 simply shows the creation of a hash object. The hash object in Figure 8, without any other SAS code, does no work and is shown here for illustrative purposes only.

For space purposes, the code is shown on the input stack. The code would actually create the hash object when it executes (when it is in the upper right hand box), not as it sits in the input stack waiting for processing.

A hash object should only be declared once in a data step; at the top of the data step. It is left to the programmer to write do-group code to insure that it is only defined once. Figure 3 uses the code:

```
If _n_=1 then do;/*Evaluate 1 time*/
      Many Lines of C-like code to
     declare the object;
End;
```

The first line inside the do group is:

```
Declare hash
   H_O(dataset:"work.small"
          ,hashexp: 4     );
```

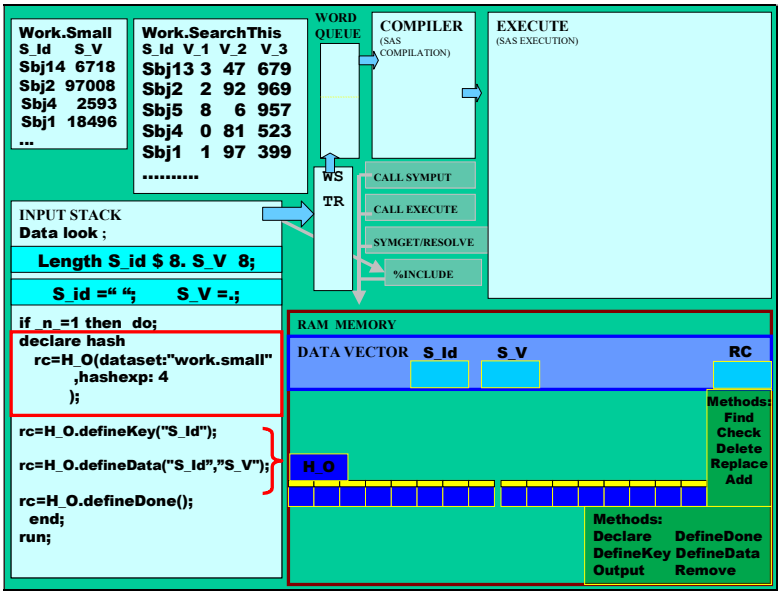


FIGURE 8

NOTE: If the declare statements are allowed to execute for each pass through the data step, the hash object will be destroyed and re-created at every pass through the data step. This is not efficient coding.

This command, when passing through the compiler, causes the compiler to recall code from a library and insert it into the SAS program. Additional dot commands, inside the do-group, affect what supporting methods are called from the library. When the SAS program executes, it will execute both the programmer's compiled SAS code and the included low-level code. The included low-level creates and manages the hash object.

Let's examine the code shown in Figure 8. We would like to create the hash object only once and so it is imbedded in a do-group that executes when the first observation is processed. The declare statement takes several parameters and will, when executing, create the hash object according to the parameters supplied.

`Declare HASH H_O` will create a hash object named H_O.

`Hashexp: 4` causes the hash object to be created with an initial size of 2^4 , or sixteen, "buckets". Figure 3 shows a hash object named H_O that has16 buckets in the top row.

`dataset:"work.small"` specifies that the hash object be loaded with information from the data set work.small. If there are more that 2^4 observations in work.small, the hash object will automatically grow "root like appendages" (see Figures 7 & 8) to hold observations.

`rc=H_O.defineKey("S_Id");` specifies that values to be used as Keys (input to the hashing function) are to be taken from the variable S_ID on the PDV. This command is an instruction, *to all methods (subroutines) that access the hash object*, to look in the variable S_Id on the PDV –for the current value of the lookup key (the value to be used by methods).

If one wants to find what the hash object holds for key "Sbj2477" , the value "Sbj2477" is put into the variable S_ID on the PDV (a SAS statement S_ID="Sbj2477"; is one way to do it) and the find method is called. Methods, like find, that manage data in the hash object will look in the variable S_Id, on the PDV, for the currrnet value of the Key…and look for an entry in the hash object with that value … ***and, if the value is found, return data from the hash object to the PDV.*** Find, I feel. Should have been named "Find and Return data".

`rc=H_O.defineData("S_Id", "S_V");` is an instruction *to all methods (subroutines) that store data on/ return data from the hash object* that the data goes to/comes from the variables S_Id and S_V on the PDV. Note that S_Id was defined as ***both*** key and data. While not required, defining the key variable as both key and data helps keep matching values of Key and data on the PDV and is generally good programming practice.

`rc=H_O.define.done();` is a command to SAS that says that there are no more parameters/commands intended for the hash object and its associated methods. This is like the end on a do-Group or the run at the end of a PROC or data step.

THE RC VARIABLE

Finally, the rc= is a very common component of dot syntax. Most methods produce a number (a return code and rc stands for return code) whenever they are called. This return code is 0 if the operation was successful and non-zero if there was a problem with executing (e.g. the find method was passed a value for key, from the PDV, and failed to find a matching observation in the hash object). The return code is automatically passed to a "system level temp variable". These "system level" variables are normally unavailable to user-written SAS code. It is good programming practice to redirect the value of the return code to a variable on the PDV, which *can* be checked using normal SAS code. The RC= part of the syntax redirects the return code from the system level temp variable to a variable called RC on the PDV. The variable does not need to be named RC.

Checking the value of RC is shown in code below. Re-direction can be tedious and is often skipped for the commands that set up the hash object (shown below). However, for methods that load/recall data to/from the hash object, checking the value of RC is the only way to check that the command executed successfully.

```
if _n_=1 then
  do;
  declare hash h_o(dataset: "work.small", hashexp: 16);
       /*if rc NE 0 then put rc= "declare problem";*/
       /*                  else put "declare OK";*/

    rc= h_o.defineKey("S_Id");
      if rc NE 0 then put "problem with key";
                  else put "key OK";

    rc= h_o.defineData("S_Id" , "sat_var");
      if rc NE 0 then put "problem with data";
                  else put "data OK";

    rc=h_o.defineDone();
      if rc NE 0 then put "problem with done";
                  else put "done OK";
   end;
```

EXAMPLES OF OTHER METHODS

If a programmer wants to find information stored in the hash object s/he can use the find method (if there are no duplicates of the key variable - more on this later.). The find method will go to the PDV and get the value of the key variable(s) – here S_id. The find method will use the value of S_id as input to the hashing function. Find uses the result of the hashing function (a number) to determine in/under which "bucket" in the hash object any data with this key should be stored. The find method will then go to top level bucket in the hash object. If that bucket holds the key (and data) that the programmer desires, searching stops and data values are transferred from the hash object back to the PDV. If the top level bucket does not hold the information desired, the root is searched until either the desired key or the end of the root is found. If there is no match in the root, the rc variable is set to a non-zero number. If the desired data is found, the RC is set to 0 and data is transferred from the hash object to the PDV.

If a programmer wants to replace information in the hash object with new information s/he can use the replace method. The replace method will go to the PDV and find the value of the key variable(s) – here S_id. The replace method uses the value of S_id as input to the hashing function. Replace uses the output from the hashing function (a number) to determine in which top-level bucket in the hash object data should be stored. The replace method will then go to the PDV to get "the new data" to be stored in the hash object (in this case the variables S_id and S_V). Replace will store both variables in one bucket in the hash object. The replace method replaces the old values in the hash object with new values from the PDV- values taken from the variables that were defined as data. Think of variables on the PDV as being key or data (or ignored by) for hash methods.

AN EXAMPLE OF USING HASHING IN A TABLE LOOKUP

OVERVIEW

Figure 9 is the first in a series of graphics showing details of using hashing to do the equivalent of a by merge-a table lookup.

As background, SearchThis is assumed to be a large master file of student information. We want all the information in SearchThis for the small group of students who are in the file named small. Small also contains a variable of interest called S_V (for Satellite Variable) and we wish to have S_V end up in the final data set. While hard to see in the graphic, the file named small is both short and narrow. All of small will be loaded into the hash object.

This program takes both variables from the file small, and stores them in a hash object called H_O. The variable S_id is defined as key. S_V is defined as data. When the file SearchThis is processed, values of S_id that come from the file SearchThis are loaded onto the PDV. For each observation in SearchThis, the find method hashes the value of the key variable. Find uses the hashed value of S_ID, to look in a top-level bucket in the hash object H_O and see if a match is

found. If a match is found the associated value of S_V is taken from the hash object and returned to the PDV and RC is set to 0. The contents of the PDV are output to the file named look.

ISSUES WITH THE PDV

Remember that only certain commands (set, merge, infile etc) cause the compiler to create variables on the PDV. The trigger to create variables on the PDV is not simply the mention of a data set name in your SAS code. In Figure 9, the file small is only mentioned in a declare statement and *declare is NOT a command to the compiler*. Declare is an execution command. Variables that are in small (and not in SearchThis), that are desired in the PDV, must be "manually put on the PDV' by the programmer. How this is done is shown in Example 1 (a better way is later) as we step through the process.

STEP THROUGH AN EXAMPLE OF A TABLE LOOKUP USING A HASH OBJECT – EXAMPLE 1

The example is broken down into three sections and the sections are shown inside red boxes in the following figures. In Figure 9 we see the first section of the example.

In the red box, SAS commands are issued that create a place for the variable S_V on the PDV. Unlike the statements in the _n_=1 loop , the statements in the red box in Figure 4 are commands to the compiler.

Hashing documentation often shows just a length command being used to create spots on the PDV, but coding just a length command will cause SAS to issue a warning when the program runs.

Code that initializes S_V (the Satellite variable) to missing will create a place on the PDV and prevent the warning. It is suggested that assigning the desired variable a missing value is sufficient.

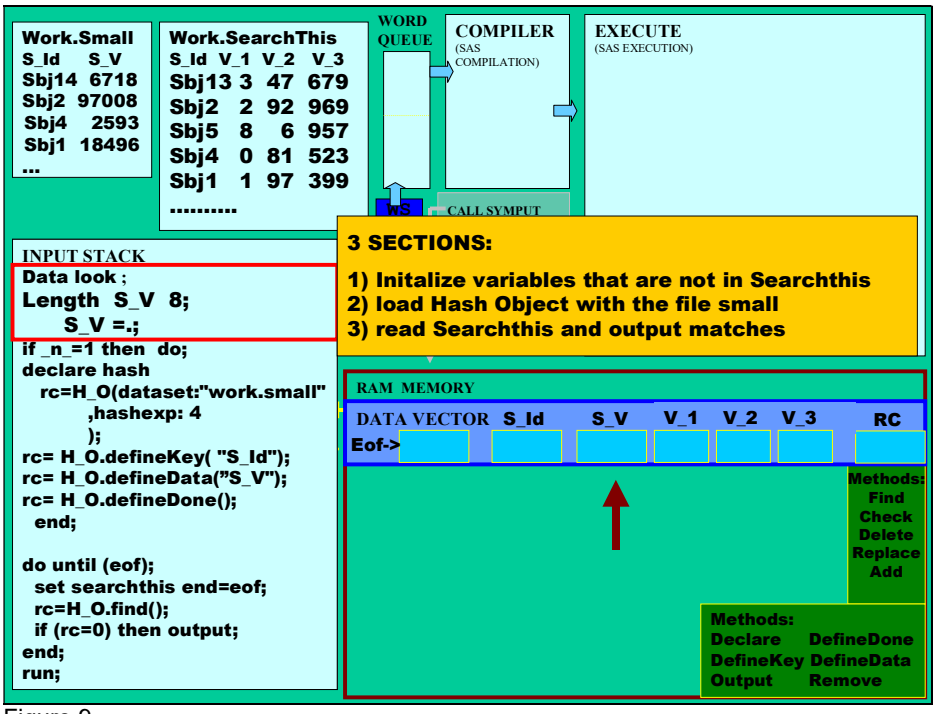


Figure 9

A set statement can be used to "put" these variables on the PDV – see later figure.

The variables S_Id and V_1 to V_4 do not need to be manually created because they are all in a dataset (SearchThis) that is mentioned in a set command and are put on the PDV as the compiler processes the Set SearchThis statement.

The set statement causes SAS to read the file header and create the PDV. S_Id is in both files (SearchThis and small). Because of its presence in SearchThis, S_Id does not need to be manually initialized (while S_V must be manually created on the PDV).

Figure 10 shows the result of executing the second boxed section of code.

All of work.small has been loaded into the hash object.

The first level of the hash object was filled up and, as more observations were hashed to “buckets already occupied”, the table grew the root-like additions.

Note that the variable RC is on the PDV.

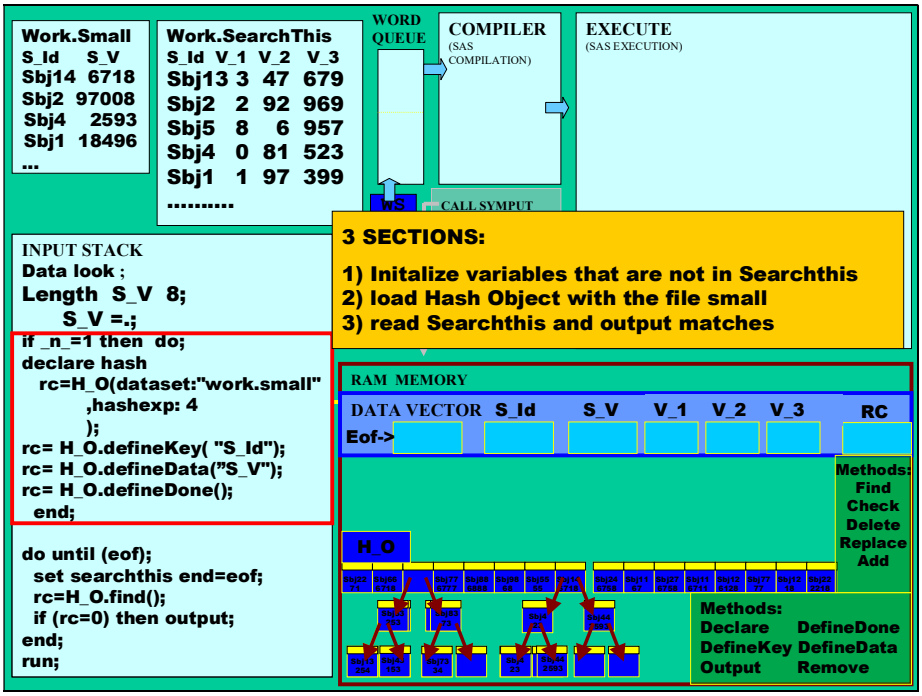


Figure 10

Figure 11 shows the execution of the third part of the code and the processing of a “failure to match”.

When the set SearchThis statement executes, the variables from SearchThis (S_Id V_1 V_2 V_3) are loaded into the PDV.

The Find method then executes.

Find goes to the PDV and gets the value of S_Id. Find hashes the character value (subj13) into a number which Find uses as the address of the top level bucket where/under which this information should be stored.

Find checks that top=level buckets in the hash object and all buckets in the root under that bucket.

In this case, there is “no match” and Find returns, to RC, a non-zero number. Because RC is non-zero, there is no output.

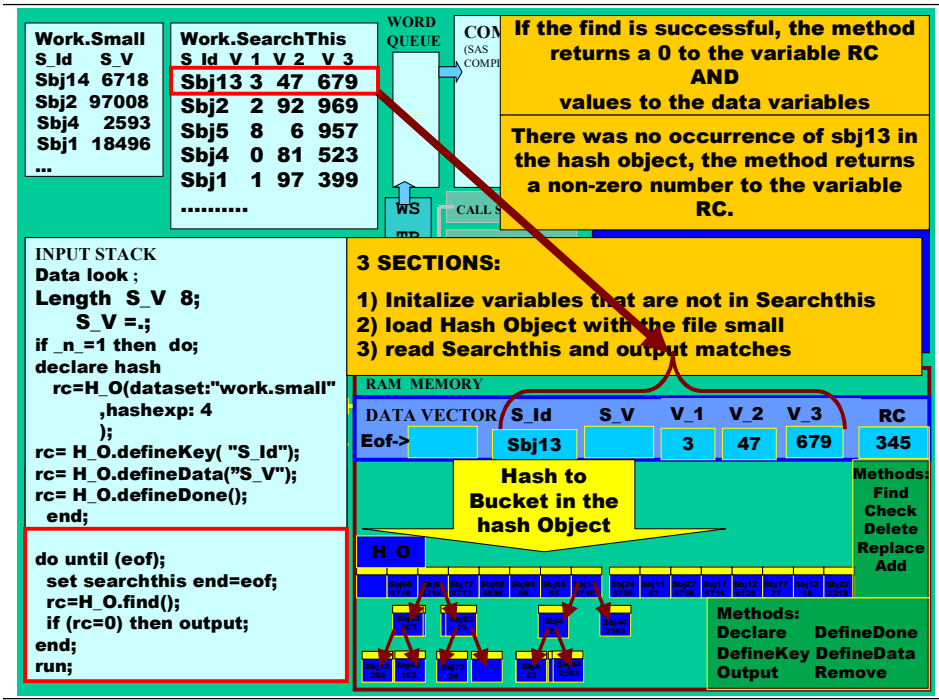


Figure 11

The do until loop executes again and another observation is read from SearchThis.

Figure 12 continues the example and shows processing of a “match”. When the set statement executes, values for variables in SearchThis (S_Id V_1 V_2 V_3) are loaded into the PDV.

Then the Find method executes. Find goes to the PDV and gets the value of S_Id. Find hashes the character value (subj2) and calculates in/under which top-level bucket (in the hash object) this information should be stored.

Find checks that bucket and all buckets in the root under that bucket. In this case, there is a match and Find returns two things to the PDV.

Find returns the value of S_V that was stored in the hash object and it returns a zero to RC. Because RC is zero, the output executes.

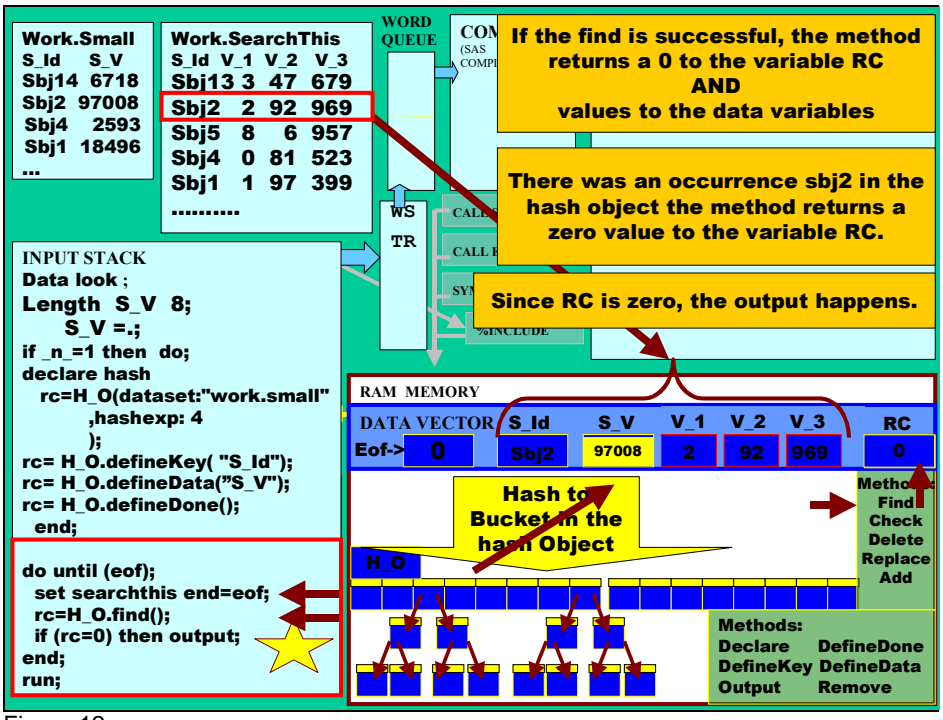


Figure 12

WHY IS HASHING SO FAST?

Figure 13 shows some of the reasons why hashing outperforms a format in table lookup. A format looks for an entry by performing a binary search. Figure 13 shows how SAS would find the value 7 in a format and in a hash object. This hash object has eight buckets in the top row and stores 32 different observations in the hash. The format also has 32 values.

FORMATS USE BINARY SEARCHES

Formats use a technique called binary search to find desired information in a format catalog. A format finds an entry by reading an observation at the middle of the format file (here that value is 16). SAS asks itself if it has found the entry it wanted. If it has found the desired entry, SAS stops searching and transfers the associated value to the PDV. Sixteen is not the desired value.

Since it not the desired value, SAS determines if the desired value is above or below the value just found. It is above. SAS no longer needs to consider rows 16 to 32. The new search range is 1 to 15. The binary search repeatedly divides the file in half. SAS picks the observation in the middle of the “current range” (the value 8) and asks if it is the desired value. If it is, SAS stops searching. It is not the desired value.

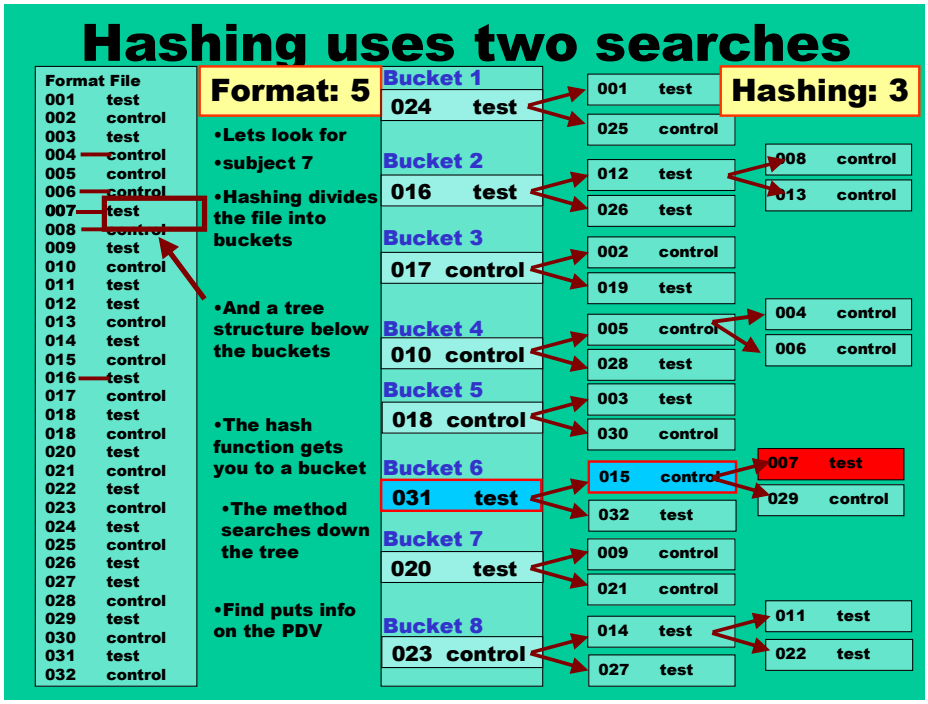


FIGURE 13

Since it not the desired value, SAS will decide if the desired value is above or below the value it just found. It is above. SAS no longer needs to consider the “bottom half “ of the old range and has a search range of 1 to 7. SAS picks the observation in the middle of the new range (4) and asks if it is the desired value. If it is, SAS stops searching. It is not.

The process of dividing the range in half continues until 7 is found. It takes five tries to find the value 7 using a binary search.

HASHING USES A TWO-PART SEARCH METHOD

Hashing is a mathematical operation that takes a number as input and, as output, produces an integer in a specified range.

Hashing is an efficient algorithm because its file has a different internal structure and a search that takes advantage of the structure. The best way to represent the hash object is a series of buckets with root structures growing from each bucket.

Methods take the key(s) off the PDV and hash the key(s) to find a top-level bucket – think of this as one step. Here, the hashing function takes the key value of 007 and returns a number; in this case a 6. SAS knows that the desired information is in bucket 6, **or in the root under bucket 6**. One-step hashing to the proper top-level bucket is one reason why hashing is faster than formats.

A method reads the top bucket and asks if this is the key SAS is looking for. If it is the desired key, the method puts the associated data on the PDV and stops.

In Figure 13, during hash table loading several keys hashed to bucket 6 and forced the creation of roots. Since the top bucket does not contain the desired key the method works down the roots. Roots are loaded/structured so that SAS only has to ask is "the desired key larger than the one in the current bucket" to be able to traverse the roots. If the desired key is smaller than the one in the current bucket, SAS looks along the upper fork in the roots. If the desired key is larger, SAS looks in the lower fork. Using this logic, SAS can find subject 7 in three steps. If the hash object had more buckets in the top level, the roots would be smaller, and the find would be faster.

Another reason that hashing is faster than format table lookup is that the hashing objects were designed for table lookup and formats are not designed for this task. Formats have a lot of overhead (do additional work) not needed for doing table lookup.

The figure to right shows a graphic of a binary search. You start at the exact middle of the file and repeatedly divide the file in half.

This makes, conceptually, searching a format file similar to searching one big root system.

Every binary search has to start at the top of the root system.

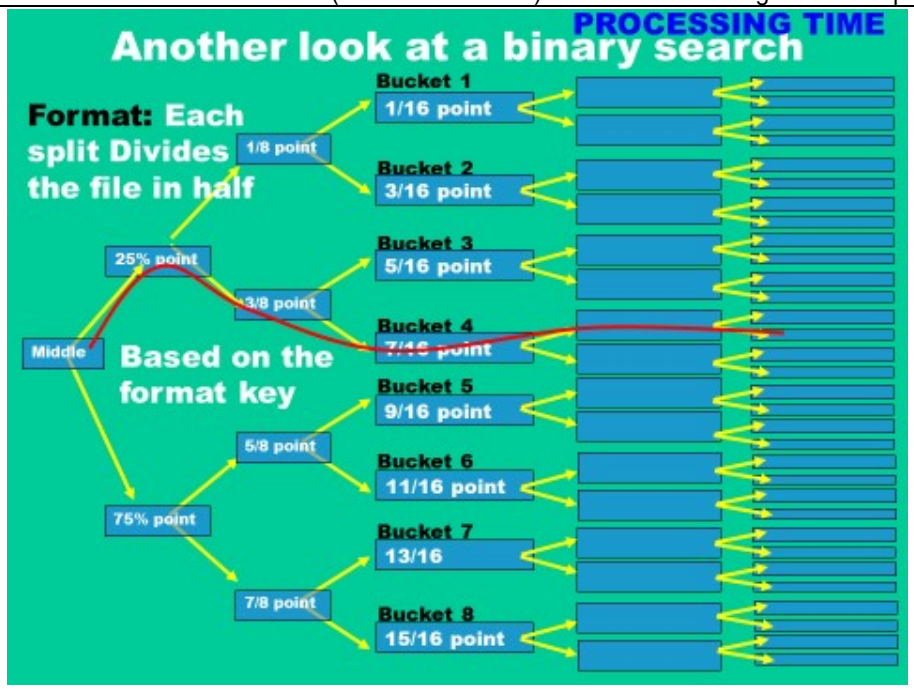


Figure 14

I think of the hash table as being like a root system with the top cut off.

If we create a 16 bucket hash table, the hash function allows us to, in one calculation, go directly to the proper bucket at the 5th level of the root system.

We can very easily create hash tables with 256 top-level buckets and the hash function, for that table, would jump immediately to the 7th level of the root system.

This can be a huge time savings.

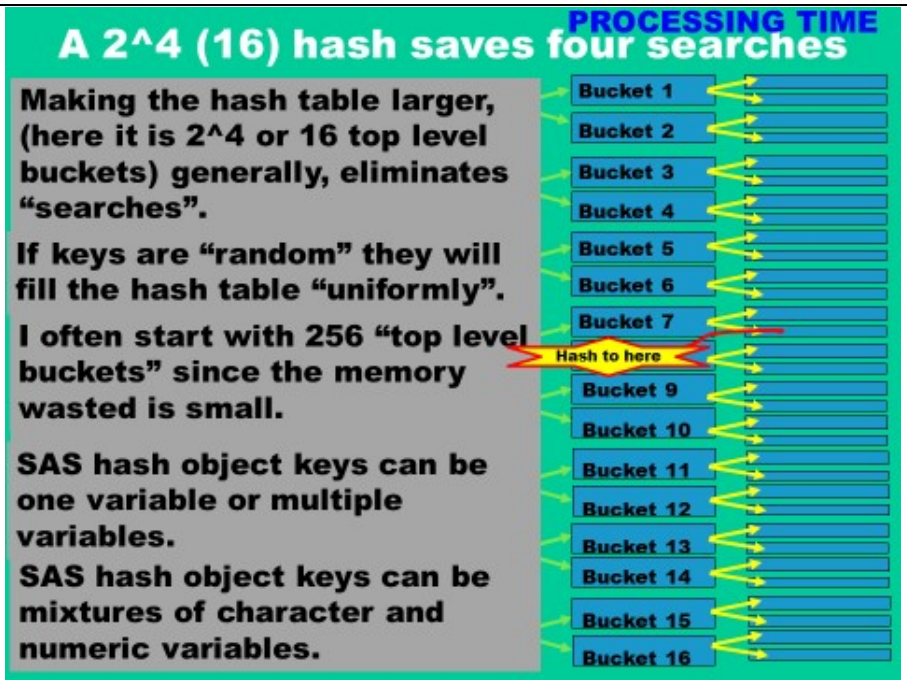


Figure 15

HOW MANY BUCKETS SHOULD I HAVE IN THE TOP LEVEL?

Any file can be stored in any hash object (limited by RAM availability). A hash object can be sized as 2^0 (just one) top level bucket and still hold a million observations. The root system would be deep and performance will likely be poor.

As can be seen from Figures 16 and 17, search time is **not** very sensitive to the depth of the roots. Every additional level doubles the capacity of the system, but only contributes to search time linearly. Search time is linear with number of levels.

Figure 16 shows how quickly capacity grows as roots grow. If a hash object has ***eight boxes*** in the top level and we load eight random observations in it, the objects will (tend to) fill up the first level. Any of these first-level observations can be found in one step. If the roots grow to nine levels deep we can find any observation in nine steps but the capacity of the table is 511 times greater.

The red level in Figure 16 shows how fast the capacity of the tree grows. Level 5 of the tree, alone, can hold 126 observations. The total hash object (with all five levels fully filled) can hold 248 obs. or 31 times as many observations as a one-level eight-box hash object. However, the max search time for a 5 level object is only (approx.) five times as long as a one level object.

Figure 17 has 16 boxes in the top row and shows how the number of buckets in the top row of the hash object affect the number of levels required to hold different numbers of observations. It also shows how capacity and search times grow. The number of buckets in the top level of the hash object is a major cause of the speed advantage of hashing so bigger hash objects can be searched faster. A suggestion might be to "create a lot of top level boxes (2^8) but not to worry too much about this issue.

Level Number	Obs in level N		Total Obs. in Object	Multiplier Vs. Lvl. 1
1 level	**8**		8	
2 level	16	24=8+16	24	3
3 level	32		56	7
4 level	64		120	15
5 level	128		248	31
6 level	256		504	63
7 level	512		1016	127
8 level	1024		2040	255
9 level	2048		4088	511

FIGURE 16

Level Number	Obs in level n	Total Obs. in Object	Multiplier Vs. Lvl. 1
1 level	**16**	16	
2 level	32	48	3
3 level	64	112	7
4 level	128	240	15
5 level	256	496	31
6 level	512	1008	63
7 level	1024	2032	127
8 level	2048	4080	255
9 level	4096	8176	511

FIGURE 17

LOADING THE HASH OBJECT AND GROWING TREES

"Even" loading of the hash object and creation of root systems of uniform depth is a desirable characteristic of a hash object. It can be achieved if the keys hash uniformly across the top level of the hash object. This is a common situation in practice. It will happen if keys are sequentially assigned, like subject id numbers, or customer numbers. However, if the keys being loaded into the hash object is oddly distributed, the hash trees can be of different sizes/depths.

ADDITIONAL HASHING EXAMPLES

When studying hashing, there is no better advice than: "read the papers that Dr. Dorfman and Don Henderson wrote". These papers contain the state of the art techniques. This paper examines techniques that Dr. Dorfman and Don Henderson published in SUG proceedings.

EXAMPLE2: FROM AN UNSORTED DATASET, PRINT OBS TO THE LOG IN SORTED ORDER

The code to the right, as a learning exercise, loads an unsorted data set into a hash object and then puts the sorted data to the SAS log.

`"If 0 then set scores"` is used to create a PDV that has variables from scores.

This statement allows the SAS compiler to see the file header for scores and put variables from scores on the PDV – but does NOT read data into the PDV.

The RC variable is on the PDV because the compiler sees the statements starting with RC=.

Code in the box creates a hash object, called HoldSort, with 2**2 (=four) buckets in the top row. Dataset:"scores" causes the data set scores to be loaded into the hash object.

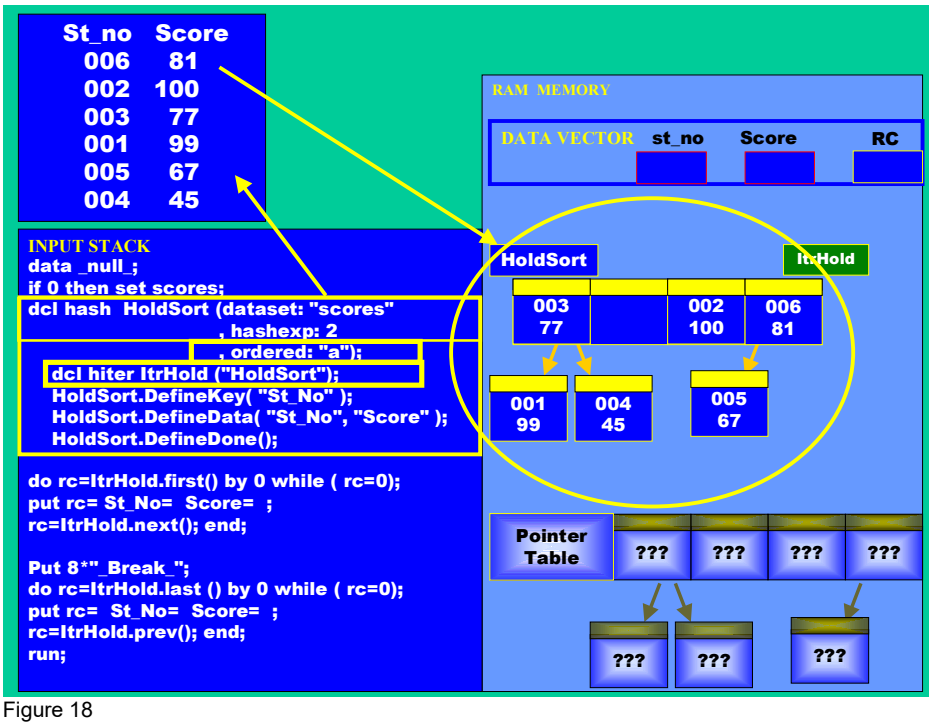


Figure 18

Ordered: "a" causes the creation of another "table" that is used by the iter method to move the pointer, in key order, up/down the hash object. There are indications that this is done using a separate (secret) hash table of pointers. The question marks were put in the graphic to indicate how little I know about this memory resident object.

```
Decl hiter ItrHold ("HoldSort");
```

causes the loading of a method that uses the pointer table to traverse the HoldSort hash object in key order. A Hiter object works on ONE Hash object. A datastep can have more than one hash object and more than one Hiter. Accordingly, this syntax names the Hiter (`ItrHold`) and specifies the hash object on which the Hiter operates (it operates only on HoldSort).

Figure 19 shows (what might be) the loaded hash object. I have not found a way to query the true structure of the hash object. Figure 19 shows a loading that *could* occur.

The boxes in Figure 19 use the Hiter method and the "secret" pointer table to access the hash object.

The line of code

```
Do rc=ItrHold.first()
      by 0 while (rc=0);
```

accesses the bucket associated with the first key in the hash object **and moves data from the hash object to the PDV**. If there is a successful lookup, a value of 0 is moved to the variable RC.

The put statement writes to the log. Note that this example uses a SAS put, which "writes" information on the one observation that is in the PDV.

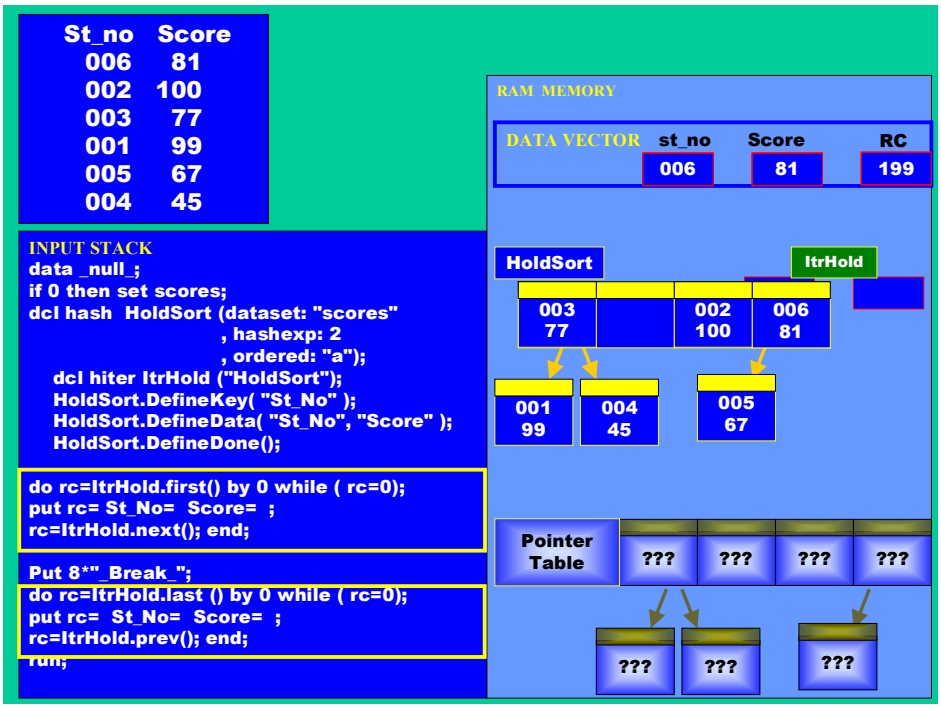


Figure 19

The code

```
Rc=ItrHold.next();
```

uses the pointer table to try to move the data associated with the next higher key from the hash object to the PDV. If there is a successful access of the hash object, a value of zero is loaded to the variable RC. The do loop continues as long as RC=0, as long as there is a successful lookup. This will traverse the whole hash object. In our code we will traverse the hash object in an upwards direction (key variables low to high) and then in a downwards direction.

Figure 20 shows the log for the above run. The double headed arrows show the boundaries of the loops.

The logic for the first loop is:
1) *Once*, return data for the **first** key to the PDV. *Many times:* Test for a successful return.
2) put, from the PDV, to the log
3) use hiter to move data for the **next higher key** from the hash object to the PDV
4) loop to 1)

The logic for the second loop is:
1) *Once*, return data for the **last** key to the PDV. *Many times:* Test for a successful return.
2) put, from the PDV, to the log
3) use hiter to move data for the **next lower key** from the hash object to the PDV
4) loop to 1)

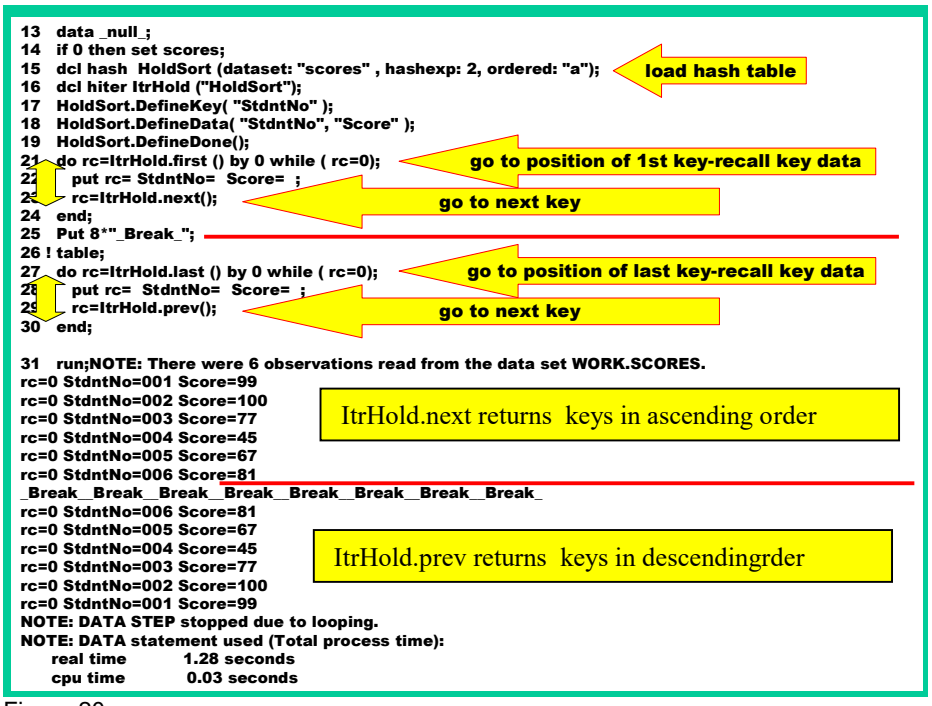


Figure 20

EXAMPLE 3: ILLUSTRATING THE FUNCTION OF THE OUTPUT METHOD

This example contains poorly constructed code in order to illustrate the function of the output method. It outputs the whole hash table for each pass through the loop.

The program is similar to previous examples and the top part of the code will not be explained.

There are some differences between this example and previous. The put statement was removed from the upper loop and

```
rc=holdSort.output
(dataset: "Asc_Sort");
```

was added to the upper loop

```
rc=holdSort.output
(dataset: "Desc_Sort") ;
```

was added to the lower loop.

Note that the hash object was created with the ordered descending option.

Figure 21 shows the log for the above code. Unlike a SAS output, which writes the ONE observation in the pdv, an output ***method*** sends the whole hash object (multiple observations) to a data set.

In the section of the log above the "__break line____break line__ " we see that the top loop sends the hash object (6 obs) to a file during every pass through the loop.

```
do rc=Iter4Hold.first()by 0
          while (rc=0);
 rc=holdSort.output
    (dataset: "Asc_Sort");
 rc=Iter4Hold.next();
end;
```

This illustrates how an *output method* differs from a SAS put or SAS output.

Note that this is a first-to-last loop on a hash object that was created with the ordered descending option.

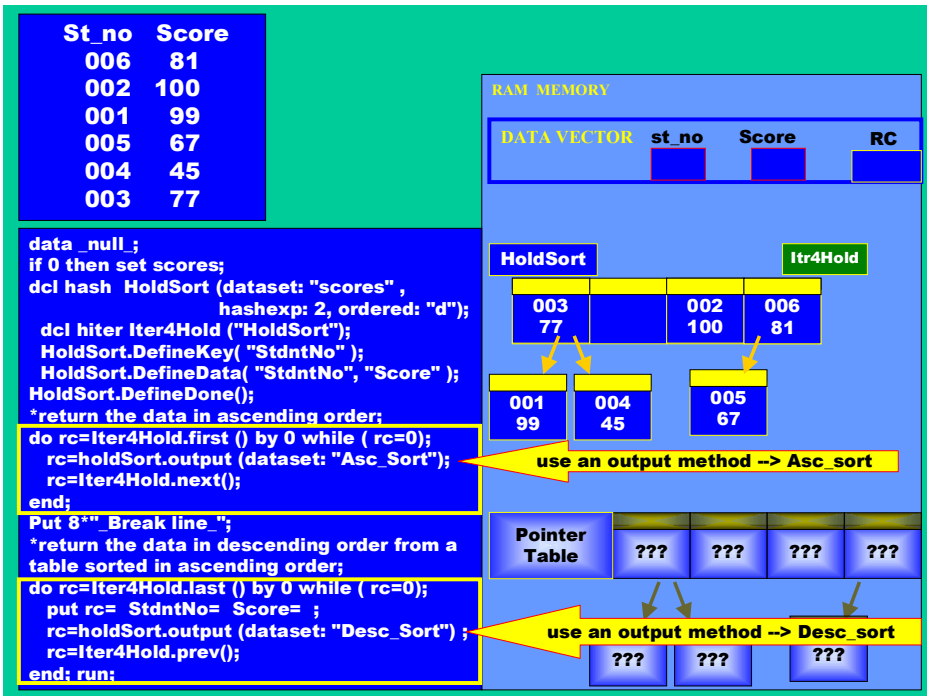


Figure 21

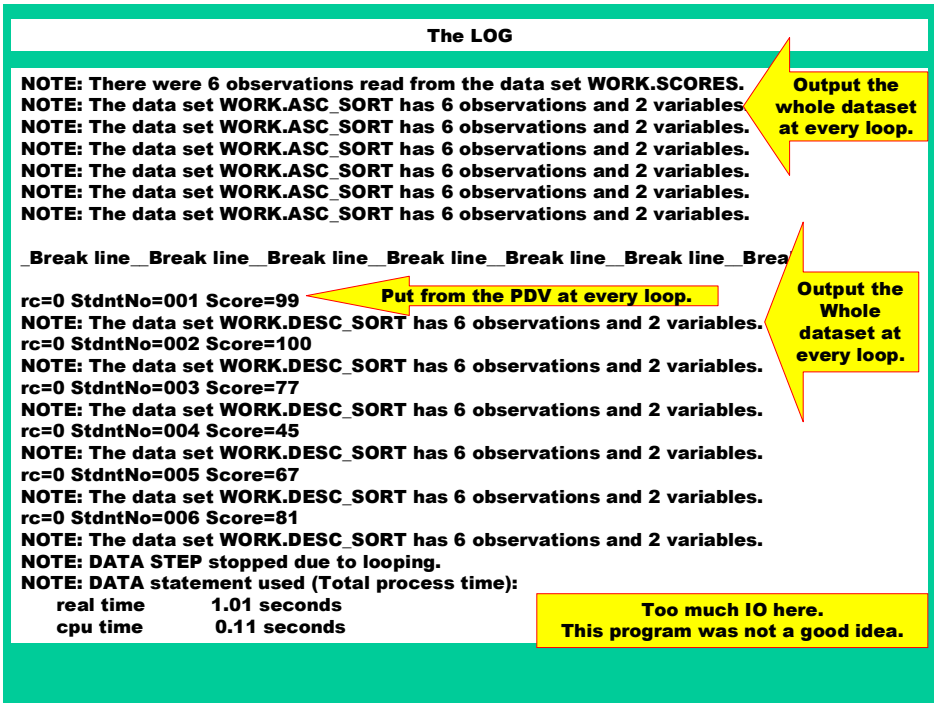


Figure 22

In the section of the log below the "__break line____break line__" we see the results of a Base SAS put as well as a hash output. The code is shown below and the critical idea to grasp is in the yellow colored comments.

```
do rc=Iter4Hold.last () by 0 while (rc=0);/*go to last element of the sorted hash*/
  put rc=  StdntNo=  Score=  ;  /*a SAS Put "writes" the PDV, not the whole hash object*/
  rc=holdSort.output (dataset:"Desc_Sort"); /*Note that this one command "writes" 6 obs */
   rc=Iter4Hold.prev();  /*try to return the previous element*/
end;
```

Figure 23 is a PROC Print on the data set, asc_sort, that was created in the upper loop in the previous example.

The data prints in descending key order, even though the output method was imbedded in a first-to-last loop.

From this we can see that the output method moves the data from the hash table to the output file, using the order option specified when the hash object was **created**.

The fact that the output method was imbedded in a first-to-last loop **did not affect** the order of the observations in the output data set.

The order is determined by the sort order of the hash object, not the loop in which the output is imbedded.

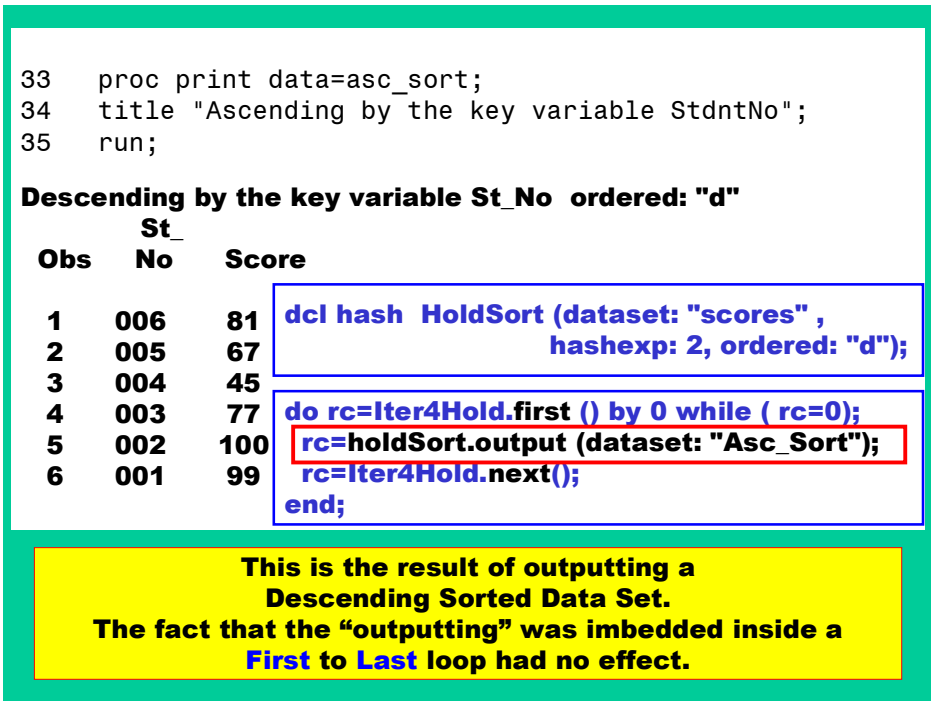


Figure 23

EXAMPLE 4: USING A HASH OBJECT TO SORT A DATA SET

Example 4 is similar to Examples 2 and 3, with the difference being that it focuses on the SAS output statement, not an output method.

This illustrates that the four statements:

```
Do rc=Iter4Hold.first()
   by 0  while ( rc=0);

Rc=Iter4Hold.next();

do rc=Iter4Hold.last() by 0
              while (rc=0);

rc=Iter4Hold.prev();
```

all move data from the hash object to the PDV.

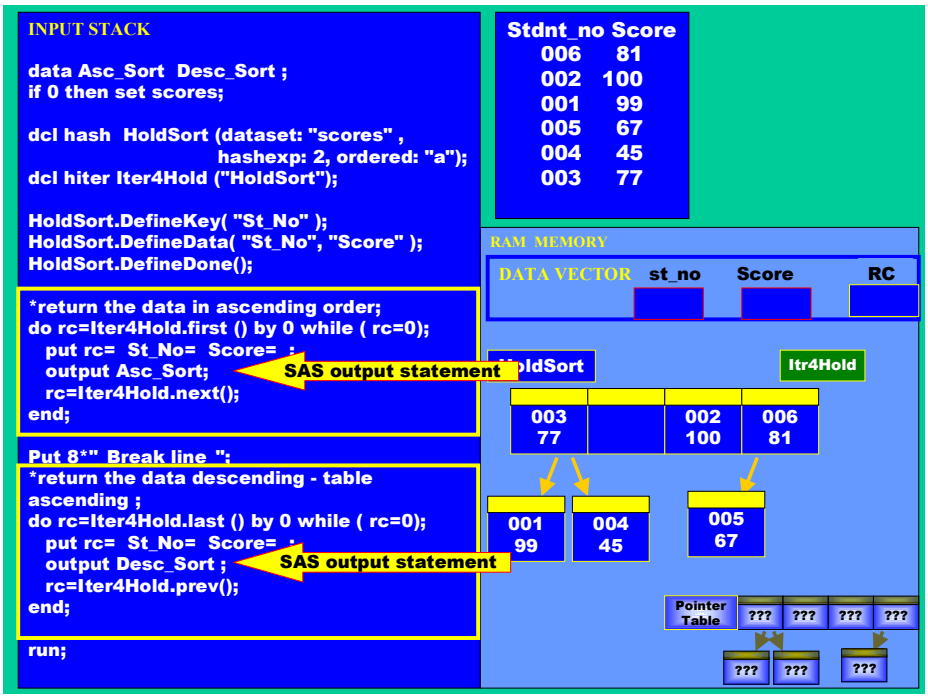


Figure 24

A SAS *output* ***statement*** moves the data from the PDV to the output file. Unlike the *output* ***method***, which moves the whole data set to the output file, the *output* ***statement*** moves the observation that is in the PDV to the output file.The choice of using syntax that combines a "first" with a "next" syntax or syntax that using a "last" with a "previous" depends on how the data set was created and what the programmer desires as an output.

In a hash table that is created in **descending** order, the statement
`do rc=Iter4Hold.first () by 0 while ( rc=0);` will first return data for the **highest** valued key and

and

`do rc=Iter4Hold.last () by 0 while ( rc=0);` will first return data for the **lowest** valued key.
The "last" observation in a "descending sorted" hash object will contain the smallest key.

EXAMPLE 5: SUM SALES FOR A ZIP CODE IN A HASH OBJECT – AVOID PROC SUMMARY

This exciting example shows how to use a hash object to replace a PROC Summary. PROC Summary is memory resident, like the hash object, but is so feature laden that it can, under certain conditions, be slower than a hash object. PROC Summary is very fast, but was designed to do many things. Hash objects can be faster, because they were designed to do fewer things.

In Figure 25 we see that the PDV and the hash table have been created.

The issue of creating proper variables on the PDV was handled by a combination of statements. The code below puts variables on the PDV.

```
length SmSales 8 ;
if 0 then set zip_sales;
```

The PDV contains variables called EOF_loading and RC because the compiler was able to "see" these variables as the data step was compiled. EOF_Loading has a different look, to emphasize its (slightly) different function as a loop controller.

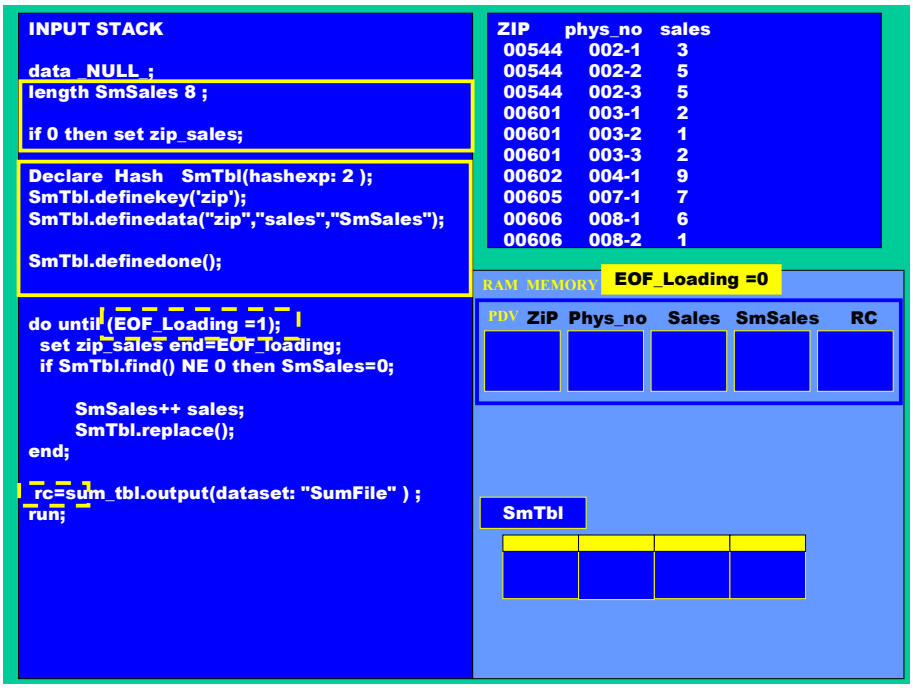


Figure 25

NOTE that the code (`if 0 then set zip_sales;`) used in the figure above Allows the SAS compiler to "see" the file header for Zip_sales and put variables from Zip_sales onto the PDV.

Figure 26 shows details of processing the first observation in the data set. The set statement pulls data into the PDV. The code

```
if SmTbl.find() NE 0 then
                SmSales=0;
```

tries to recall data for 00544 from the hash object and fails.

Failure causes RC to contain a non-zero value and this triggers the assignment

```
SmSales=0;
```

The statement below:

```
if SmTbl.find() NE 0 then
                SmSales=0;
```

is an elegant bit of code. This one statement does a lot for us. This is a good learning example

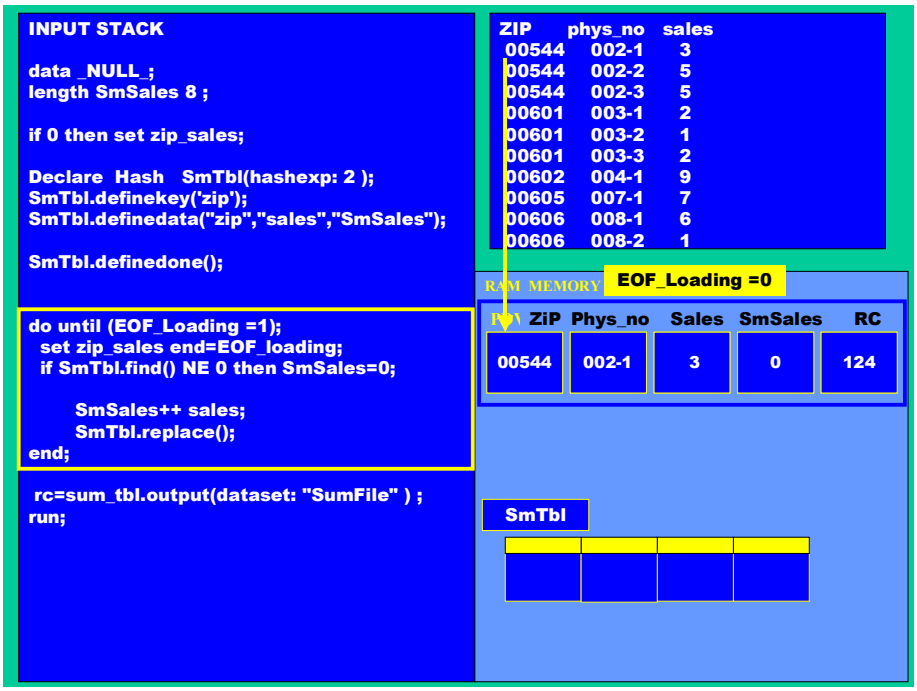


Figure 26

SAS does allow modification inside the elements of a hash object and this example can be made faster. Above, we recall the data from the hash object - to the PDV – modify it in the PDV and send it back to the hash object. The code in the Box above does that and helps illustrate how the hash process operates.

Figure 27 shows the summing (0 +3) happening in the PDV.

The summing happens through the action of

```
SMSales ++ Sales;
```

The code

```
SmTbl.replace();
```

sends the value from the PDV back to the proper place in the hash object.

Since there was no key 00544 in the hash table, the replace method functions as an add method and added this observation to the hash table.

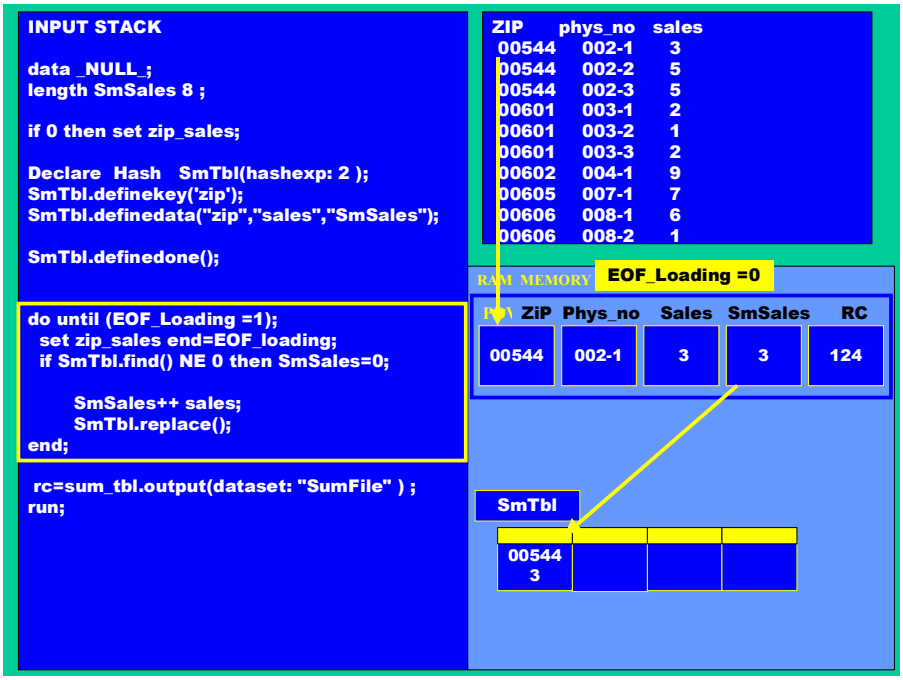


Figure 27

Figure 28 shows the final step in the process. It shows the processing of the last observation from zip code 00606.

After the last observation has been processed, the flag EOF_Loading is set to 1 and control escapes the loop.

The whole file is sent to a data set via the output method.

The hash object contains the sum of sales for a zip code and use of a PROC Summary has been avoided.

Again, SAS has created PROC Summary to have many useful features and these features can slow PROC Summary down.

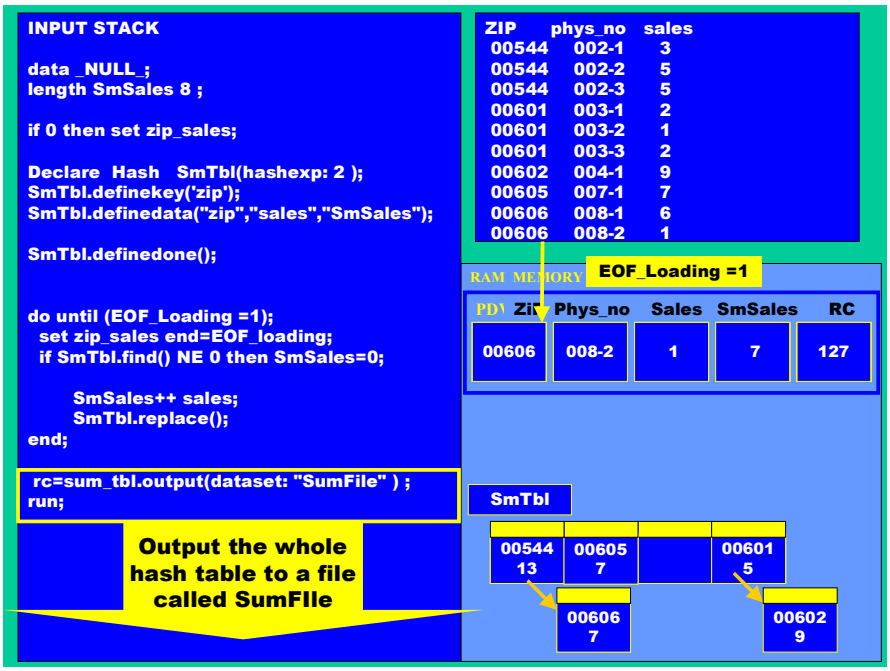


Figure 28

As SAS executes PROC Summary code, it must “stop and think” if a certain helpful option was selected by the programmer. Even if the option was not selected, the periodic checking can slow SAS down a bit. The Hash objects were designed to just do this process and have “low programming overhead”.

EXAMPLE 6: CREATE AND USE A HASH TABLE – AVOID A PROC SUMMARY

This example is an extension of the previous example. This shows the *creation, and use,* of a hash object in one data step.

Here we calculate the each physician's percent of sales in his/her zip code. The data values in the hash object, the sum of sales for the zip codes, will be used as denominators in the percentage calculation.

Assume the hash object was created as was shown in Example 5.

EOF_Loading and EOF_Using are in the PDV. They have a different look, to emphasize their (slightly) different function as loop controllers.

Figure 29 shows the state of the system after the creation of the hash object. The code in the yellow box is about to execute.

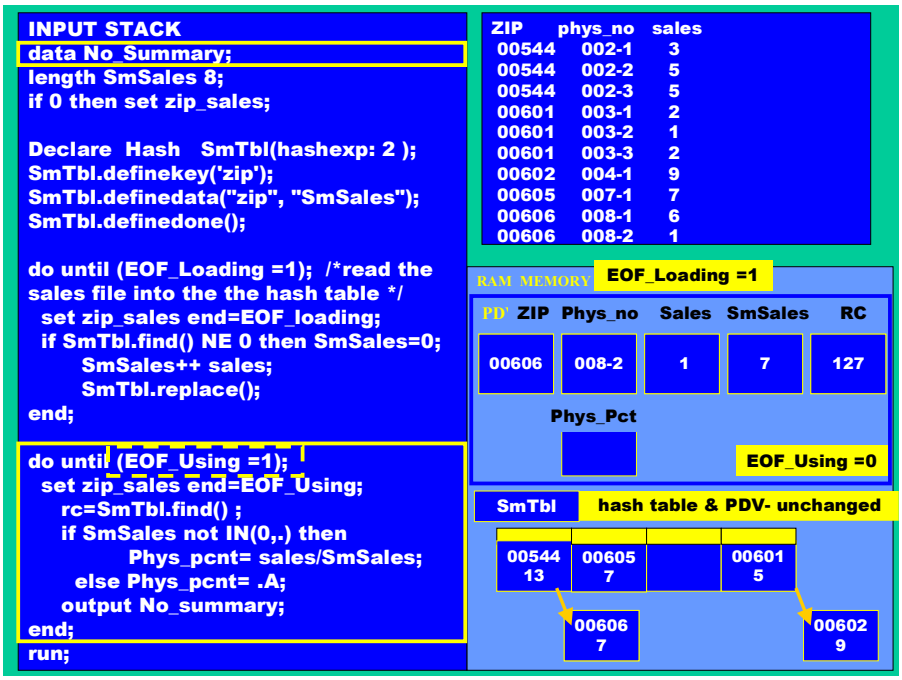


Figure 29

Figure 30 shows the execution of the first pass through the loop in the yellow box.

The set statement reads the data. The find recalls the sum of sales (SmSales) for that zip back into the PDV.

An IF statement is used to avoid division by zero if the recall "fails".

A SAS output **statement**, inside the loop, is used to send the contents of the PDV to the output file.

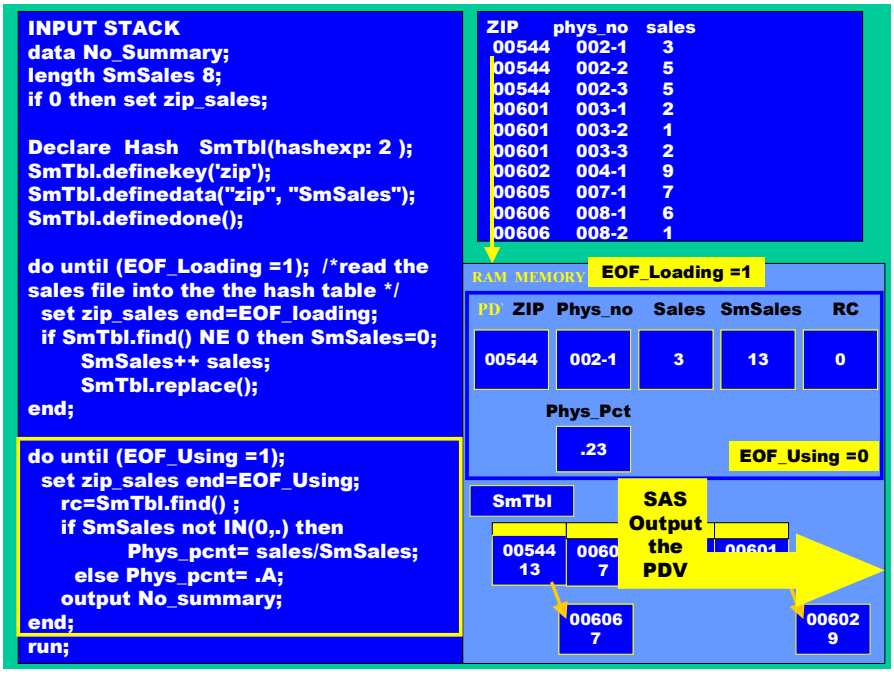


Figure 30

EXAMPLE 7: SPLIT A SORTED FILE BY A VARIABLE VALUE - USING A HASH TABLE

Example 7 shows how a sorted file can be split using a hash table. The author thinks that the normal way of splitting a file:

Please note that the code

```
Data Males Females;
Set SASHELP.class;
If sex="M"
   then output Males;
Else if sex="F"
   then output Females;
Else put "odd sex at " __n__;
Run;
```

is a more practical way of achieving the goal of splitting files.

In this example, the data step must be sorted before this hashing based technique and the "normal" way to split a file does not require sorting.

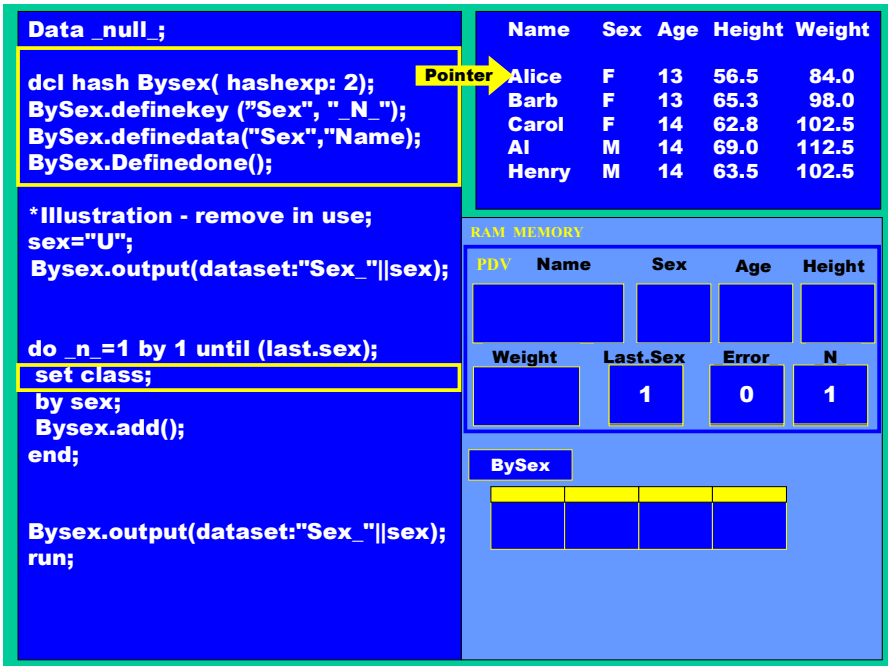


Figure 31
The example contains extra code to (commented as illustration) that provides extra information on the process.

It is possible to have a datastep process an unsorted file and, using multiple hash objects created inside one data step, and send girls to one hash object and boys to another. Having tow hash objects would eliminate the need to sort the source data and be faster. However, the example above is of theoretical interest, because it illustrates how a programmer can make the PDV and the hash object interact.

The code in the yellow boxes creates the PDV and the hash table. The Sex="U" and the following statement allow us to see that the Hash table is being cleared and re-used.

Each observation is read into the PDV, processed and loaded into the hash table. The composite key (sex and _N_ are both used by the hashing function) insures that all Females are not sent to one "top level bucket" and Males to another one other "top level bucket", as would be the case if Sex alone were the key. Composite keys add great flexibility to the hash table.

As review, as each observation is read from a SAS file, SAS advances a "Read this Row" pointer that it uses to keep track of how far down the data set it has read.

The "Read this Row" pointer is how SAS keeps track of how far it has read as it processes a file. Figure 24 shows this pointer pointing to Alice

Figure 32 shows the state of the system after the last female has been read. When last.sex has a value of 1, it causes control to exit the loop and dump the current contents of the hash object to a file. The until test happens at the bottom of the loop and allows processing of the observation where last.sex=1.

The hash object contains only females and the PDV has a value of F for sex. The name of the output file is created by concatenating the string "Sex_" with the value of sex from the PDV and the file ends up with a correct name.

In one pass from Data _null_ to run, we have processed all the females.

After escaping the loop, SAS processes the output method and loops to the top of the data step.

Importantly, the pointer in the data set does not advance. It remains where it is, positioned to read the first male.

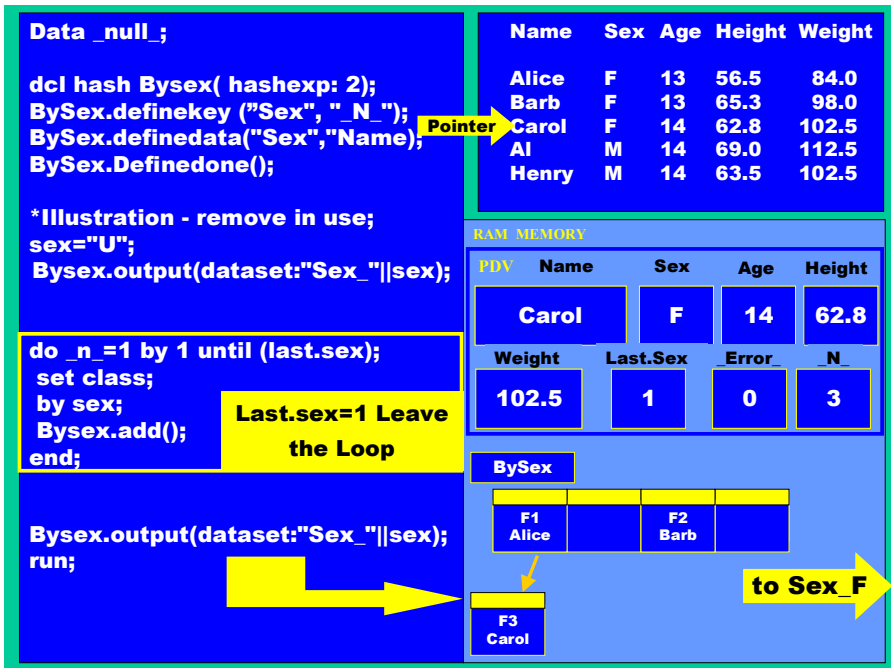


Figure 32

Between figures 32 and 33, program control has moved down to the run statement and back to the top of the data step.

Note that the dcl hash code is not inside a

```
if _n_=1 then do;
   -CODE-
 end;
```

and executed for every loop through the data step.

As the declaring (dcl) statement executes it causes the hash object to be cleared. This can be verified by the fact that the data set sex_U has zero observations.

This clearing of the Hash object is critical to this process and allows the table to be re-loaded - with only males.

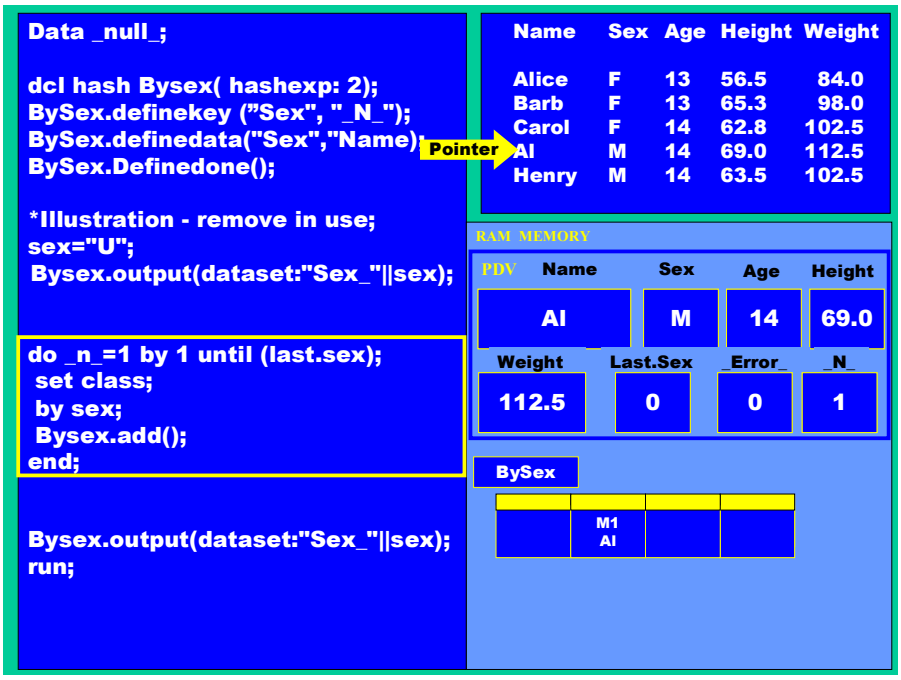


Figure 33

Figure 33 shows the state of the system after it has processed the data for Al. New values are being loaded into the hash object and when last.sex has a value of 1, the hash object will be dumped to a file called Sex_M.

This is an example of a nested loop with an unusual control mechanism. The outer loop is being controlled by the normal base SAS Data –run loop. This outer loop controls reading different values of sex.

The inner loop is the do _n_=1 code. The inner loop, controlled by the do-end loop, is used to read observations with the same value of sex .

EXAMPLE 8: HASH TABLE WITH DUPLICATE KEYS

When SAS introduced the hash table it did not have the ability to store duplicate keys. The hash table had been designed only for fast table lookup – but programmers quickly saw that hash tables could be a programming tool with much more power.

The figure to write, on the top half **(Output FAILURE)**, shows a failure to load multiple keys into the same hash object. We wanted to load seven values into the hash object and are only able to load four.

The bottom half of this figure **(Output success)** shows how, by coding multi-data:"Y", SAS is able to load duplicate keys into the hash table.

The next problem will be to retrieve data from the hash table when key values are duplicated in the file we wish to load into the hash table.

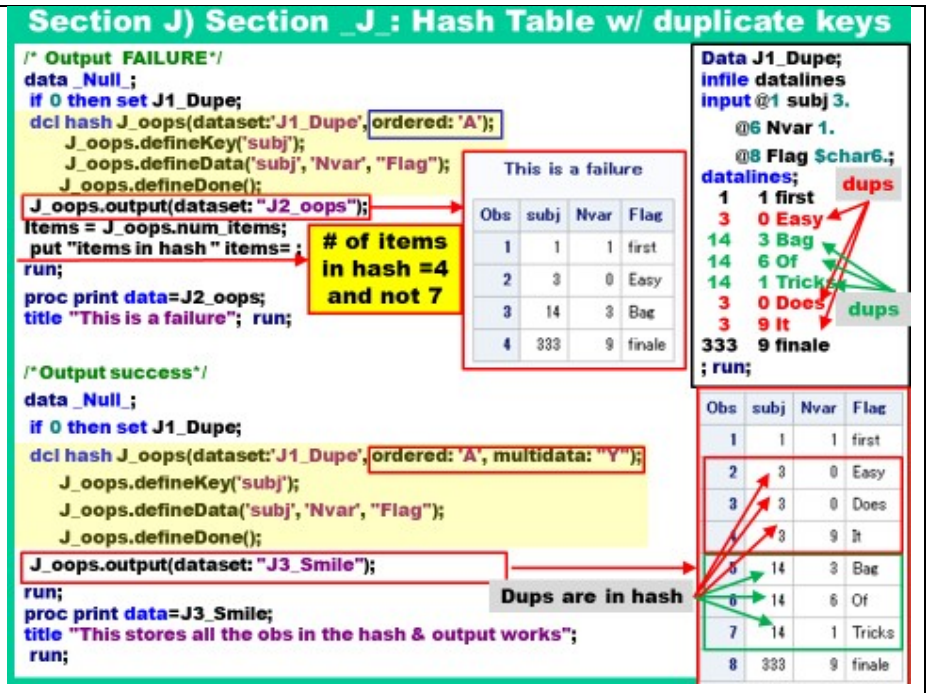


Figure 34

The figure to right, like the figure above, shows a compare and contrast situation.
The top half of the figure (**Traverse FAILURE**) shows how the .find method fails to recover all of the values when there are duplicate keys in the hash table.

The bottom half of the slide (**Traverse SUCCESS**) shows the code required to retrieve values from a hash table that has repeating key variables.

The next figures will show my attempt to provide a graphical illustration of the SAS system in this situation.

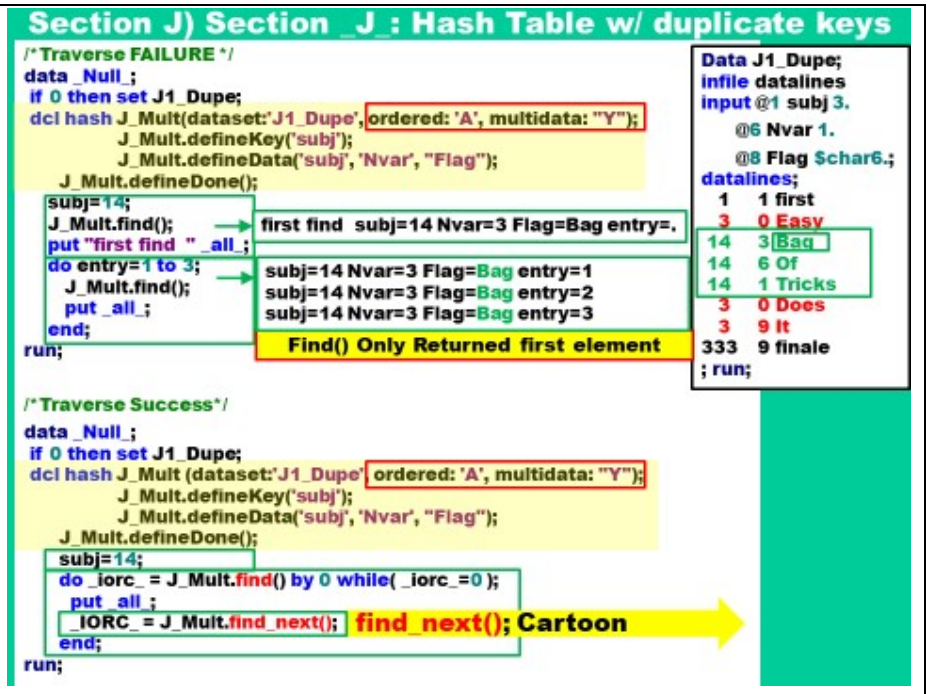


Figure 35

The figure to write shows more detail about this example. Please pay particular attention to the representation of the hash table.

Under subj IDs 3 and 14, SAS stores multiple rows in the hash table.

Note that subj has been defined as key.
Note that sunj, Nvar and flag have all been defined as data.

Our task will be to move values from the program data vector to the hash table and then back to the program data vector.

Defining variables as key and data tells the methods, that do the moving of data between the PDV and the Hash table, what parts of the PDV will be involved.

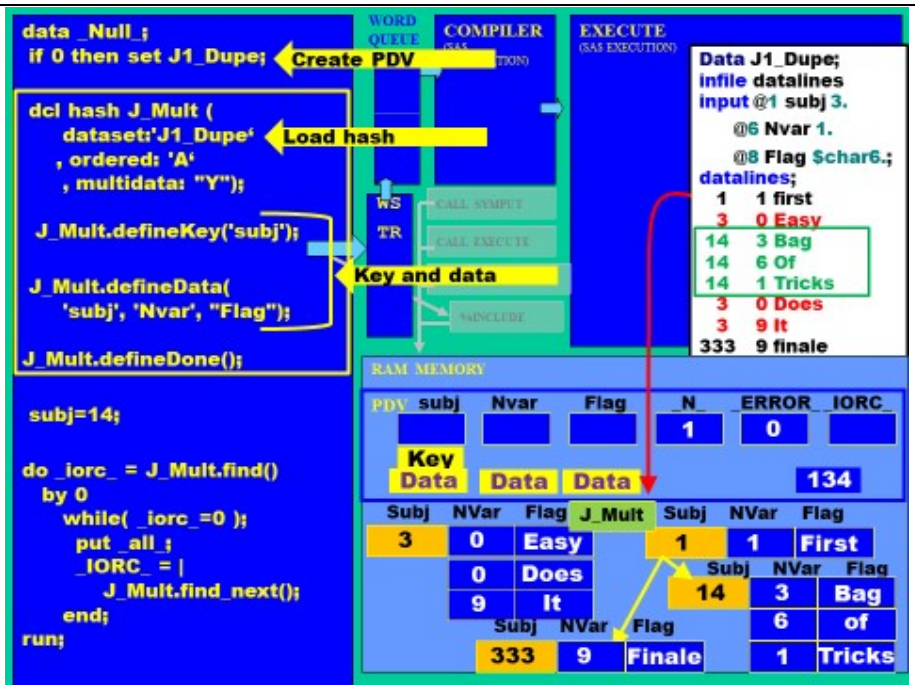


Figure 36

To make this example easier to cartoon, just below the gold box on the left-hand side, **I hardcode subject equals 14.**

Let's step through how that is processed.

14 is put into SUBJ on the PDV.

The next statement starts a loop and the find takes the 1st observation from SUBJ=14 and moves that value to the data variables on the PDV.

There is a pointer on the hash object that keeps track of which piece of the hash object has just been read.

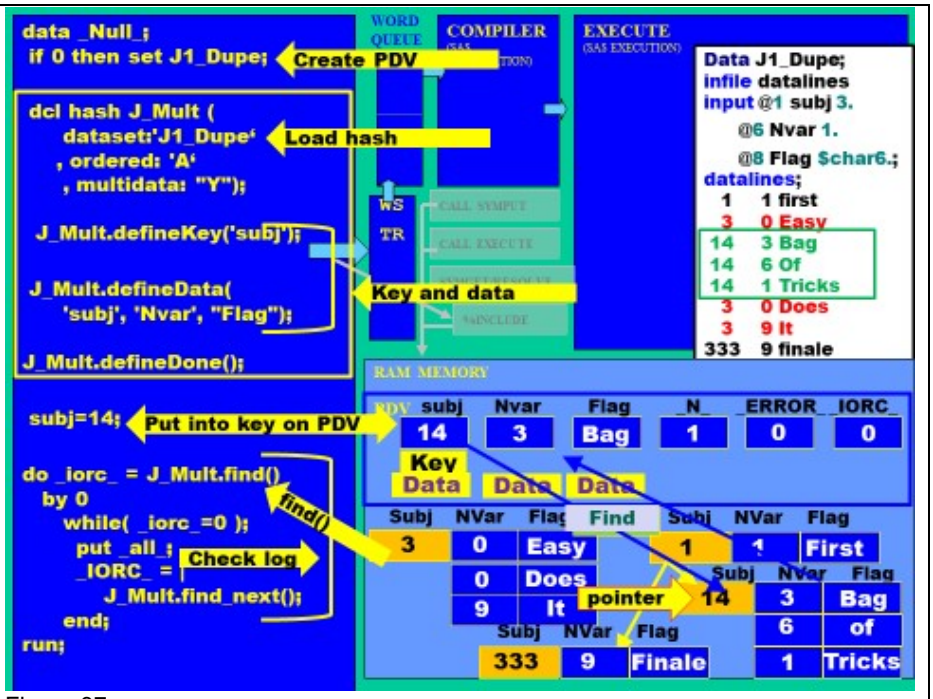


Figure 37

In the figure to right we see that the put statement executes.

This is done to facilitate learning of the internals of this process.

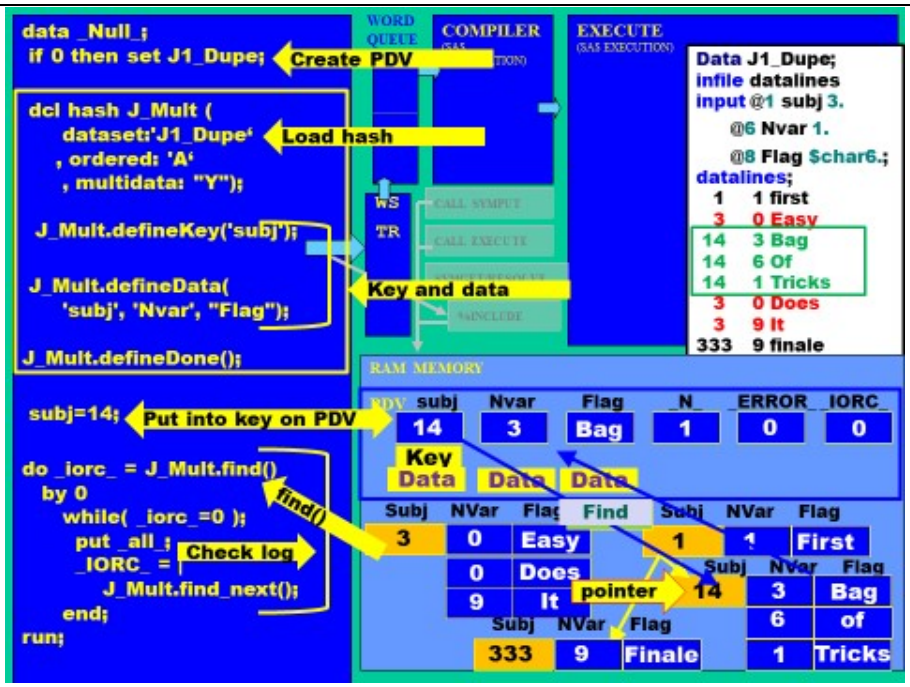


Figure 38

In the figure to right we see the find_next method execute.

It looks for the **next value** for subject 14.

The method places its “return code” of zero, in _IORC_.

A successful find_next returns a 0 (note that the while statement is testing for a 0)

The PDV now contains information from the 2nd entry for subject 14.

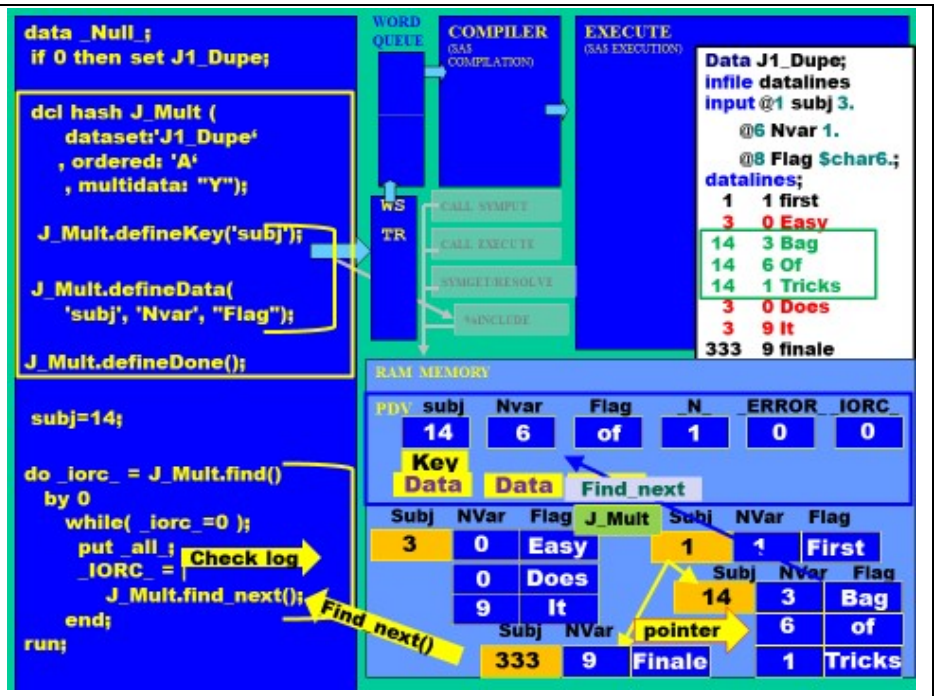


Figure 39

The figure to right shows the next execution of the find_next.

Note the values on the PDV.

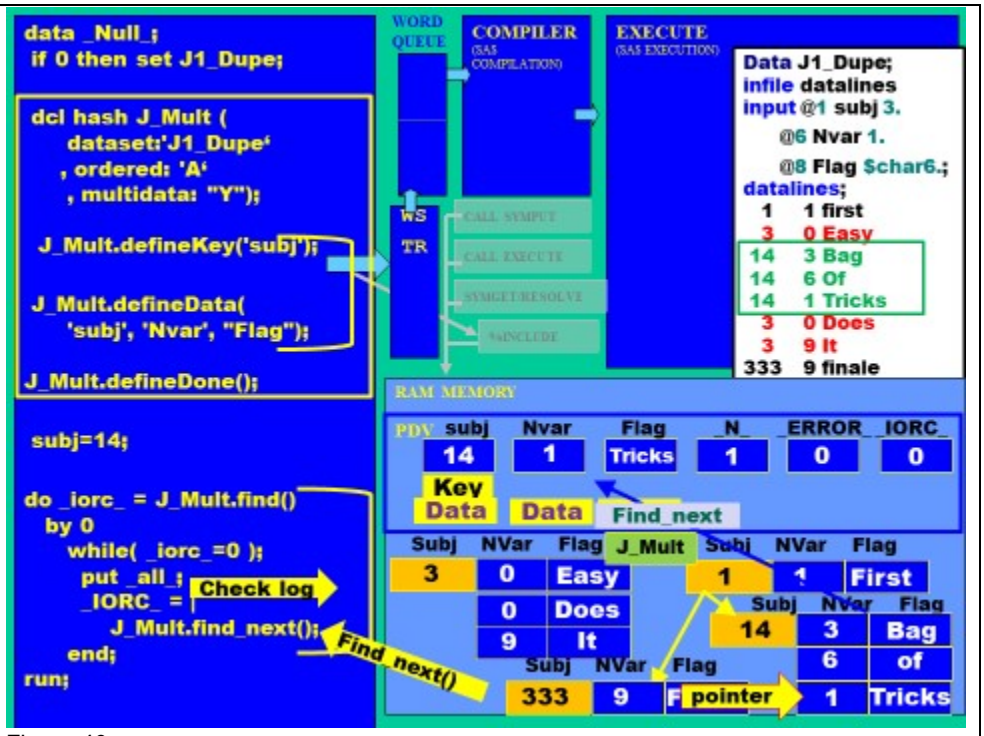


Figure 40

The figure to right shows how SAS escapes the loop.

When the pointer has read the last observation for subject 14 the find_next returns a non-zero value to the variable _IORC_.

That causes SAS to escape from the while loop.

I wish it were possible to do other examples that showed a more complete, and practical, use of the multiple valued hash table.

However; time and space required that I concentrate just on the principles and hope that this might ease the reader into other papers or books. Dorfman and Henderson have written the definitive on SAS hashing and it can be purchased from SAS press.

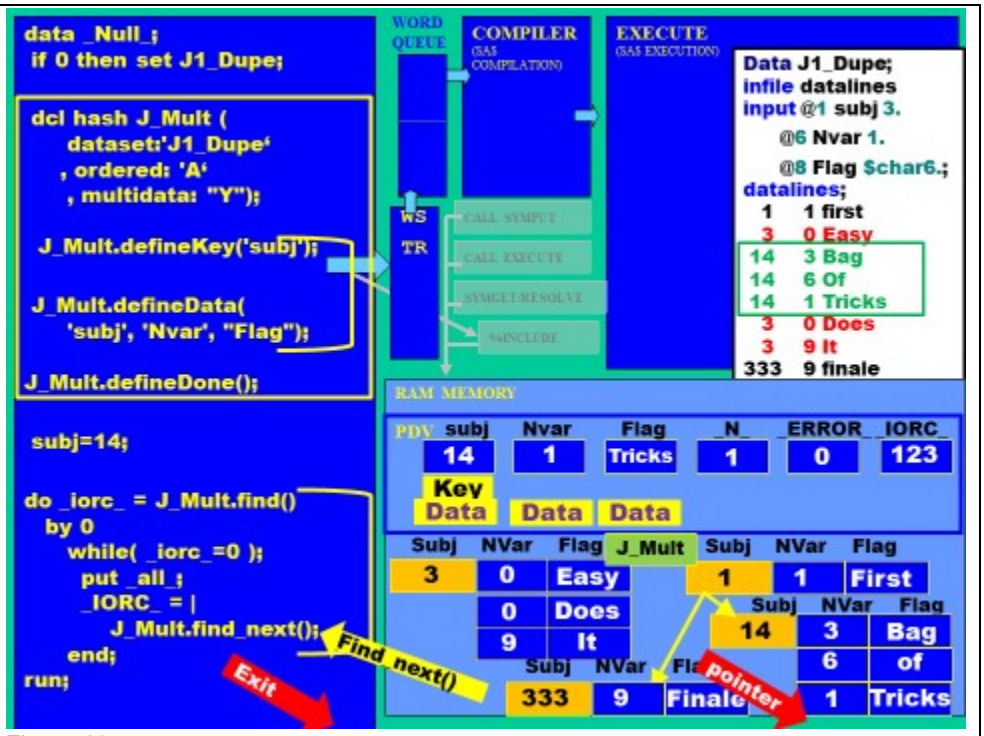


Figure 41

EXAMPLE 9: A HAS REPLACING A PROC SUMMARY – AND A DATA STEP

In the figure to right you can see code that allows a programmer to eliminate a PROC Summary by using a hash table.

As a 2nd process, inside the same data step, the hash table is used to apply logic that requires information from the summarization.
We will calculate some students percentage of their age inside their gender and use the numbers form the Hash table as the denominators.

I admit that this is a silly example and I cannot find a business reason for doing this particular example. However; it does show the power and programming flexibility of hash tables.

Section K) Hash Table replacing PROC Summary

```
Data J1_Cool;
 /*the has table will hold denominators and so is called hDen*/
 length age_sum 8; /*avoid messages in the log - this var is NOT in SASHelp.class*/
 if 0 then set SASHelp.class(keep=sex age); /*avoid messages in the log*/
   if _n_=1
   then do; /*set up the hash in memory - Define the hash*/
     dcl hash hDen( suminc:'age', hashexp:2 ); /*dcl statement has to be first*/

                Put variables on the PDV

    hDen.defineKey("sex"); /*define key:no define data - there is an internal accumulator*/
    hDen.defineData("sex", "age");
    hDen.defineDone();
  do while(^ EOF); /*load hash w/ sums of ages- calculate sums of ages by gender*/
    set sashelp.class end=EOF;
     /*REF method add values of sex to the hash &increments the INTERNAL accumulator.
      Ref looks for a key(sex) in the hash object. If key NOT found, key/data added to hash.*/
    hDen.ref();
  end;
      hDen.output(dataset: "J1_hash_sumAges");
end; /*if _n_=1*/
/*this is a normal SAS data read loop*/
set sashelp.class end=EOF_Class;
 hDen.sum(sum: age_sum);
 pct_of_Gndr_age= age/age_sum;
 output J1_cool;
if EOF_Class=1 then hDen.output(dataset: "J1_Summing_results"); run;
```

Figure 42

Please note the use of suminc: (immediately above the yellow box in Figure 42). After the suminc:, we must code a character string/expression which will be used as the name of a data step variable whose value is to be aggregated. This variable will be part of the hash table. It is not going to be part of the data section of a hash entry. Memory, behind the scenes (whatever that means) is reserved in which SAS will sum the values of the named variable.

Note these limitations on aggregating using the hash table.
First: Only one variable can be summed.
Second: Since this is being aggregated "behind the scenes" it cannot be accessed via "normal" SAS code. A method must be used to retrieve this value.

The figure to write shows how the code creates the PDv and the structure for the hash table.

You can think of the code inside the gold box as having 2 sections. The 1st section creates the hash table. The 2nd section, the do while, reads the file SASHelp.class into the hash table.

The code inside the gold block only executes one time.

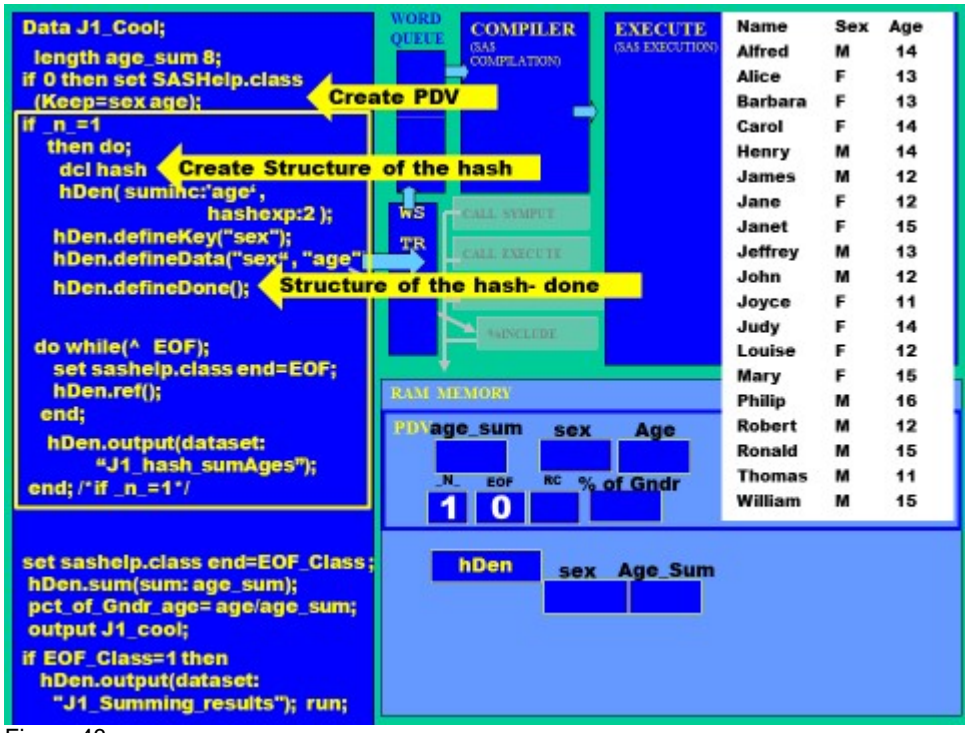


Figure 43

The figure to right shows the 1st observation flowing through the P DV and into the hash table.

Note the use of SumInc. as the table was defined.

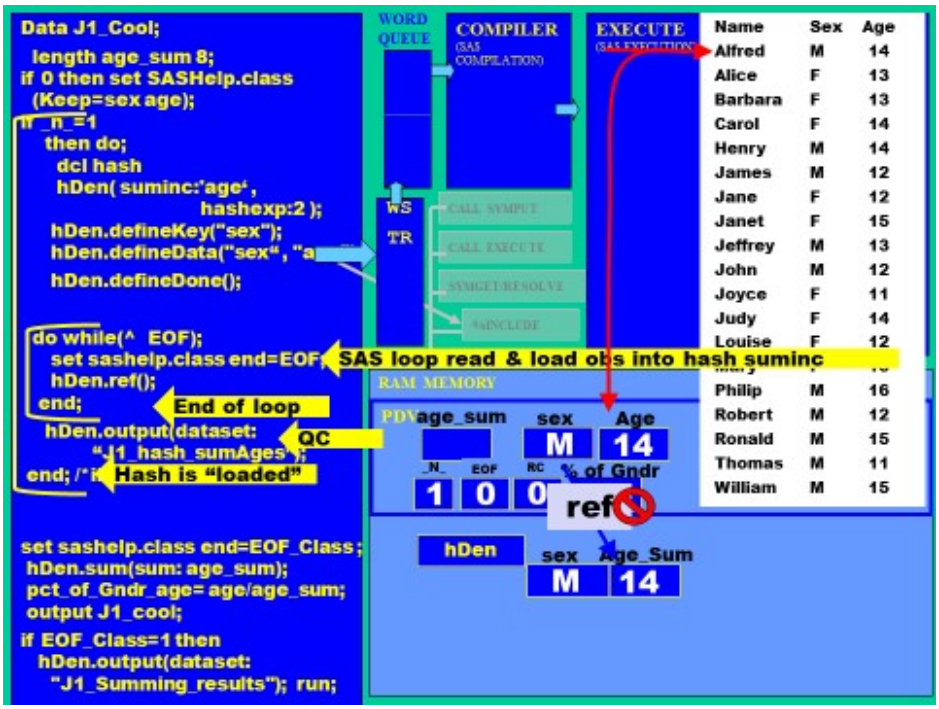


Figure 44

The figure to right shows how the 2nd observation, for a female, creates a 2nd entry in the hash table.

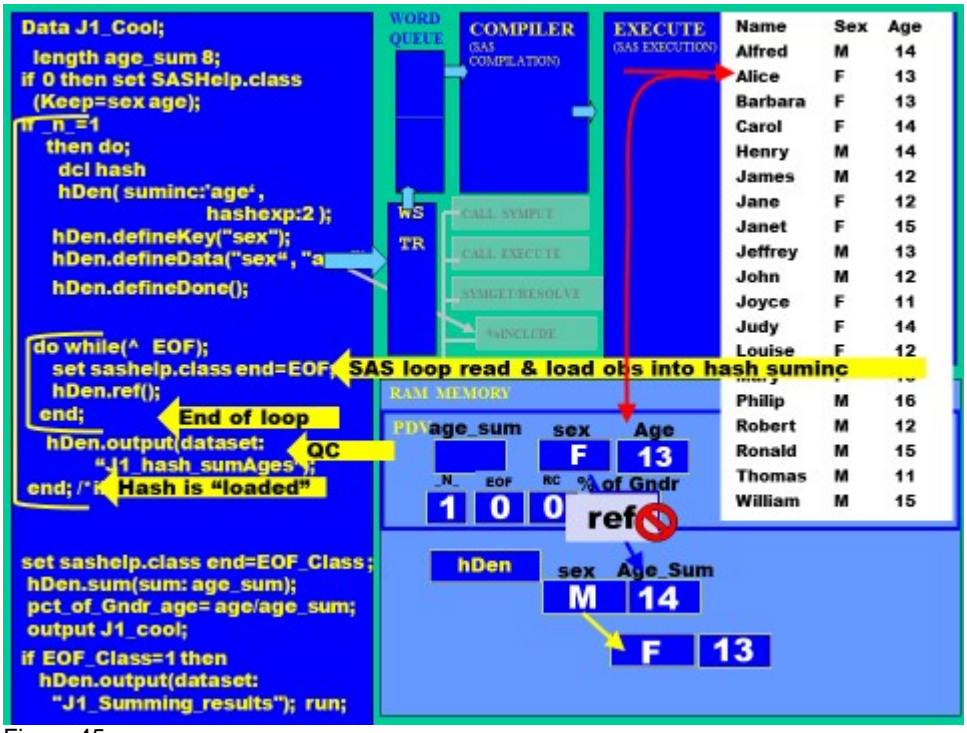


Figure 45

This figure to right jumps to the end of the loading process.

All of the rows in sass help.class have been processed and the hash object now contains the denominators we will need to calculate our percentages.

We are going to do a rather silly operation. We're going to calculate each persons age as a percentage of the total ages for their group.

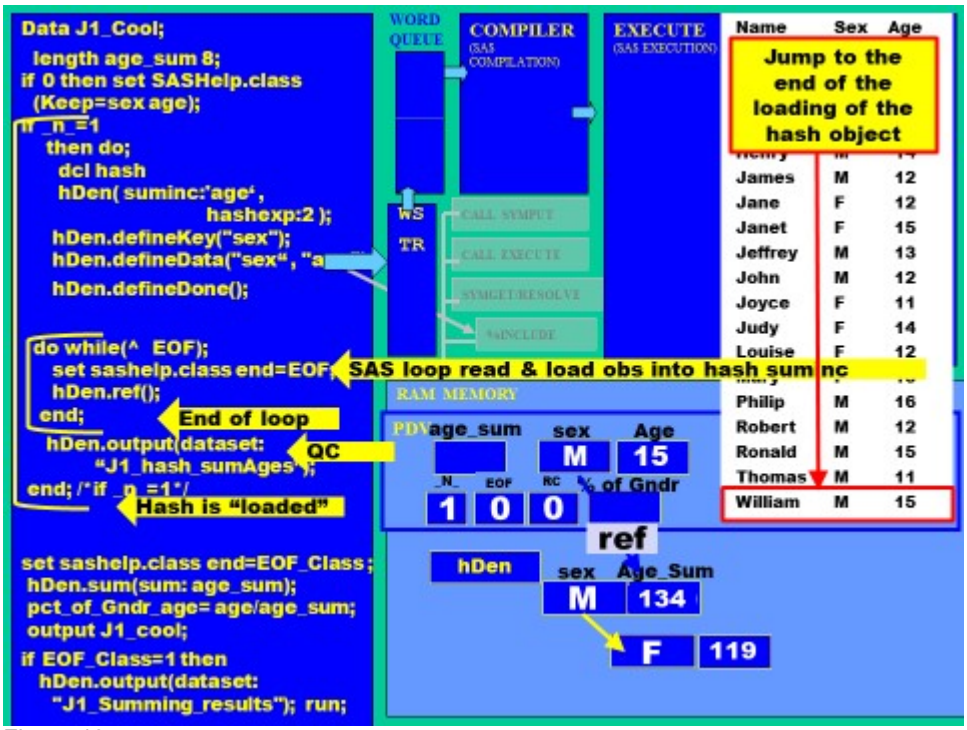


Figure 46

The figure to write shows the 1st use of the hash table to provide denominators for our calculation.

Alfred is a male that causes the recall of 134 to the variable age.sum. 14 is divided by 134 giving .104 and that number is output to the file named J1_cool.

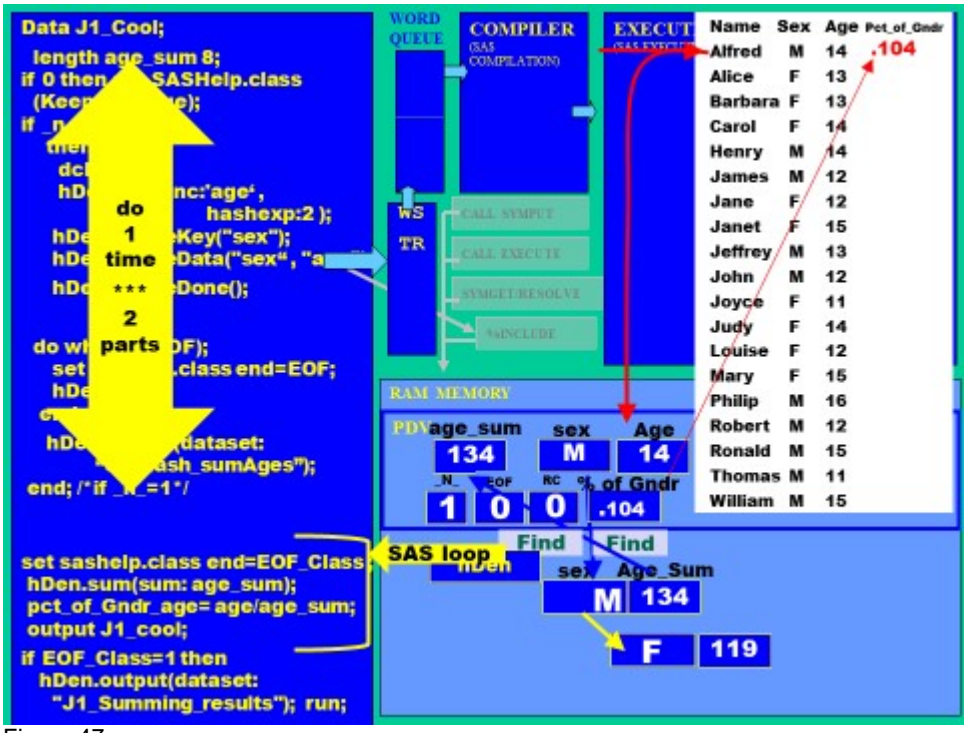


Figure 47

The figure to write shows the 3rd observation in SASHelp.class being processed

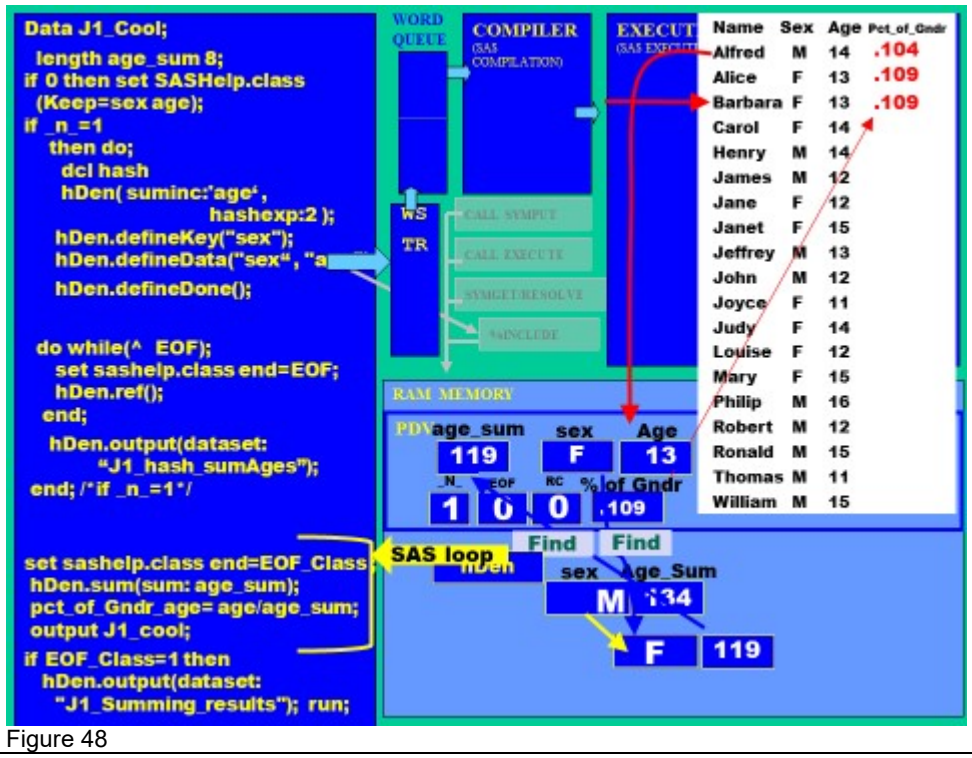


Figure 48

EXAMPLE 10: THE REPLACE STATEMENT

In this example we want to track a sales representatives activities. We want to build a variable called Hist that tells us the type of physician the salesperson visited upon and the date of the visit.

We are going to build a string that gets longer and longer and were going to limit that string to 90 characters by coding

```
length Hist $90;.
```

To right, please see all the code. Future figures will show details of execution.

This is another silly example. Often silly examples are hard to understand. I could not think of a new situation where this technique would be used to solve an important and common business need.

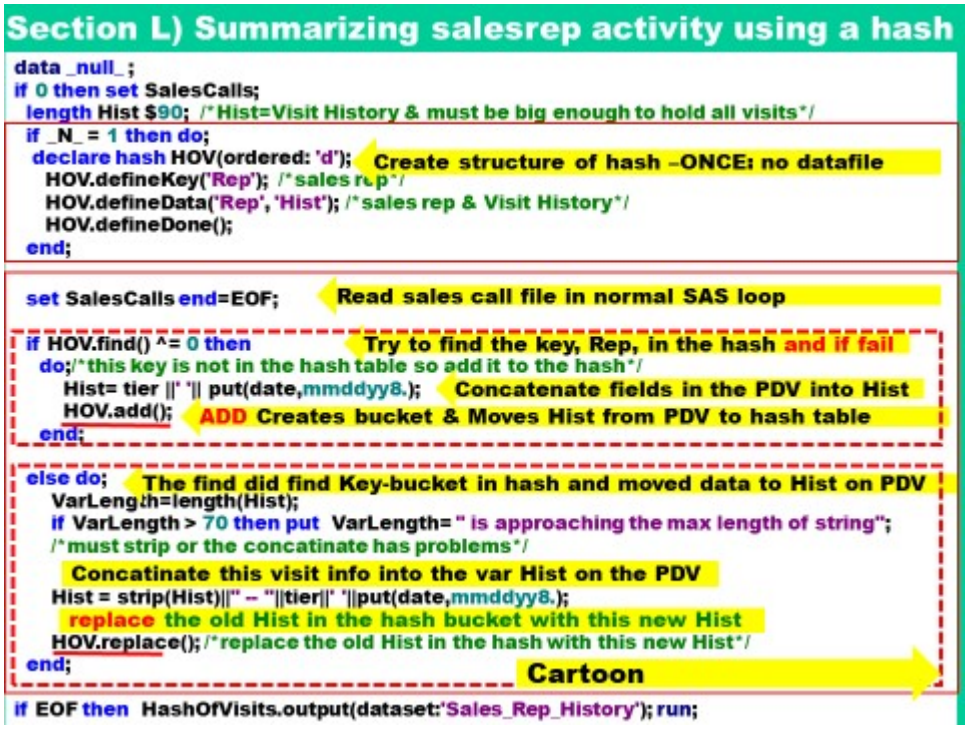


Figure 49

The figure to right shows the creation of the hash table and that we are reading the file SalesCalls.

HOV stands for History of Visits

If we do not find the rep in the hash table we create a variable called HIST on the program data vector and then move that variable (HOV.Add) into the hash table.

This figure illustrates what happens when a rep is not already in the hash table.

Tier and date are concatenated and then added to the hash table name HOV.

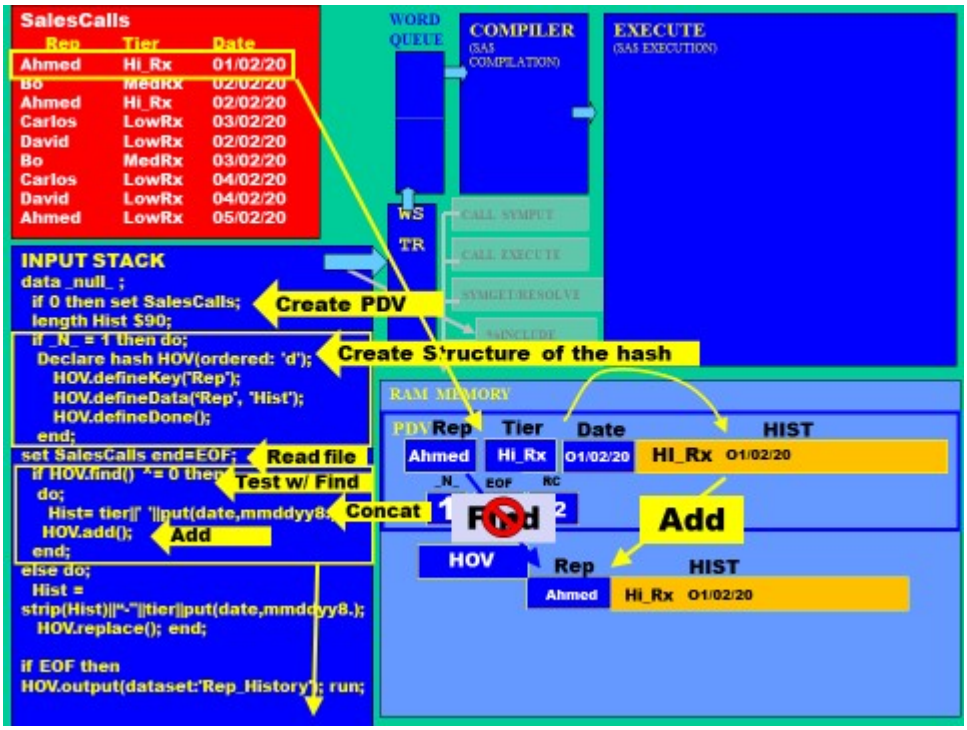


Figure 50

The figure to right shows a 2nd sales rep being added to the hash table.

Bo was not in the table.

Tier and date are concatenated and then an element is created for Bo.

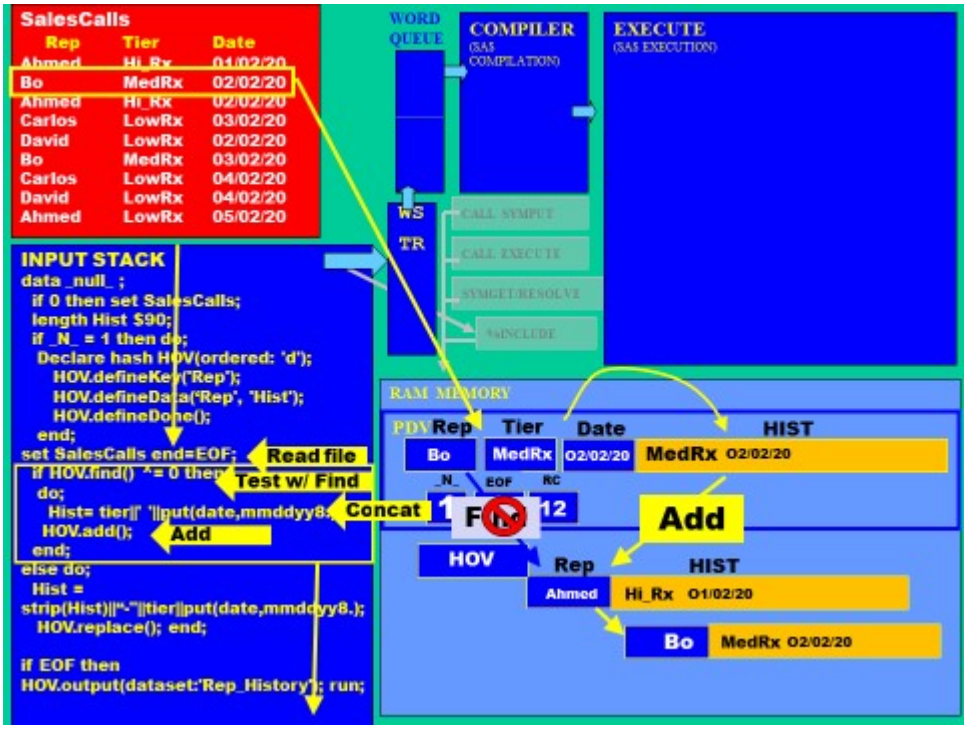


Figure 51

This figure illustrates the processing for the 2nd time we have seen Ahmed.

We need to recall Ahmed's information from the hash table into the PDV so that we can update it.

The find statement finds Ahmed and returns the data part of his element from the hash table to HIST in the PDV.

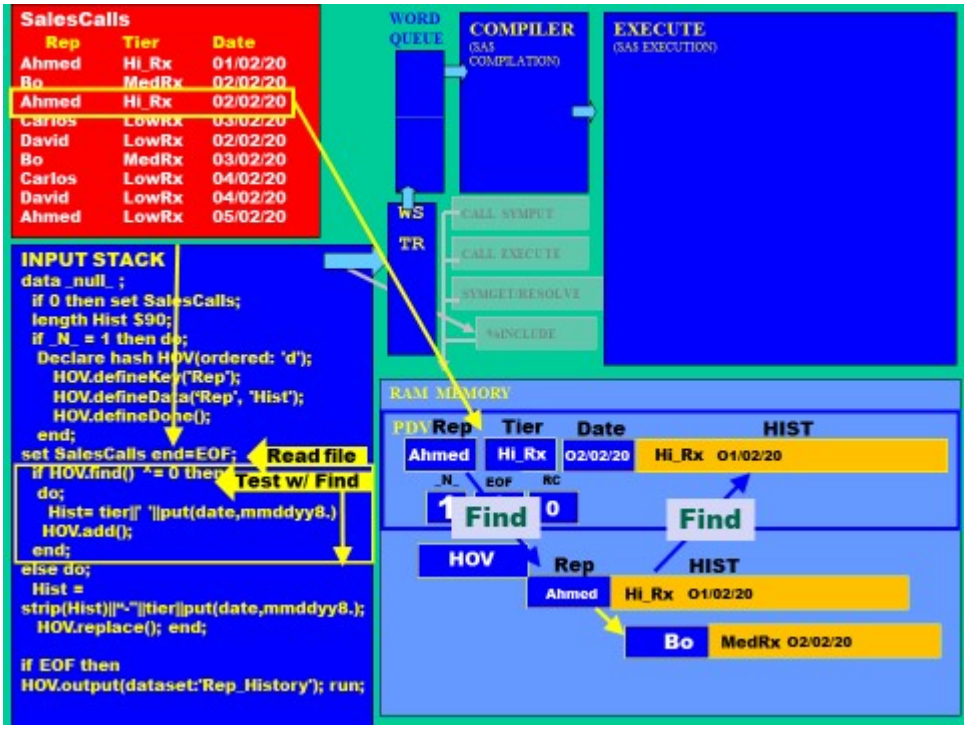


Figure 52

This figure shows the rest of the processing for Ahmed.

SAS strips blanks from the value in HIST and then concatenates information from the variables Tier and date on the PDV into a new value for HIST.

Then SAS uses the replace method to replace the Ahmed's old data in the hash table with the value of hist on the PDV.

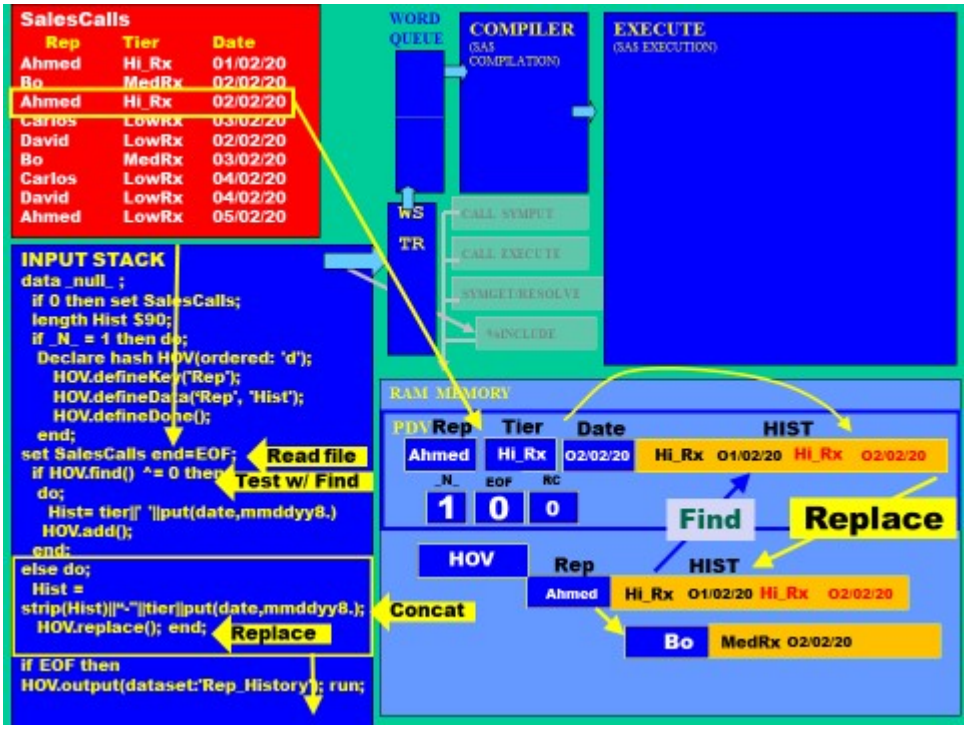


Figure 53

EXAMPLE 11: A HASH OF HASHES

I've shown many examples of hash tables without showing one important fact. When you create a hash table, SAS will create "invisible" variables on the PDV so that it can perform necessary operations.

Creating a hash of hashes requires some understanding of these "invisible" variables.

In the figure to right, I started the debugger (if you want to learn more about the debugger, google "animated lavery debugger SAS") so that I could look at variables on the PDV.

The left-hand side of the figure shows the code I wrote. Note the 2 colored underlinings I added for this figure.

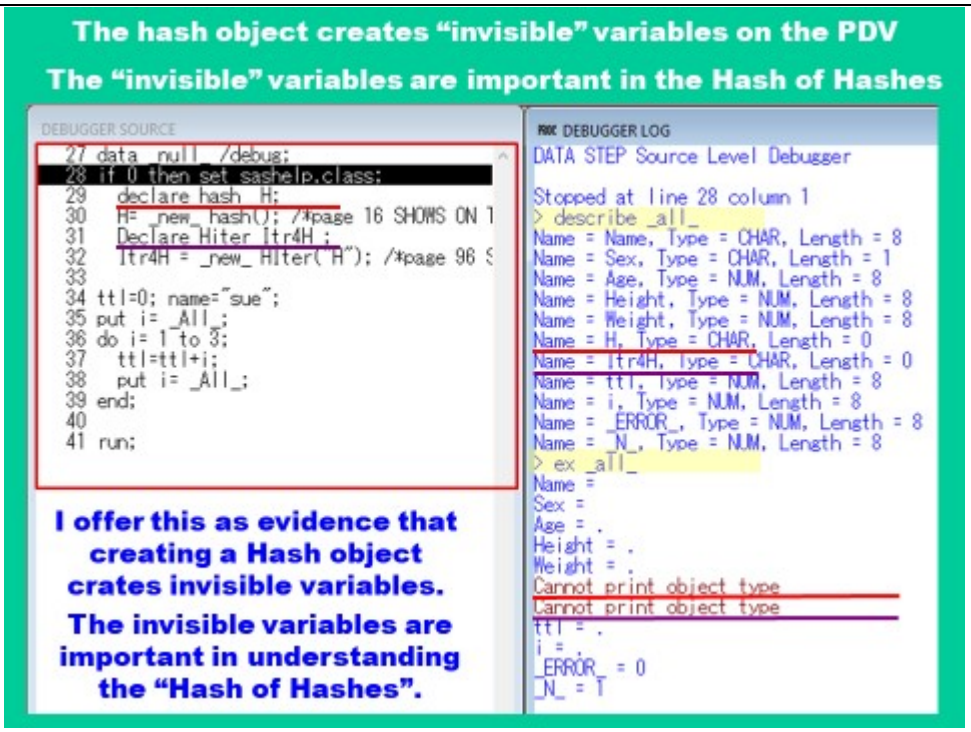


Figure 54

On the right-hand side of the figure you can see the output from the data step debugger. I asked the debugger to describe all the variables on the PDV. It found all the variables including the variable named age, which is the hash object I created, and the variable Itr4H which is the hash iterator method associated with the hash table named H. Both of these are defined as character with length 0. They are different from "regular SAS variables and are really pointers to objects.

The hash table, and the hash iterator, are objects (you might have heard about object oriented programming) and these two objects exist in RAM independent of the PDV. The way SAS will communicate with these objects is through these two variables, H and Itr4H, on the PDV. These two variables will contain memory pointers (the memory locations in RAM) for the two objects.

Since these variables are pointers to places in RAM, they are not considered character or numeric variables and things like the Debugger, developed long ago, has a bit of trouble dealing with them. Because these variables are intended to be invisible, SAS just says they are character of length 0. I guess that SAS figures that anybody who knows enough to start up the debugger and ask for these variables has some idea of what they really are. In summary, the debugger can find these variables on the PDV.

In the bottom half of the right-hand side I ask SAS to print out ***the values*** of all the variables on the PDV. The debugger says it cannot print object type variables. It can only print character and numeric variables. In summary; the debugger can find these variables on the PDV, but cannot print their values.

I offer Figure 54 as support for my statement that the PDV has invisible variables that are used to control hash tables.

The figure to right shows the code that we are going to use in this example.

The code will do no useful work and only illustrates the creation of a hash of hashes.

A hash of hashes describes the situation where a hash table entry is a pointer to another hash table (plus some coded logic to support use).

Because this code creates a RAM resident data storage it can, in the hands a creative programmer, make for fast execution of business logic.

Remember accessing data in RAM is typically thought of as being 20,000 times faster than accessing data on disk.

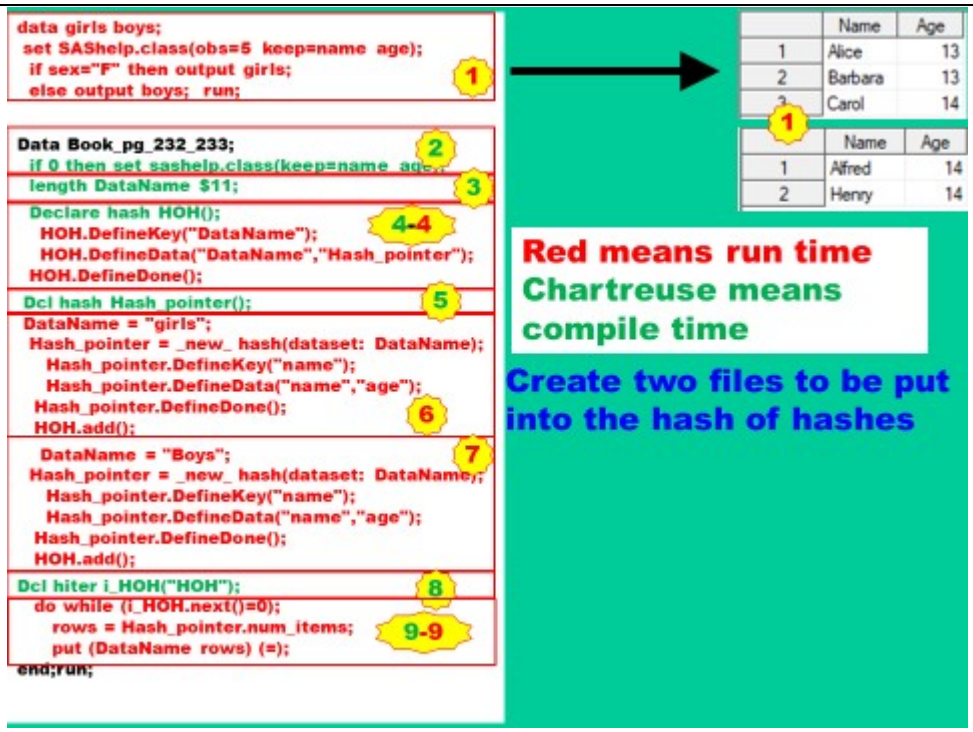


Figure 55

Part of the difficulty in understanding a hash of hashes is the execution sequence is unusual. Some of the commands are intended for the SAS compiler and some of the commands are intended for SAS execution. Some of the commands are intended for both SAS compile and execution.

The figure above shows step one executing and creating two files. The rest of the code will load these 2 files into hash objects and then, simply, query them for some characteristics (number of rows).

The compile actions are in the green—ish color above.
Item 2 creates the PDV from the header information associated with a table SASHelp.class.
Item 3 adds a variable. DataName, to the PDV.
Item 4 adds HOH to the PDB and, I think, creates a SAS table placeholder at the RAM address loaded into the PDV.
Item 5 adds Hash_pointer to the PDV and, I think, creates a placeholder at some address in RAM.
Item 8 adds a variable calledI_HOH to the PDV and creates that object in RAM. I_HOH contains a pointer to the memory address for that hash object.

The run actions are in red.
Item 4 establishes the structure (both key and data) of the hash object named HOH and establishes links between the object and the PDV.
Item 6 adds a value to the variable DataName on the PDV and that value is “girls”.
new is a runtime command that creates a new hash object. It gets the data for this new hash object from the data set girls.

The last statement in section 6 (`HOH.Add();`), takes variables from the PDV and uses them to create an entry in that hash object named HOH.

Item 7 adds a value to the variable DataName on the PDV and that value is “girls”.
new is a runtime command that creates a new hash object. It gets the data for this new hash object from the data set girls.
The last statement in section 6, takes variables from the PDV and uses them to create an entry in that hash object named HOH.

I suggest that the explanation, while necessary, is pretty confusing and will try and illustrate the steps involved in a hash of hashes in the figures to follow.

This figure illustrates the compile actions.
`If 0 then set SAShelp.class`
is a trick to create variables on the program data vector.
The length statement also is used to create a variable on the program data vector. The next statement tells SAS to create a variable on the PDV.

The pink code in this slide is runtime code and, honestly, the exact timing of this eludes me. When the pink code is run it will create the hash object named HOH.

I think the green, at run time, might assign the memory location to that hash object – but I’m not really sure. On this slide I show step 4 also creating a spot in memory that will hold the hash object HOH.

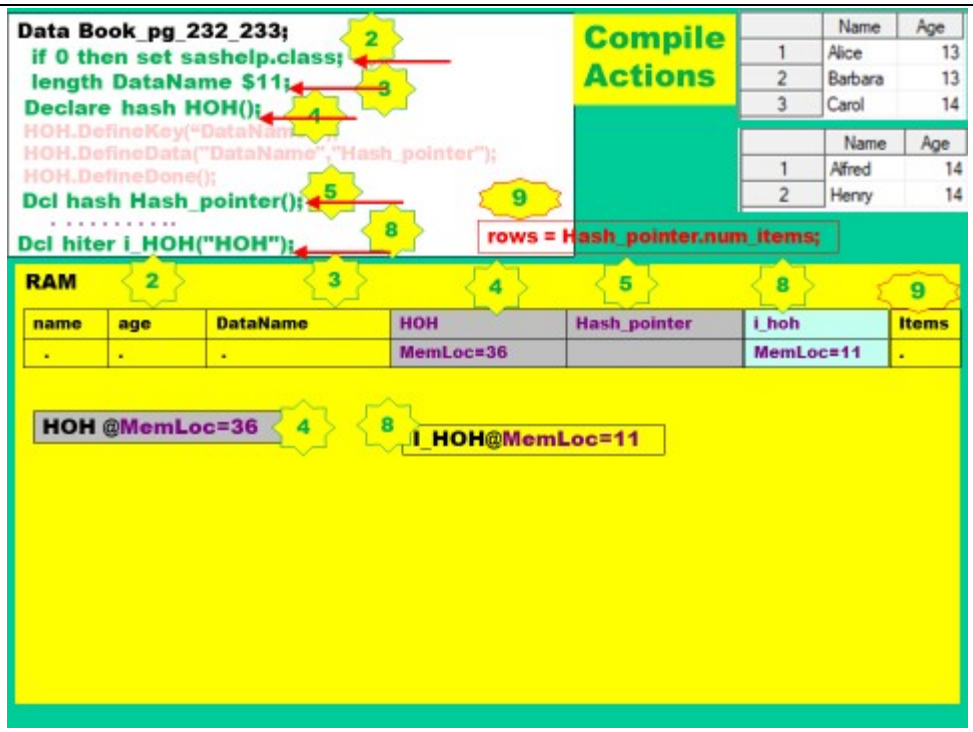


Figure 56

Star 5 illustrates the effect of declaring a hash object with the name of Hash_pointer. This creates a spot on the PDV, but because of the coding done, does not create a memory location. We are going to create a hash object called girls and another hash object called boys. They will exist in memory. Hash_pointer will be used to point to those memory locations. Loading different memory locations into the variable named Hash_pointer will allow us to point to different hash objects.

Star 8 shows where SAS creates a variable on the PDV that will be used to control the iterator for the hash object named HOH. The iterator is a subroutine that is created to allow us to access, sequentially, elements of the hash table. A hash iterator is an object that resides in memory and the variable i_hoh, on the PDV, points to the memory location for the iterator. Having a memory location on the PDV allows SAS to access that iterator. Think of the iterator as being a subroutine and this tells SAS where the subroutine is resident in memory.

This slide starts to show the run time actions involved in a hash of hashes.

When star 4 executes it tells the hash object named HOH that it should get key information from a variable on the PDV with the name of DataName and it should exchange information about the data values in the hash object called HOH with variables called DataName, HOH and HashPointer.

I show the variable HOH taking the value “MemLoc=36 to indicate that it is a pointer and it’s actually pointing to the spot in memory where SAS has placed the hash object named HOH.

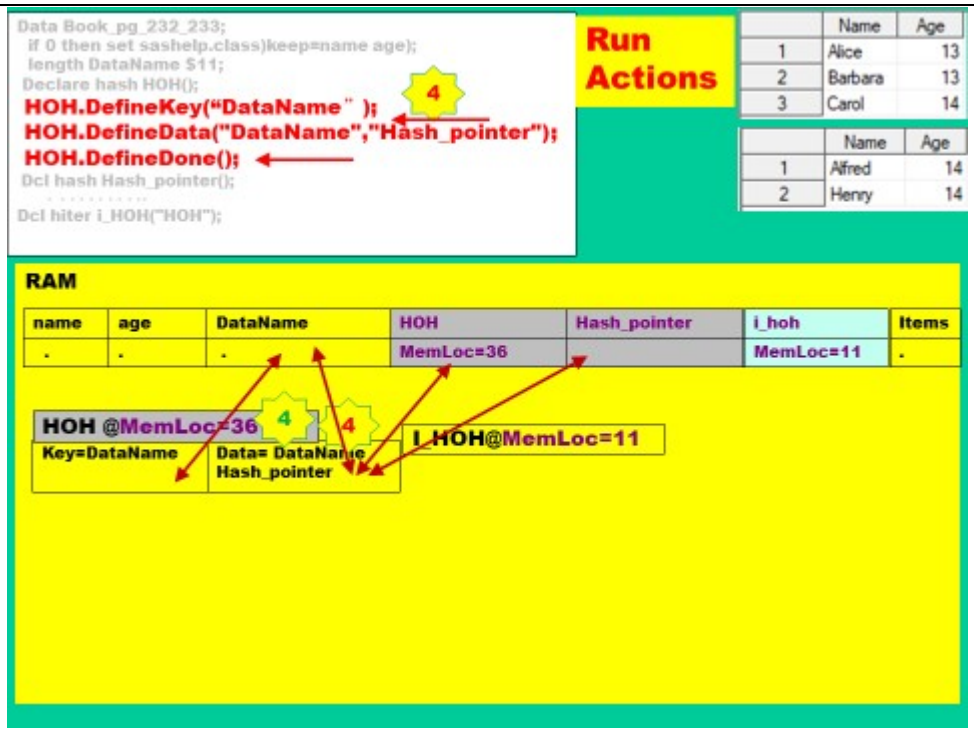


Figure 57

This slide shows the run time execution of star 6 on Figure 55. Figure 55 shows all the code.

DataName is assigned a value girls.

New is a run time command that creates a hash object while code is executing (not compiling). It creates a hash object with the name of girls at a position defined by SAS (here MEMLOC=500).

DefineKey and DefineData establish, during run time, the relationship between variables on the PDV and characteristics of the hash object (key and data).

Note that the HOH.add stores, in HOH, the address in memory of the hash table named girls.

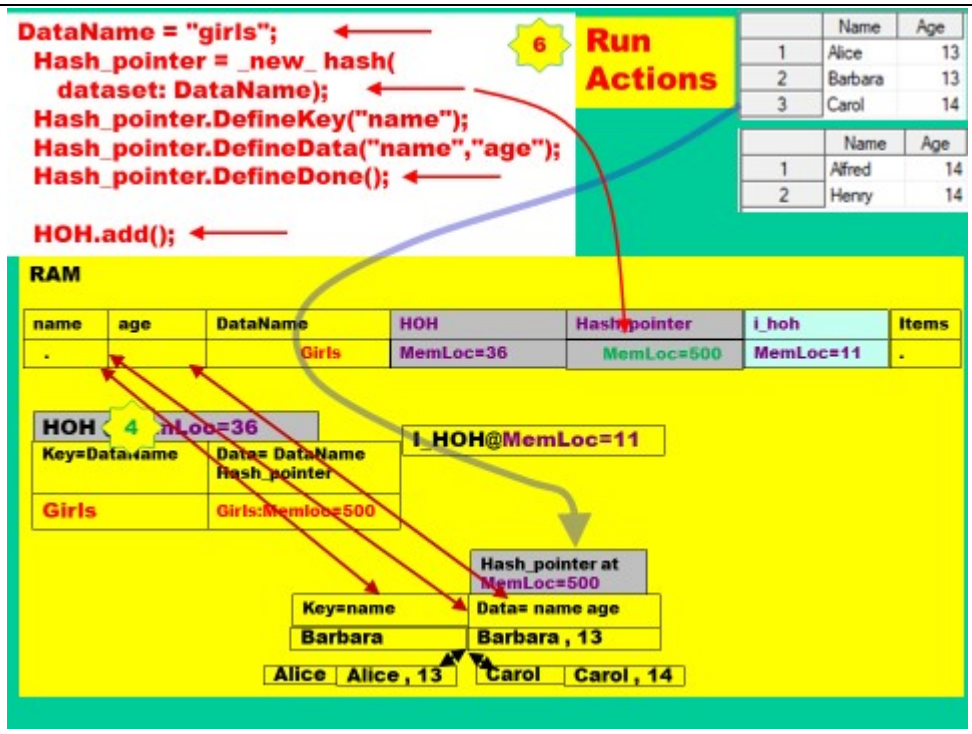


Figure 58

This slide shows the run time execution of star 7 on Figure 55.

Data name is assigned a value boys.

New is a run time command that creates a hash object while code is executing (not compiling).
It creates a hash object with the name of boys at a position defined by SAS (here MEMLOC=200).

DefineKey and DefinedData establish, during run time, the relationship between variables on the PDV and characteristics of the hash object (key and data).

Note that the HOH.add stores, in HOH, the address in memory of the hash table boys.

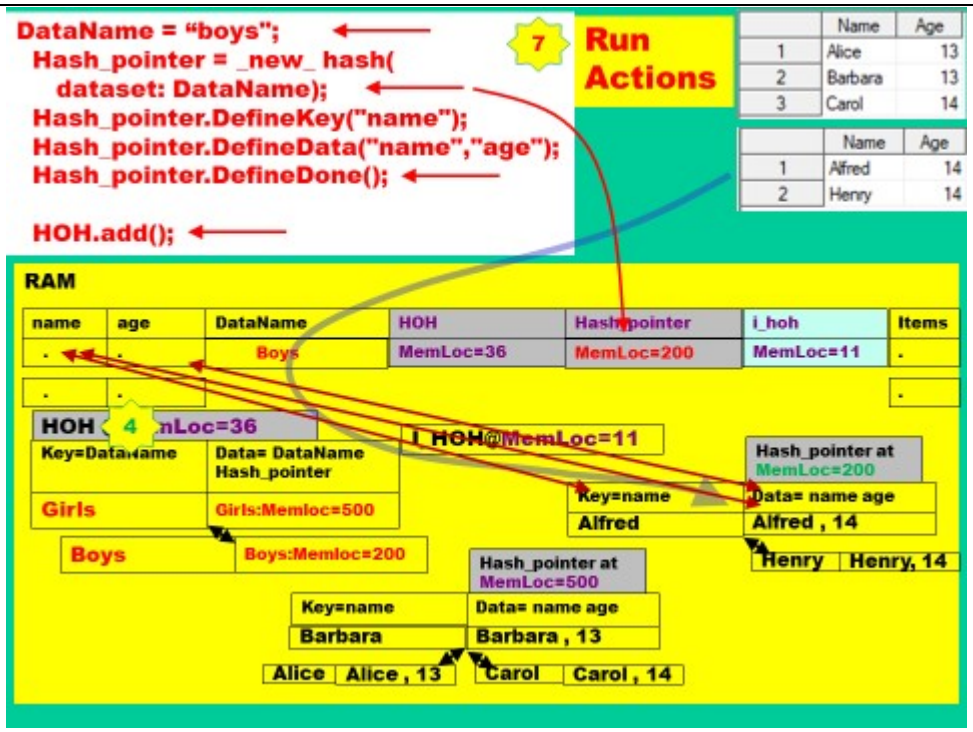


Figure 59

The iterator object, named I_HOH, executes in a while loop.

The next method will position, initially to the 1st element of the hash object so it finds, in HOH, the element for girls.

It returns the memory location for girls to the hash_pointer on the PDV this makes girls, at memory location 500, the "active" hash object.

SAS executes the method NUM items (which is a little subroutine that returns the number of items in a hash object) against the hash object that's located in memory position 500.

Girls has 3 elements and so a 3 is stored in the variable items on the program data vector.

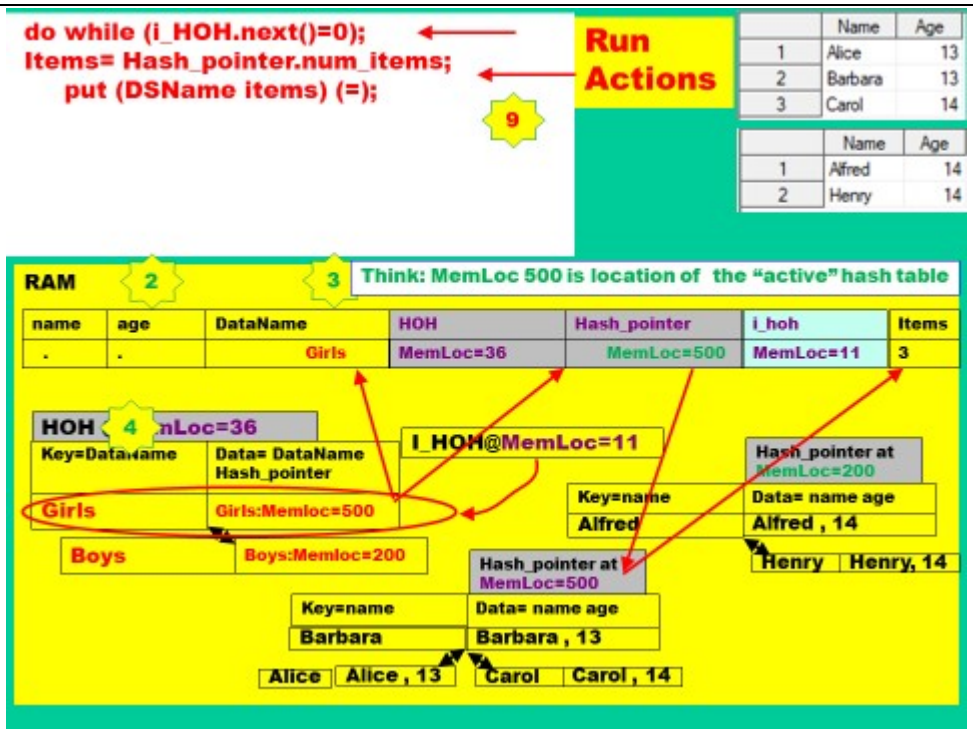


Figure 60

When `next()` next executes, it returns the elements in HOH from the element with the key boys.

This returns the data part of the boys element in HOH to the program data vector. The value of hash_pointer is now 200.

You can think of the fact that the value of hash pointer is now 200 has as making boys the active hash object for all subsequent actions.

SAS now runs the subroutine NUM items against the hash object whose location can be found in the PDV in the variable named hash_pointer.

This returns a 2 to the variable items. The loop would escape on the next attempt to read from the hash object HOH.

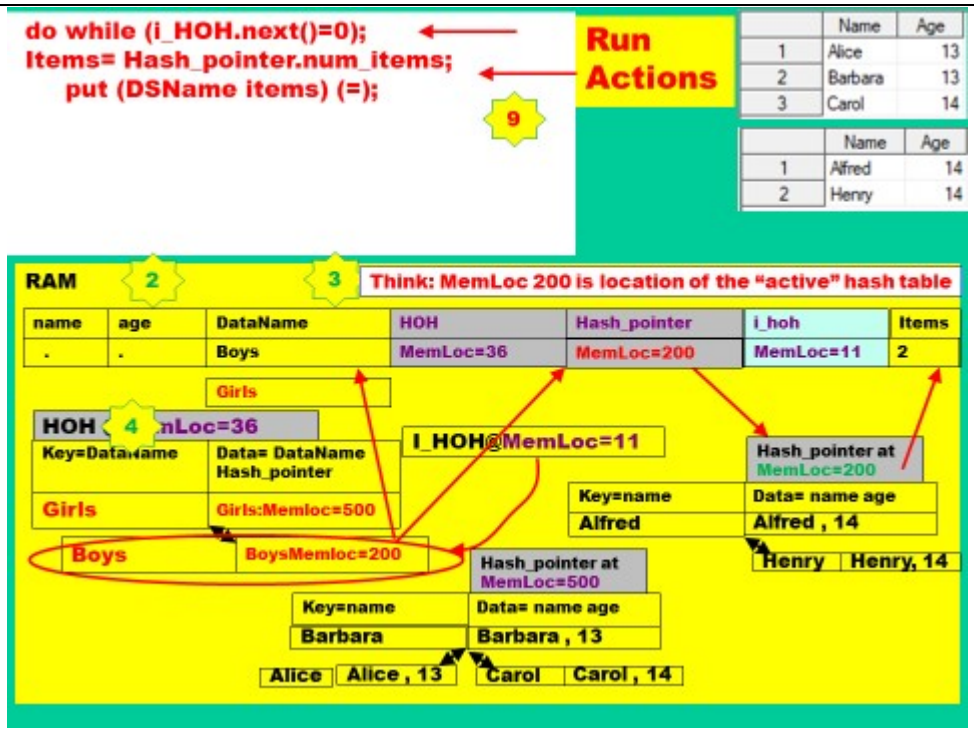


Figure 61

CONCLUSION

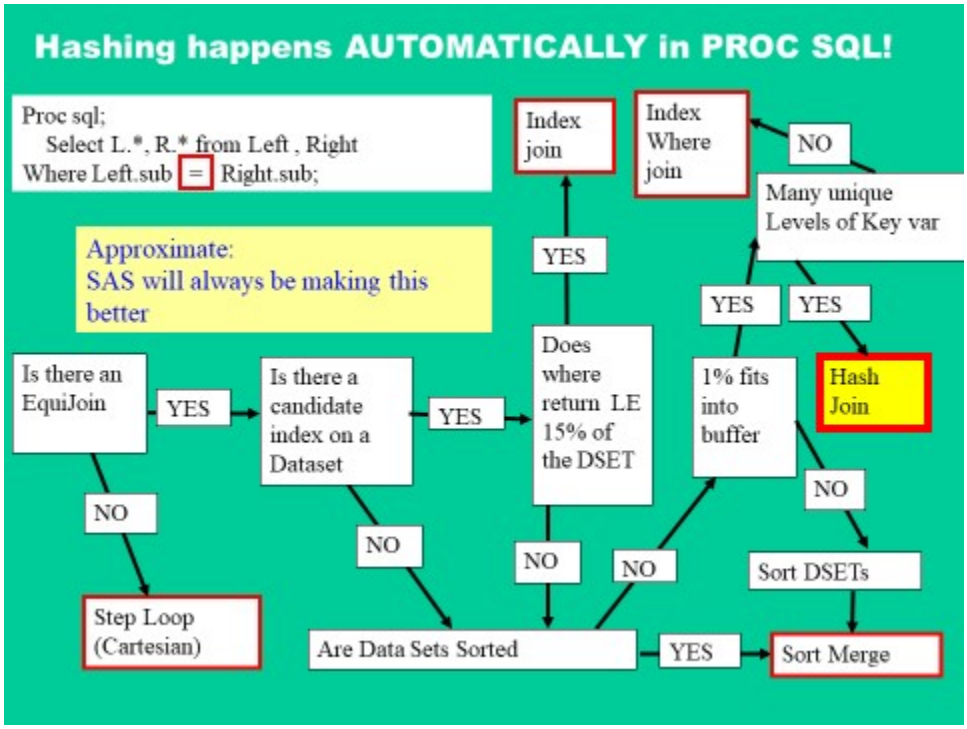
This is a time of great competitive change. We need to make our SAS programs faster to so that our companies will continue to use SAS and SAS programmers. We need to learn the new techniques in order to allow ourselves, and the programs we write, to compete in the IT world. We need to provide value to our customers and hashing can allow us to write faster running programs and provide more value to clients.

Thanks to Paul Dorfman, Don Henderson, Richard DeVenezia , Robert Ray and Jason Secosky.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.:
Russell Lavery, Independent Contractor for Numeric Resources
Email: russ.lavery@verizon.net

Appendix:



Above is an approximate (very dated) graphic of how the PROC SQL optimizer decides if it should use a hash table in a join. For more information please see The SQL Optimizer Project: _Method and _Tree in SAS®9.1 (https://www.lexjansen.com/phuse/2007/cs/CS11.pdf) This is a fifty page paper and I do not think it is worth studying. However the first 8 pages, where an overview of the optimizer is described, might be of interesting.