Paper ###-2019 (if SAS author, use SAS###-2019 or SAS####-2019)

A HOW on Hashing

Russ Lavery

ABSTRACT

Hash tables are underutilized, and very fast, techniques developed for table lookup – that now do much more. A key feature of has use is the avoidance of CPU and RAM intensive sorting of data during table "merges". Understanding the behavior, and even the "shape", of this memory resident object can be a bit of a challenge to the typical SAS programmer. This paper, with its cartoon format, endeavors to make this new programming technique more approachable.

Hash tables use a new syntax format that is similar to the object oriented syntax of other languages. Hash tables are created in a data step and exist only as long as that data step is running. It exists in RAM and are separate from the program data vector.

It should be remembered that SAS has hash tables due to the work of Dr. Paul Dorfman and the responsiveness of the people at SAS to user input.

INTRODUCTION

This paper is divided into the following sections:

1) what is table lookup and why learn hashing
2) check your system
3) table lookup via assorted by merge
4) numeric table lookup using format
5) character table lookup using format
6) IORC as table lookup
7) key indexing as table lookup
8) why hashing is fast (versus a format lookup)
9) hash table with variable creation on the PDV failure
10) hash table with uninitialized note:
11) successful creation of a hash table
12) hash table lookup – the original purpose of the hash table
13) duplicate keys in a hash table
    hash table creation fail – no use of multi-data
    hash table creation success
    hash table traverse failure (use of find instead of find_next)
    hash table traverse success
14) hash table replacing a PROC Summary and emerge
15) hash table summarizing sales rep activity
16) using a hash table as part of a three way join

1) WHAT IS TABLE LOOKUP AND WHY LEARN HASHING

Table lookup is what we do when we don't want to type out 1000 lines of if statements. If we needed to get the state abbreviations to match a bunch of ZIP Codes we could code an if statement as is seen below:

- If zip="19103" then state="PA";
  Else if zip="08535" then state="NJ";
  Else state="??";

The problem with this is that there are about 40,000 ZIP Codes in America and that takes a lot of typing.

The most common way to do table lookup is our old friend the sorted by merge and "lookup the value of state in another table" as seen below:.

```
data A3_Cool_zips_by_merge;
  merge A1_Get_these_zips(in=G) /*smaller file*/
        A2_zipcode; /*larger master data file*/
  by zip;
  if G=1 ;
run;
```

The problem with this merge is that merging requires sorting the files to be merged. While SAS has, at least, one patent on sorting and sorts very fast, it is still true that sorting is a very expensive operation. (Pause to think about that statement for a while. Sorting's been around for a long time and a lot of really smart people have worked on sorting, so to get a patent on sorting is a pretty impressive thing. Even though the SAS sort is very fast, sorting takes a lot of CPU cycles and can involve slow disc access because the "hidden" system files that SAS creates to do a sort can be 2 to 4 times the size of the file being sorted.

So, while a sorted by merge is commonly done it is something to be avoided if the size of the files "stress" your computer system.

Figure 1 shows why we want to use hashing for table lookups (and now for other techniques as well).

Before hashing, the fastest way to do a table lookup was with a format statement. While the time advantage of hashing over format table lookup varies with the size of the file and the percentage of the small file that is present in the big file – the bottom row in the chart in Figure 1 shows the dramatic advantage (3.9%) of hash tables over format table lookups.

Please note that there is one technique that is faster than a hash table and we will discuss it in this paper.

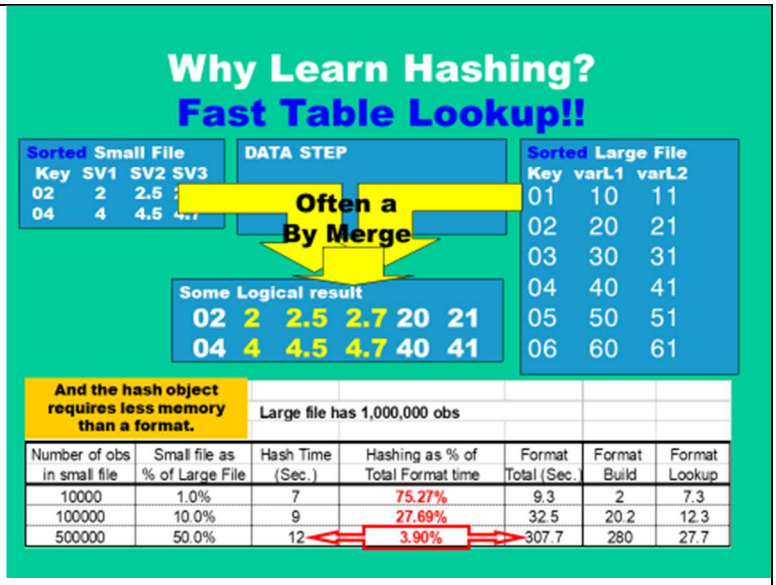

Figure 1

This paper, and the HOW that it supports, will concentrate on hash table lookups but will also review some of the older competitors to this technique.

2) CHECK YOUR SYSTEM

As we will see below, the hash table grows dynamically as you feed data into it. It is not like an array which needs to have its size defined when the array is created. You can fit as much data into a hash table as your RAM allows. Since the hash table must reside in RAM, it makes sense to check RAM once.

The default install setting for SAS is to allocate two gig of RAM to SAS. Many people, in the days when SAS was commonly installed on a PC, would upgrade their RAM and not change the setting in config.sys – negating the effect of the extra RAM. It is common for SAS on servers to still be limited to two gig. Use the code below to check how much RAM you have available.

```
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;
```

Hash tables are very memory efficient and checking hash size is usually only done by admin type people dong preemptive checks on routinely run jobs that use large files. Most people, in practice, don't estimate the size of a hash table before they use it. People just "do it" and see if the Data Step runs. In common practice, only when a Data Step Crashes, with an out of memory message, does a programmer investigate the memory needed for their hash table.

To get a rough estimate of the size taken by the table itself, not counting supporting methods etc., load a single record into the hash table. Use the item size attribute to get the size of one "bucket" (a term to be defined later) and multiply that by the number of keys that you plan on storing in the hash table.

3) TABLE LOOKUP VIA A SORTED BY MERGE

A sorted by merge was shown , in Figure 1, and should be familiar to all. It is a very powerful technique and simple to code. A lot of the simplicity comes from SAS's automatic looping through the data step. Examples, later in this paper, will combine a hash table with the automatic looping in a data step to do some interesting things.

4) NUMERIC TABLE LOOKUP USING FORMAT

Figure 2 shows all of the steps involved in a table lookup using a numeric format. It has a trick that allows a programmer to create a format without typing – to programmatically create a format.

Using a program to create a format is very useful when the programmer is given an Excel spreadsheet, with a few thousand ID numbers, and told to pull all records from some "master file" for the IDs in the spreadsheet. In this situation, creating a format by hand would be tedious.

The small data (1) set is read into a data step (2) this data step creates a SAS file that looks like a "bare-bones" file created by a PROC Format.

Note that neither file used in Figure 2 is sorted.

A PROC Format (4) with the cntlin is used to read the table created in (2) into a PROC Format. This step creates an entry in the format catalog based on the logic coded in (2).

The final step in the process is to read the “large” file from top to bottom and apply the format (5) to each row in the large file.

While Figure 2 is a little complicated it does show all of the steps and, therefore, can be a useful tool.

It should be noted that you can only bring one field through from the small file.

Here, the programmer could bring the “yes” through the merge into the final data set. This is a limitation of a format table lookup.

For more detail google: An Animated Guide: Power Merges: The format table lookup.

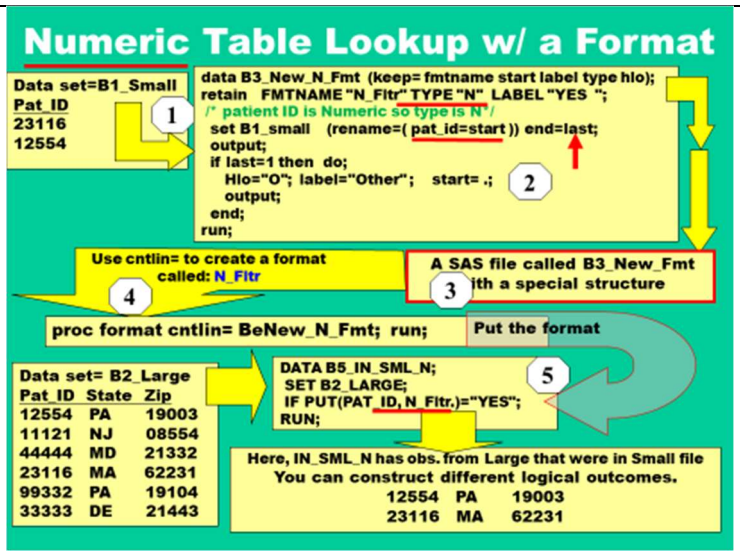

Figure 2

5) CHARACTER TABLE LOOKUP USING FORMAT

Figure 3 shows a picture of a character format being created and used in a table lookup.

There is very little difference between a numeric format table lookup and a character format table lookup. The processes are very similar and a reader should concentrate on the code that’s above the red lines.

Both are commonly encountered in the maintenance of older code.

It should be noted that you can only bring one field through from the small file.

Here, the programmer could bring the “yes” through the merge into the final data set. This is a limitation of a format table lookup.

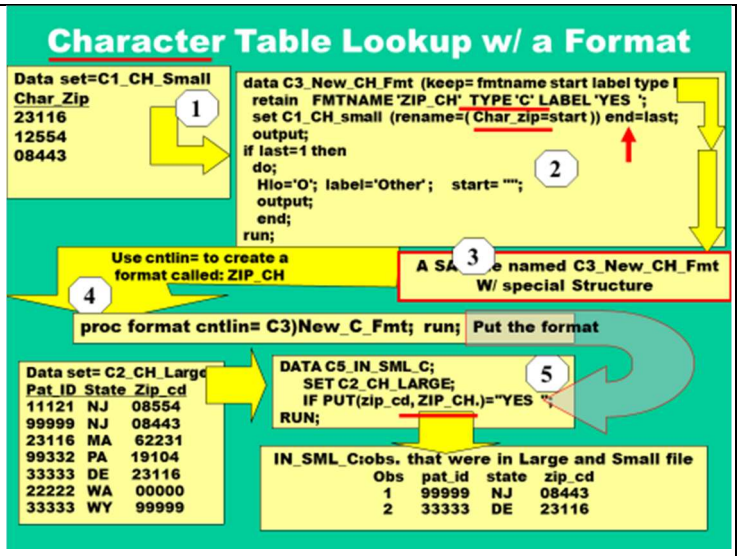

Figure 3

Note that neither of the two files used in Figure 3 is sorted.

It should be remembered that a format table lookup **was** the fastest commonly applied technique until hashing was introduced in SAS 9.2.

These two kinds of format table lookup have, generally, been replaced by hash table techniques.

There is, and always was, a technique that’s faster than a format table lookup or hashing– however, it has a more limited use case than a format table lookup. We will see this in a future section.

6) IORC TABLE LOOKUP

Another technique for doing a table lookup, without having to sort either of the files, involves building an index and using what is called an IORC (Input Output Return Code) table lookup. An IORC merge allows a programmer to bring many variables, from the small file, into the results of the table lookup.

Unfortunately, this technique cannot be shown on one PowerPoint slide and, accordingly, will be given very light coverage in this paper and in the HOW that this paper supports.

While code illustrating this technique is included in the SAS file that is associated with this HOW, this technique has a lot of moving parts and is best illustrated by a long series of PowerPoint slides. Explaining this technique requires, and is worthy of, a paper in itself. An interesting reader can search for the title below.

An Animated Guide: Speed Merges with Key Merging and the _IORC_ Variable

This technique has, generally, been replaced by hash table techniques.

7) KEY INDEXING AS TABLE LOOKUP

This technique is the fastest table lookup, not just in SAS, but in IT. Its speed, comes from the fact that it asks the computer to do very few things – and the things the computer must do are things it does well and quickly.

Key indexing, another technique introduced by Dr. Dorfman, is still in use when a programmer is lucky enough to be “merging” by an integer numeric variable.

A key index can only bring one variable through the merge and that is a real disadvantage in many situations.

Key indexing is an array technique. The key to the speed of this technique is that the two files are being” merged” by an integer numeric variable.

This technique uses values in the merging variable as the “cell number” for the array.

A series of pictures might be useful.

Figure 4 shows an array being created. The array name is _Keyindex and it is a very large array.

Make the array temporary because SAS stores temporary arrays in a contiguous memory space. This means that the storage and retrieval will be fastest.

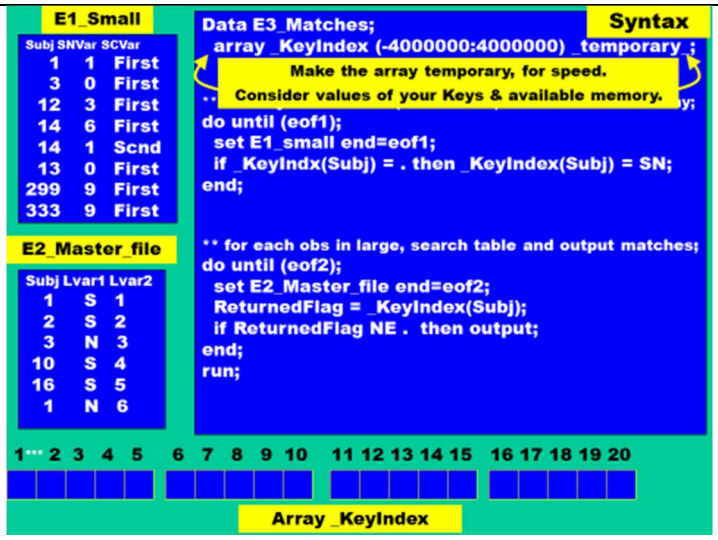

Figure 4

This technique proceeds in two steps.

The first step is a do until loop that loads the contents of small into the array. Note that the subject ID identifies the cell number in which the information for that ID will be stored.

The information for subject 1 is stored in cell number 1 of the array. The information for subject 12 is stored in cell number 12 of the array.

As you can see in the picture, this can be very memory intensive if the series of subject numbers have large gaps.

In Figure 5, the cells between 15 and 298 are not used. It is this wasteful consumption of RAM that spurred the invention of the hash.

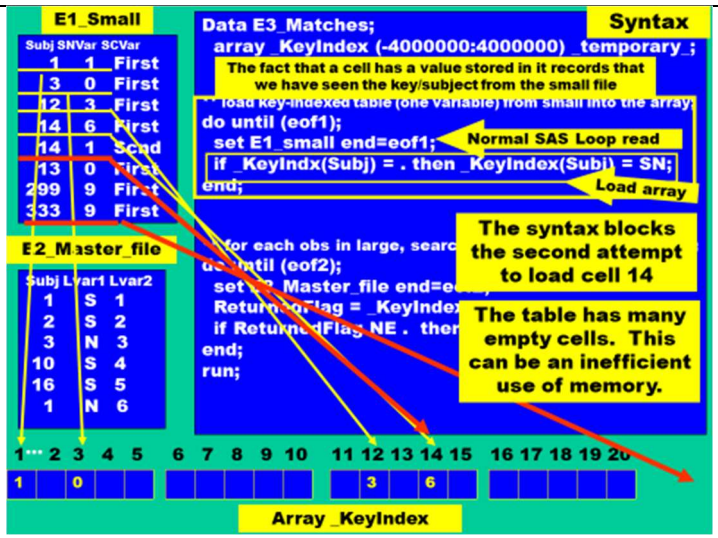

Figure 5

Figure 6 illustrates the second part of this technique.

In another due until loop, the larger file is read.

The subject ID in the larger file is used to identify a cell in the array. If that cell in the array is valued, then that subject ID must have been in the small file.

The logic of what to do, when one knows whether there is a "match" between subject ID in the small file and subject ID and a large file really depends on the situation that the programmer is encountering. She could desire a left join an inner join or a right join and all can be coded using this technique.

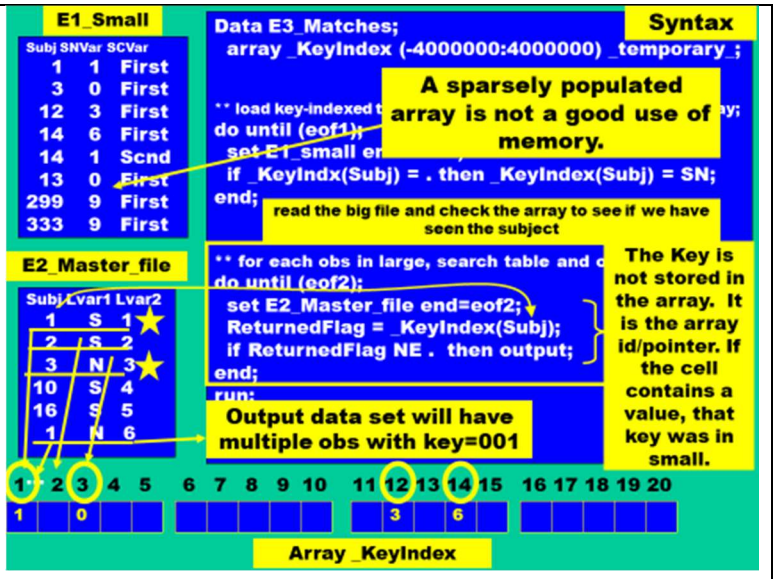

Figure 6

8) WHY HASHING IS FAST (VERSUS A FORMAT LOOKUP)

To see why hashing is considered fast let's compare it to the previous gold standard, the format.

On the left side of Figure 7 you see a format catalog. It contains a subject ID, the formatted variable, and the value that's associated with the format – either "test" or "control".

Formats use a binary search. This seems simple but has been proven, mathematically, to be the fastest search algorithm in a wide variety of situations.

Imagine were looking for subject seven in the format catalog.

The binary search starts at the exact middle of the file, the 15.

After finding a value SAS asks is this the value it wants.

If it is, SAS stops.

If SAS does not have the value it wants (it has a 15 and wants a 7), it decides if it is looking for something larger, or smaller, than what it has.

In this case the answer is smaller and SAS no longer considers the bottom half of the file.

It knows the value it is searching for must be between 001 and 015.

SAS picks the observation exactly in the middle of the new range– 008. It repeats the questions and realizes it must be looking for a smaller value.

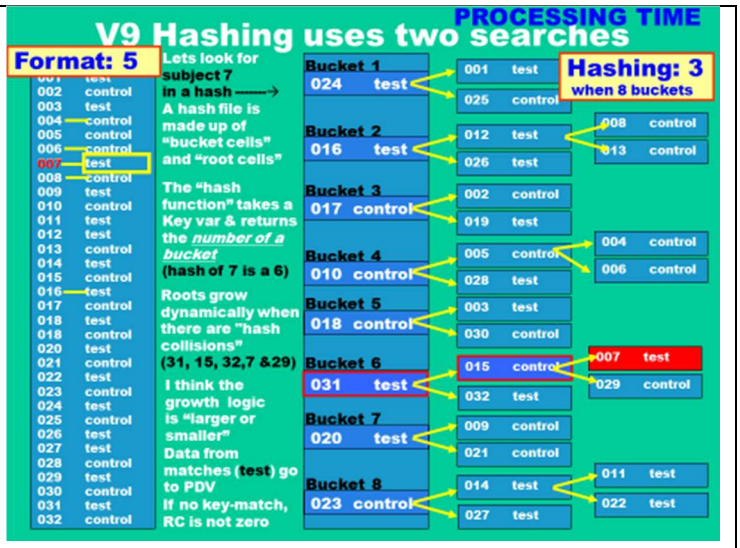

Figure 7

It takes the middle of the "range to be considered" and selects a 4. That number is too small. SAS continues to divide the file in half until it gets the number it is searching for.

It is important to notice that if the format catalog becomes twice as large SAS only has to perform one more search. If the format catalog becomes four times as large, SAS only has to perform two more searches. Format search time is relatively insensitive to the size of the format. In this case the format took five "search cycles".

The right side of Figure 7 shows an eight bucket hash table. Think of the hash table as an array that grows roots. A hash function (kind of a mathy thing) takes the key value of 7 (keys can be character or numeric or a combination) and must return a value between one and the number of buckets in the hash (here an 8). In this example hashing a seven returns the number six. That means that subject seven will be stored in buckets six or in the root below it.

You can see, in Figure 7, that many different subject IDs hashed to a six. 31, 15, 32, 7 and 29 all hash to a six. All of these subject IDs will be stored in, or under, bucket number six. As more subject IDs hash to bucket six the roots grow deeper and deeper. As the roots grow deeper they must be "balanced" to allow for fast searching/retrieval. It seems that the search logic is "bigger or smaller" as is shown in Figure 7.

There are two reasons for hashing being faster than formats.

The first is that hashing was designed for quick retrieval of data and does very few things other than that. Formats, on the other hand, were designed to map one number to another and do a wide variety of things (picture formats, multiple formats, ranges of acceptable values mapping to the same number, etc.) and whenever a format is called SAS must check to see how the format should be applied this particular time.

The hash table does very few things and, because of that, it has little overhead.

The second reason that hashing is so fast is the search pattern that we will see in the next few figures.

If we laid out a format in the same pattern as is used to represent hash tables it would have the appearance of the graphic in Figure 8.

A format proceeds on a binary search by asking if the number is larger or smaller than the one it has. You can see that, to get to the format level that has eight choices, requires three “search cycles”.

To get to the bottom of this “format graphic” requires five “search cycles”.

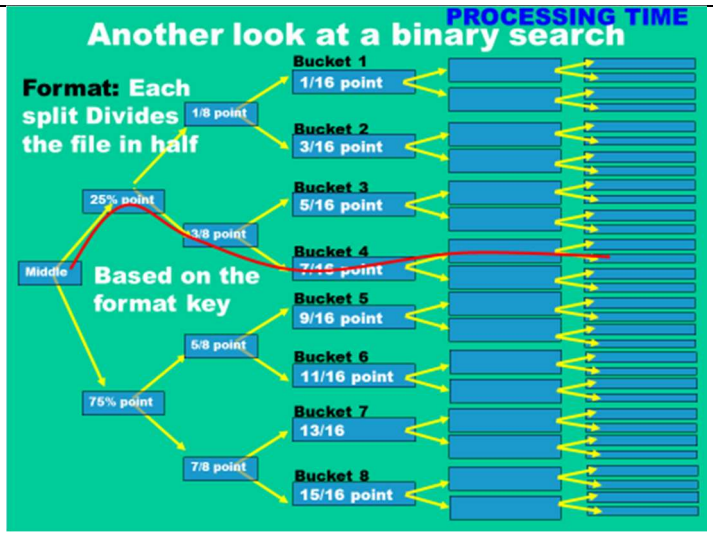

Figure 8

A similar representation for a hash table looks as if the left-hand side of the tree is cut off.

The hash table is created with a certain number of buckets at the top level (here 16) and when the hash function is applied to a key it returns a number between 1 and 16.

For a format to get to this level would take four search cycles”.

To get to the bottom level (deepest root) of this “hash graphic” requires two “search cycles”.

A format would take five search cycles to get this deep and hash search cycles are faster than the search cycle for a format.

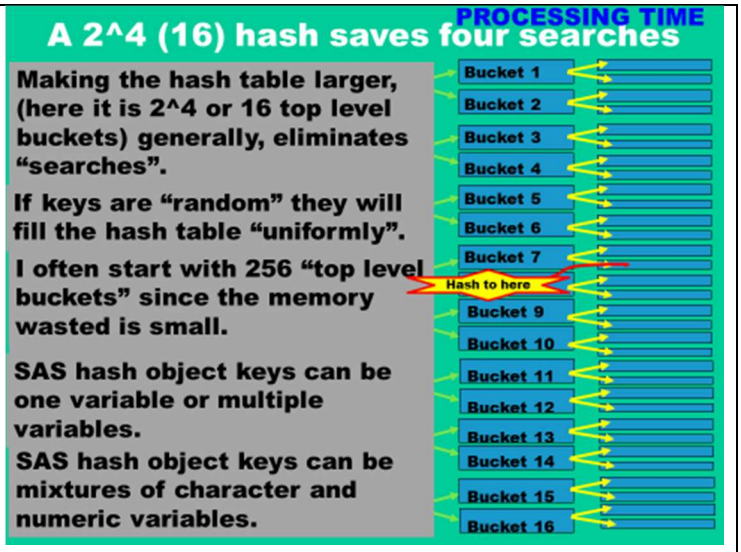

Figure 9

A different representation of the hash table, one that shows it in relation to the other components in the SAS system will be helpful in understanding hash table processing. That representation is introduced in the figure below.

Hash tables exist in RAM as individual "objects".

The major task for a programmer who hopes to use hash tables in their work is to learn how to move data back and forth between the program data vector and the hash table (also called the hash object).

The process of interfacing with the hash object is done with "methods" – think of special subroutines designed to help move data back and forth between the hash table and the program data vector.

Either E1_small or E2_master or both can be stored inside hash tables. We'll see details in the next sections.

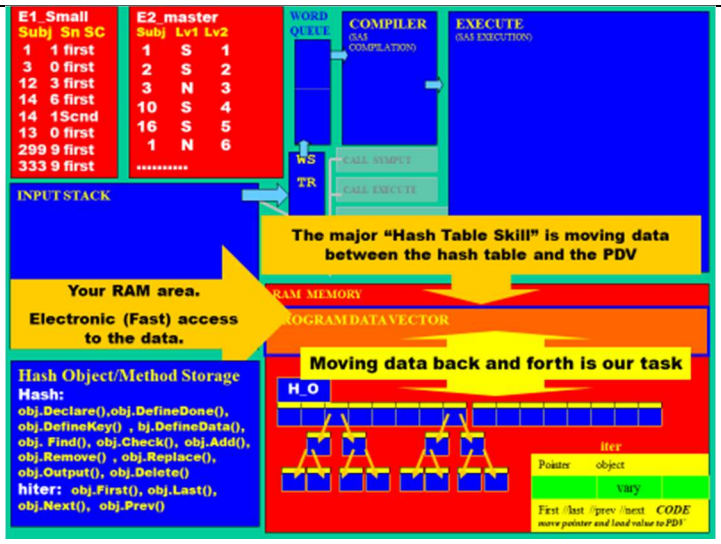

Figure 10

While SAS files can be ordered, or not, a basic hash table cannot be ordered. The data is stored for retrieval using the "is larger/is smaller" logic that can be seen in Figure 7. If you pull data from a basic hash table into a SAS table it will appear in the SAS table in a very unusual sequence.

Ordering a hash table is often done by creating a second hash table that holds the sequencing of the data in the first hash table. Actual details are proprietary.

Figure 10 shows the SAS supervisor – the underlying structure for base SAS. As review; when you submit code it goes to the input stack. WS/TR is the word scanner token router and it is a subroutine that pulls characters off the input stack and assembles them into tokens.

Here's the important part. The compiler collects tokens until it sees a step boundary (a run, a quit, or another PROC). At that point the compiler does its job of checking syntax and creating the program data vector.

The compiler reads the header part of files that it finds in set statements, or merge statements, and puts variables from the file headers onto the PDV. If the compiler sees an assignment statement (like age_mo=age*12) it puts the created variable, age_mo, on the PDV.

The statements that are associated with creating the hash table are not statements for the compiler. Those statements are for the SAS execute mode that follows the compiler.

declare hash A_h_obj(dataset: 'E1_small', hashexp: **16**);

is a statement for SAS execution and even though the data set E1_small is mentioned in the code, ***that "declare hash" statement is not processed by the compiler***.

That means that variables that are in E1_small are **not** put into the PDV. This is a very common problem and the error messages are, sometimes, not very straight forward or helpful.

This issue deserves a few examples and we will explore this problem in the next few figures.

9) HASH TABLE WITH VARIABLE CREATION ON THE PDV FAILURE

Figure 11 shows code that fails because variables that “SAS Execution” expects to be on the program data vector are missing.

The put_all_ sends the contents of the program data vector to the log.

You can see that the PDV only consists of the three RC variables and the automatic variables _error_ and _N_.

During SAS execution, the part of SAS that is establishing the link between the program data vector and the hash table wants to “Mark” the variable SUBJ as having the property of “key”. SAS execution cannot find SUBJ on the PDV and fails.

We will see how to fix this soon.

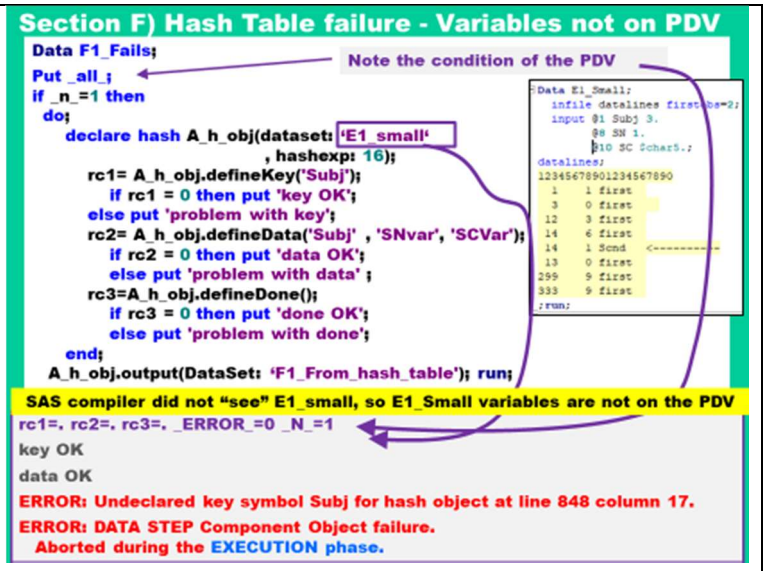

Figure 11

This example would do no useful work. Its sole purpose is to create a hash table and demonstrate hash table syntax.

Figure 12 illustrates the fact that variables in E1_small are not on the program data vector.

Variables on the PDV are assigned additional characteristics when the hash table is created.

A variable, or variables, can be designated as holding key information – holding information that can be used by the methods.

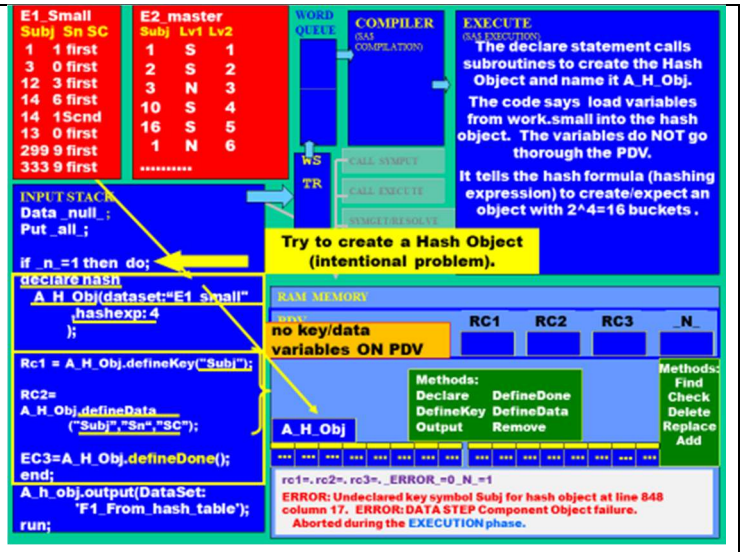

Figure 12

A variable, or variables can be designated as holding data – as holding information that is to be moved from the PDV to the hash table object or from the hash table object to those positions on the program data vector that are designated as holding data.

When the define key method tried to assign a “key” attribute to SUBJ, it failed because SUBJ did not exist on the PDV.

10) HASH TABLE WITH ON INITIALIZED NOTE: SECTION H SUCCESSFUL CREATION OF A HASH TABLE

This example would do no useful work. Its sole purpose is to create a hash table and demonstrate hash table syntax.

As Figure 13 shows, just having the variable on the PDV is sometimes not enough to have a clean log. This code runs but leaves a note in the log that would not be allowed in some companies.

The length statements are seen by the SAS compiler and they *do* create variables on the PDV. Says execute has no problem running the code that creates the hash object.

At a later stage in execution SAS execute sees that variables are uninitialized and writes a note to the log.

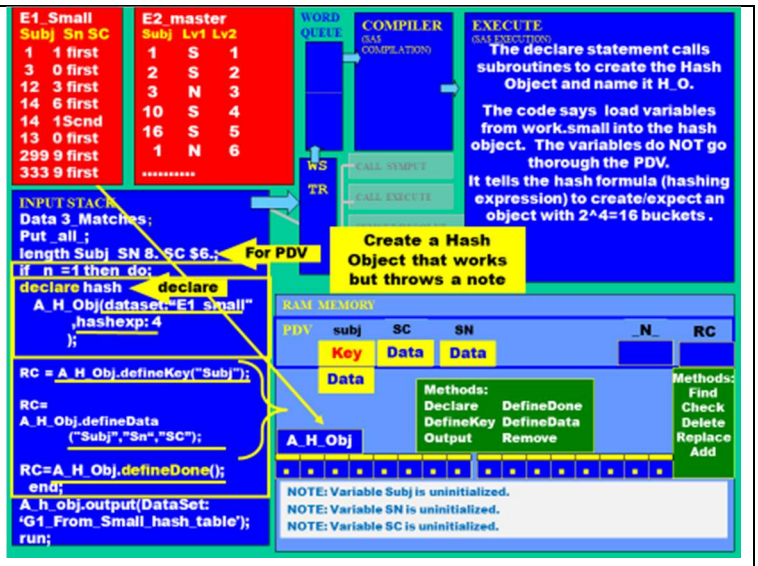

Figure 13

11) HASH TABLE CREATION SUCCESS

This example also does no useful work. Its sole purpose is to create a hash table and demonstrate hash table syntax.

Figure 14 shows the very common and very useful trick to avoid the problems in the previous examples.

```
if 0 then set E1_small;
```

Above is a normal if statement and since zero is considered a false it never runs.

What this does is allow the compiler to see "set E1_small;" and create all of the variables from that data set in the PDV. This solves a lot of problems and requires little thinking or typing.

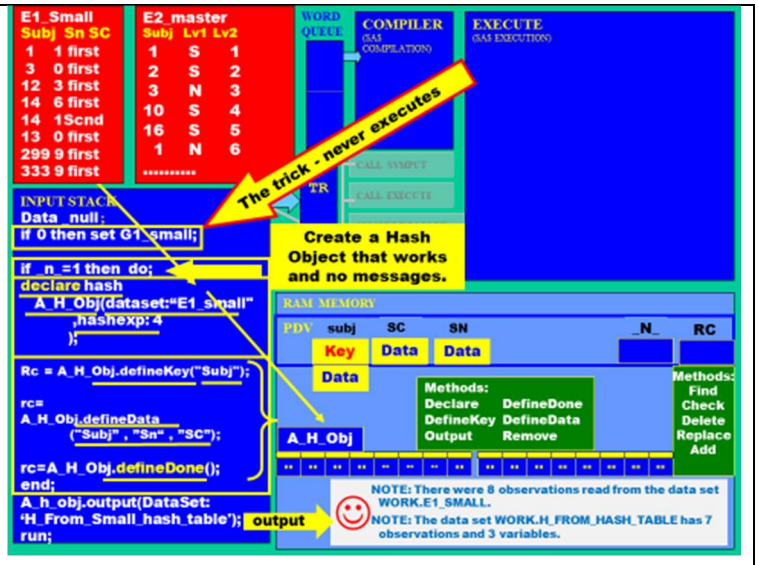

Figure 14

This code successfully creates a hash table and sends it to a SAS file.

It would now be appropriate to discuss the syntax for the hash object.

Since the code that creates the hash object is "SAS execution code" we want to keep it from running every time control loops through the data step. The typical way to do this is to embed all the hash table creation code between

```
if_N equals one then do;
```
and
```
end;
```

```
DECLARE HASH A_H_OBJ(DATASET:"E1_SMALL", HASHEXP: 4);
```

The above statement tells SAS execute to create a hash object with the name “A_H_Obj”. That hash object will have 2^4 or 16 buckets in the top level and the data set E1_small should be loaded into that hash object. Please note that with this syntax E1_small does not go through the program data vector as it gets loaded into the hash object. It is loaded directly into the hash object.

Rc = A_H_Obj.defineKey("Subj"); tells SAS that the variable SUBJ on the PDV, will be used as the key variable. If you want to access data associated with a subject (say subj 003) that is in the hash object you must load 003 into the variable Subj on the PDV and then issue a method like A_H_Obj.find().

Think of the variable that is defined as “key” as holding a pointer that identifies where information exists in the hash object. Find will only “look” in “key” variables for information identifying what data is to be returned in the hash object.

rc=A_H_Obj.defineData("Subj” , ”Sn“ , ”SC”); tells SAS that the variables Subj , Sn and SC are defined as data on the PDV. If you are loading data into the hash table, that data must be stored in the variables Subj , Sn and SC. If you are retrieving data from the hash table, that data will be moved from the hash table into the variables Subj , Sn and SC on the PDV.

rc=A_H_Obj.defineDone(); tells SAS Execute that it has seen all of the commands it needs to create the hash object the programmer wants to have created

12) HASH TABLE LOOKUP – THE ORIGINAL PURPOSE OF THE HASH TABLE

Figure 15 shows the code required to do a basic (no repeating keys) hash table lookup. The hash table was designed to perform this function and ,with this basic syntax, duplicate keys will be ignored when if the file being loaded into the hash table as a duplicate key.

Since this is the function of a hash table we should work through this fairly slowly.

Imagine you have been given a small file and told to find all matching records from some big file (maybe a customer master).

It is generally recommended to load the smaller file into the hash table and not the larger file.

Code Section I: hash table lookup – original purpose

```
Data I_found_in_big_file;
/*If 0 lets the compiler see the file &prevents reading of the file*/
 if 0 then set E1_small;        <- The trick

if _n_=1 then                   <- Ignore the duplicate Key in the small file
do; /*this just sets up the hash table and loads G1_small into it*/
     declare hash A_h_obj(dataset: 'E1_small', hashexp: 4);
        A_h_obj.defineData('Subj' ,'Sn', "SC");
        A_h_obj.defineDone();   <- Create and load Hash
end; /*if _n_=1 then*/

do until (eof); /*loop through the big file- data from Master in on PDV*/
  set E2_Master_file end=eof; /*r...   <- SAS loop read of master
  rc=A_h_obj.find();            <- Use find() method to move data to PDV
/*use Subj in the PDV, from the large file, to "find"
 (and try to move data vars back to the PDV) that subj in the hash*/
  if (rc=0) then output;        <- Output if find() was successful
end;/*do until (eof)*/
run;
Proc print data=I_found_in_big_file; run;
```

Figure 15

Remember the main skill to be learned is how to move data between the program data vector and the hash table/object.

You can see the `if 0 then set E1_small;` being used to load the program data vector with variables from E1_small. The hash table/object is going to be named “A_H_obj”.

`A_h_obj.defineKey('Subj');` tells SAS that methods (think subroutines) that are used to search for data in the hash object should look at the variable Subj, on the PDV, to get information on which data element should be accessed on the hash table.

A_h_obj.defineData('Subj' , 'Sn', "Sc"); tells SAS that methods that are used to move data between the PDB and the hash object should take data from these variables and also move data to these variables.

As Figure 16 shows the compiler will create, on the PDV, variables from E1_small (because of our if zero trick) and also E2_master (because it is in normal code and can be seen by the compiler.

This merge is done inside two sections of code.

The first section only executes when _N_ =1. We've seen hash objects be defined before and need not review.

The second section of code is a do until loop and it is used to read the large file.

It is interesting to note that _N_ will only take the value of one in this data step execution.

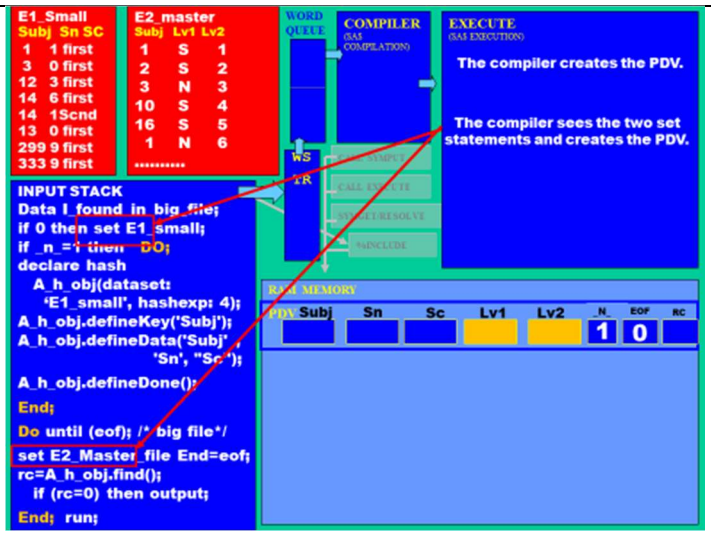

Figure 16

The thing that's new for us to discuss is just below the bottom red box.

`rc=A_h_obj.find();` tells SAS to look in the variable SUBJ on the PDV and get a value for subject.

That value of subject is passed to the methods (think subroutines) that have been written to move data back and forth between the PDV and the hash object.

The method called is the find method and that bears a little discussion. The full syntax is the name of the hash object followed by a period followed by Find(). I found this two-part syntax a bit confusing for a while and then I realized that you can have more than one hash object in existence at the same time. When more than one object exists then there must be a technique that allows you to identify the object that you would like to have perform an action.

Imagine your son coming into the house one evening and saying:
"Mom, can I have $20?"
"Dad, can I borrow the car?".

This is very similar to the syntax of object oriented programming. The programmer must specify the thing that should do the action and also the desired action.

For quite some time I was a little annoyed/confused by the naming of the find command. The find command does more than just ***find*** something in the hash object. The find command will find the key in the hash object (if it exists) and (if the key was found in the hash) move the associated data back to the PDV. Data is moved into variables on the PDV that have been defined as type data.

It should be noted that only data is moved from the hash back to the PDV - and not the key. If you're compulsive about keeping things in sync you can define the variable, that you are using as key, to also be data. That is what has been done in this example – just to be cautious.

I would've much preferred that this method be given a name that really suggests the complexity of what it does. I would have preferred the command to be "get" because that suggests that you're going to get something back from where it is currently located. My annoyance lessened as I've gotten used to the find. "Find and return" would be a great name for what this method does but is a bit too long.

`rc=A_h_obj.find();` is the full statement and deserves a little more explanation.

When the find method takes a value from SUBJ in the PDV and attempts to find that in the hash object the method sends a return code (hence naming the variable rc, but it could have any name) back to SAS and the PDV. With the syntax above we are saving the return code to a variable called rc.

If the find method was successful in finding the key in the hash object it sends back a zero and that is stored in the variable rc. A zero as a return code means a successful find. If the key (the value of SUBJ) could not be found in the hash object the method sends back a return code that is not zero. A non-zero rc indicates a failure to find.

In this example we are doing something very similar to an inner join. If the return code is zero the subject ID must have existed in both E1_small and E2_master. Only those observations will be output, using the normal SAS output, to the data set named I_found_in_big_file.

That is an overview of the process and the next several figures will show details of the process.

Figure 17 shows the execution of the first section and the creation of the hash object. The top level of the hash object is 2^4 buckets wide. You can see the variables that are defined as key and as data. We are about to load the contents of E1_small into the hash object

Please note that with this syntax, where were using the data set: syntax, E1_small does not go through the program data vector as it is loaded into the hash object. Avoiding the PDV saves time.

There are methods that are specifically created to move data from the PDV to the hash object and we will see them in future figures.

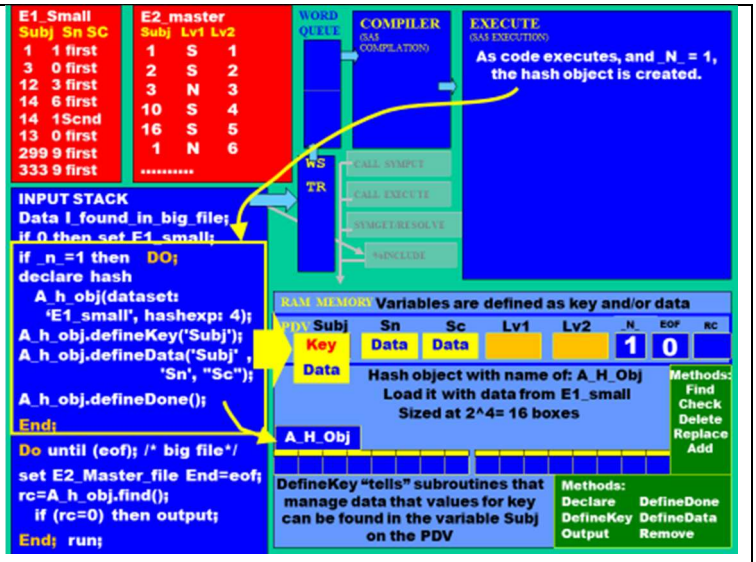

Figure 17

Figure 18 shows the hash object after it's been created. Buckets in the hash object contain both key and data elements.

As you can see, when multiple key values hash to the same top bucket on the hash object, the hash object grows roots to store the data.

This automatic growing of roots is a major feature of hash objects. You do not have to pre-allocate memory to a hash object and any hash object, even one with only two buckets at the top level, can store any amount of data (limited by your RAM).

The code in the yellow box has finished executing and never executes again.

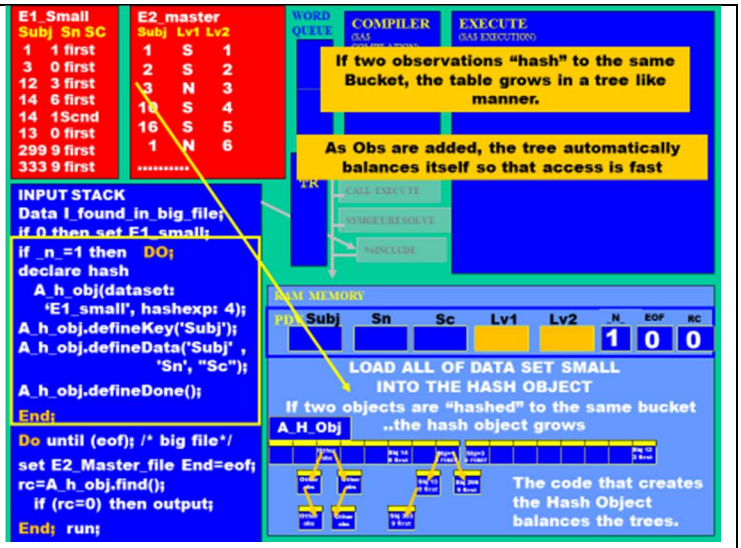

Figure 18

Figure 19 shows the start of the execution of the second block of code (see golden rectangle).

This block will take advantage of a "SAS loop read" of the data set E2_master file.

The first row of data has been read into the PDV. SAS is about to execute the find method.

You can see that there is a value in the variable SUBJ and that will be used as the key in looking for data in the hash object.

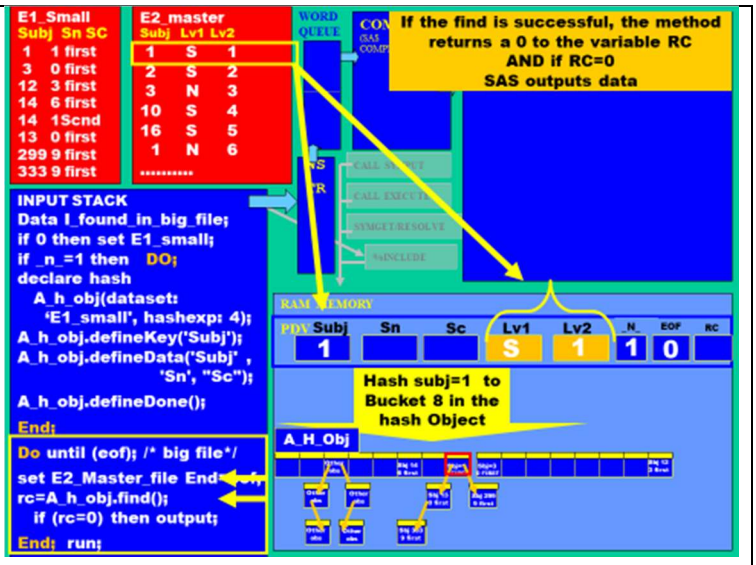

Figure 19

Figure 20 shows the result of a successful find.

The key was found in the hash object and the data that was associated with that key has been moved back to the PDV.

Since the lookup was successful the return code gets a value of zero.

Since the return code is valued at zero this program data vector will be sent, via a normal SAS output, to the file named I_found_in_big_file

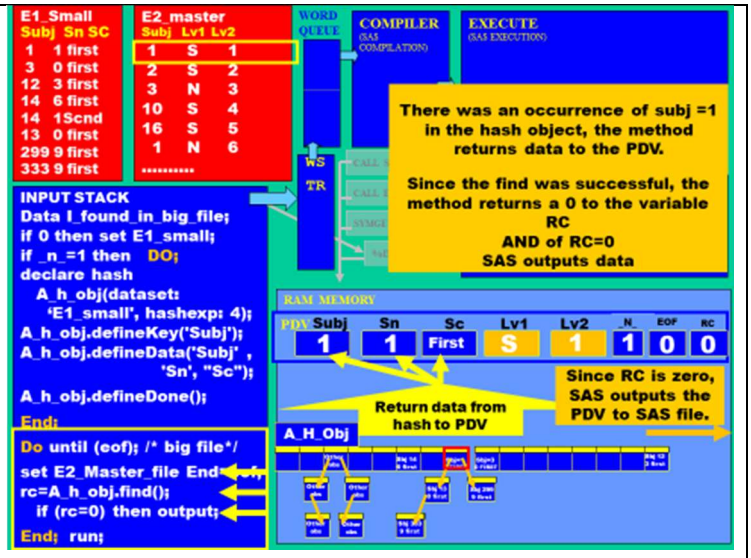

Figure 20

So far we have seen how the hash table originally functioned in SAS. Original implementations were limited to situations where keys were unique. As we've seen before duplicate keys cannot be loaded into a hash table and nothing is written to the log if such an attempt is made. The next section will discuss duplicate keys.

13) DUPLICATE KEYS IN A HASH TABLE

In order to better understand the subtleties of using duplicate keys and the SAS hash table we will use a compare and contrast technique. We are going to see examples of different coding syntax, some of which will be incorrect, to illustrate duplicate keys in a hash table.

hash table creation fail – no use of multi-data & hash table creation success

Figure 21 shows two examples. Neither example would do any useful work. They are intended solely to illustrate the creation of a hash table.

The top example fails to store duplicate keys. It uses the syntax we have seen before.

```
dcl hash J_oops
(dataset: 'J1_Dupe'
 , ordered: 'A');
```

The bottom example uses new syntax as it defines the hash object.

dcl hash J_oops(dataset:'J1_Dupe', ordered: 'A', multidata: "Y");

tells SAS to expect that data being loaded into this hash table will have repeats of keys.

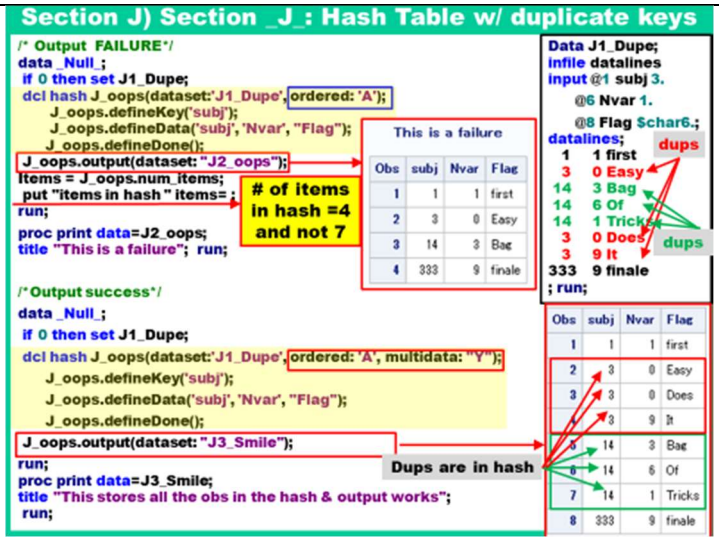

Figure 21

Note that the code, in Figure 21, uses an output method. This is not the same output that we used in regular SAS. This one method call outputs all of the observations in the hash object to a SAS file and only needs to be called one time. The file was printed and can be seen in the right bottom corner of figure 21.

hash table traverse failure (use of find instead of find_next) & hash table traverse success

Figure 22 also shows two examples.

The first example uses ONLY a find method in an attempt to pull data back from the hash object. As you can see find will only pull back the first element of the repeating hash entry.

The bottom example uses a find method to position the hash object pointer to the first element of the repeating hash entry and then a find next method to “traverse” all of the elements of the repeating hash entry.

This is complicated enough so that details of this will be shown in the next few figures.

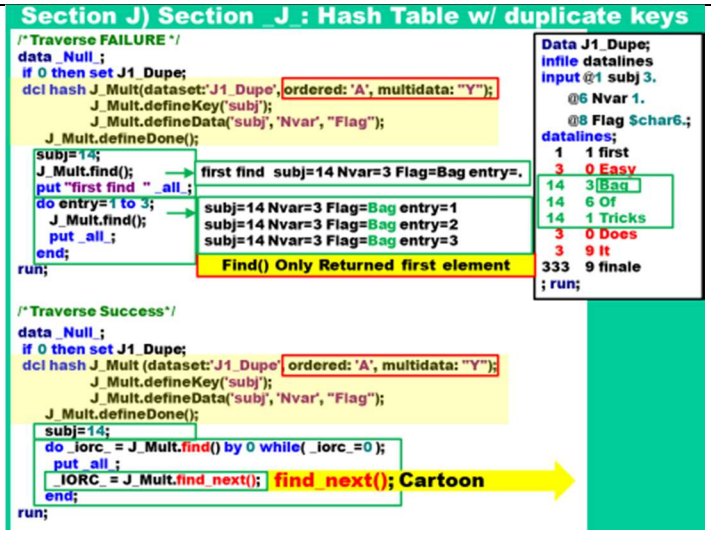

Figure 22

Figure 23 shows my interpretation of the hash table with repeating values for keys.

The name of the hash table is J_Mult and it has two buckets at the top level.

One top-level bucket holds data for SUBJ=3 and the other top-level bucket holds data for keys where SUBJ equals 1, 14, or 333.

This figure illustrates the execution of the code in the gold box shown at the top of the input stack. The has object has been created.

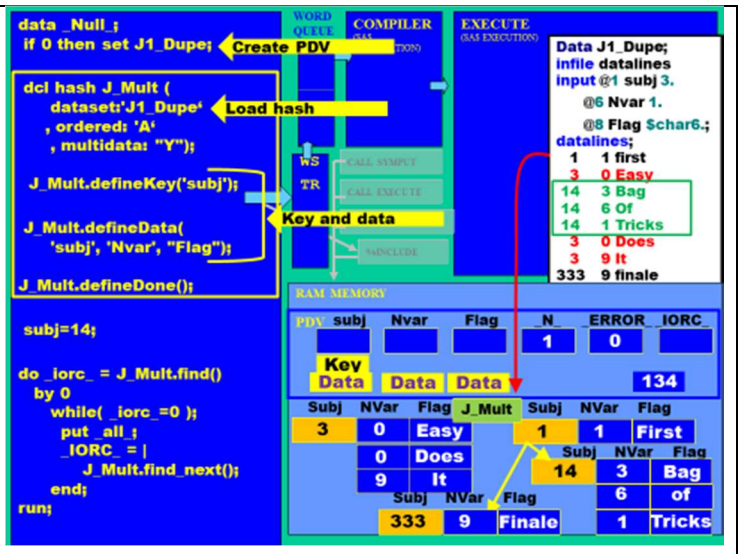

Figure 23

This messy figure shows find retrieving data (3 and bag) from the hash object.

SUBJ equals 14 is hardcoded and put onto the PDV when that line of code executes.

Then the do _IORC_ block executes. _IORC_ is a reserved variable on the PDV that SAS uses to monitor, on its own processes, return codes. _IORC_ stands for input output return code.

Dr. Dorfman, and this example follows his work closely, used IORC because it (like _N_ and _ERROR_) is automatically removed from the PDV on output – thereby saving him the trouble of writing a “drop”.

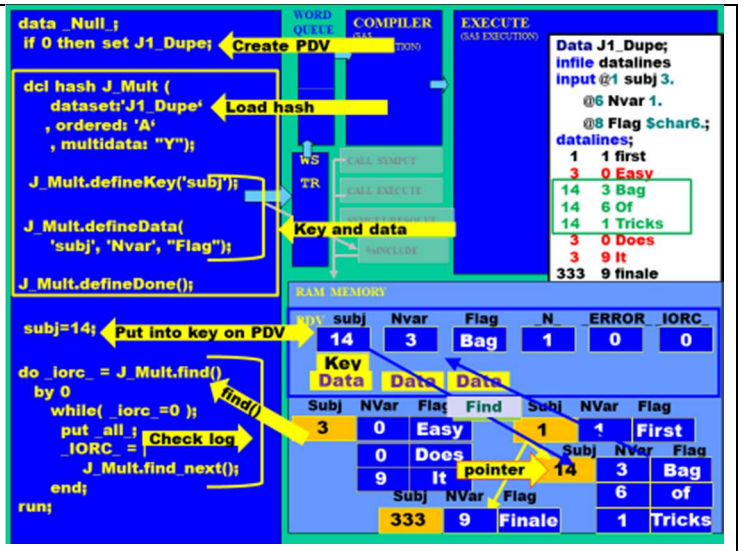

Figure 24

Figure 25 shows the next execution of the do _IORC_ loop. This loop uses a find_next and not a find.

The find_next automatically traverses the different elements inside that entry in the hash table and returns data to the PDV.

When the traverse goes past the last element inside an entry in the hash table, the return code is something other than a zero and SAS exits the loop.

There are several methods not covered so far and I encourage an interested reader to look for articles. There are many good SUG papers.

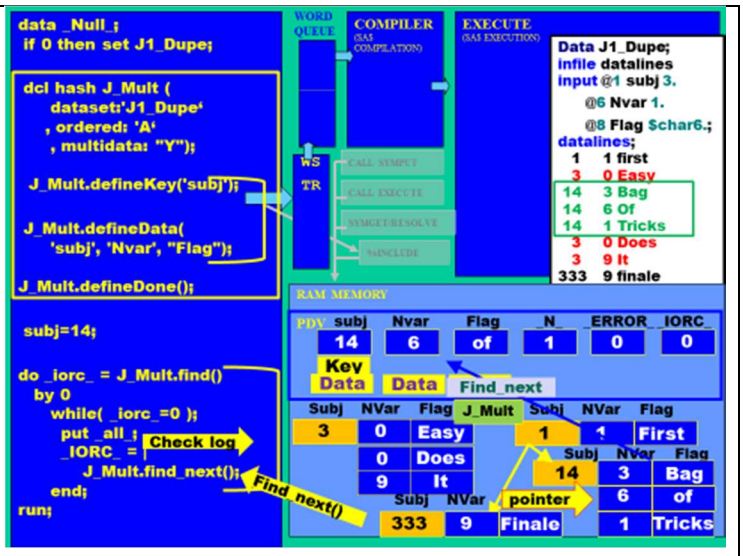

Figure 25

14) HASH TABLE REPLACING A PROC SUMMARY AND A MERGE

While the hash table was designed to be used for table lookup, programmers quickly found many other uses. Figure 26 shows how a hash table, with one of the other methods associated with hash tables, can be used to replace a PROC Summary and perform an calculation inside one data step.

The red box puts variables on the PDV.

I was stretched, while writing this paper, to come up with a short but interesting example of this functionality. I'm afraid that my example is not very good and will confuse a reader. In the next figure, I will show, and explain, the output in an attempt to help understand the wacky logic in this example.

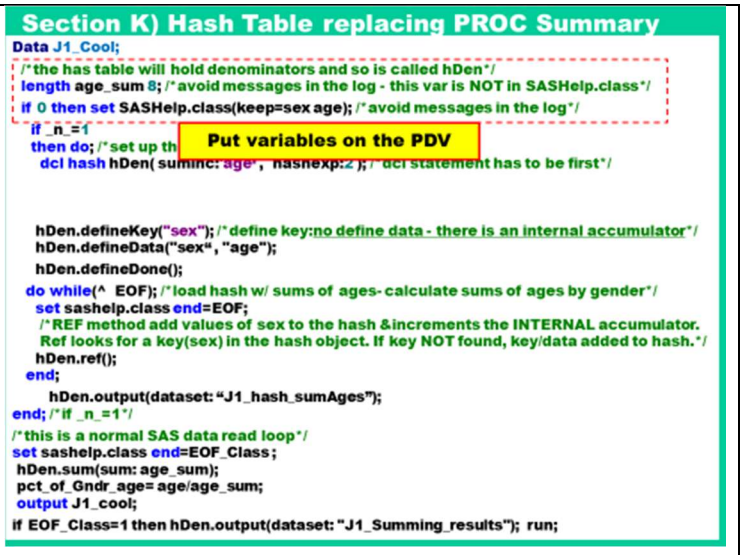

Figure 26

For some (strange) reason the client wants to assign to each student the percentage of the “total ages by gender” in a data set for which that student is responsible.

The total ages of females in this data set are 119. Joyce, with an age of 11, is responsible for .092437 of that total.

Judy, with her age of 14, is responsible for .117647 of the total ages for females in that data set.

I struggled to find a business parallel for this functionality.

This example is strange but it does fit on one PowerPoint slide and illustrates the functionality.

Please accept my apologies for such a nonsensical example.

People with the same age and gender have the same %

Name	Sex	Age	age_sum	pct_of_Gndr_age
Joyce	F	11	119	0.092437
Jane	F	12	119	0.10084
Louise	F	12	119	0.10084
Barbara	F	13	119	0.109244
Alice	F	13	119	0.109244
Carol	F	14	119	0.117647
Judy	F	14	119	0.117647
Mary	F	15	119	0.12605
Janet	F	15	119	0.12605
Thomas	M	11	134	0.08209
James	M	12	134	0.089552
John	M	12	134	0.089552
Robert	M	12	134	0.089552
Jeffrey	M	13	134	0.097015
Alfred	M	14	134	0.104478
Henry	M	14	134	0.104478
William	M	15	134	0.11194
Ronald	M	15	134	0.11194
Philip	M	16	134	0.119403

Figure 27

Please see Figure 26 if one needs to see all of the text. The red box executes one time and both creates the structure of the hash object and uses a SAS loop read to load individual observations into the hash object.

Suminc:”age” creates an internal summation inside the hash object that holds the sum of the ages.

This SAS loop read puts individual observations onto the program data vector and uses the ren method (see Figure 27).

Ren does several things. If a key is not in the hash object, Ren will create that entry in the hash object and load the value of age into that entry.

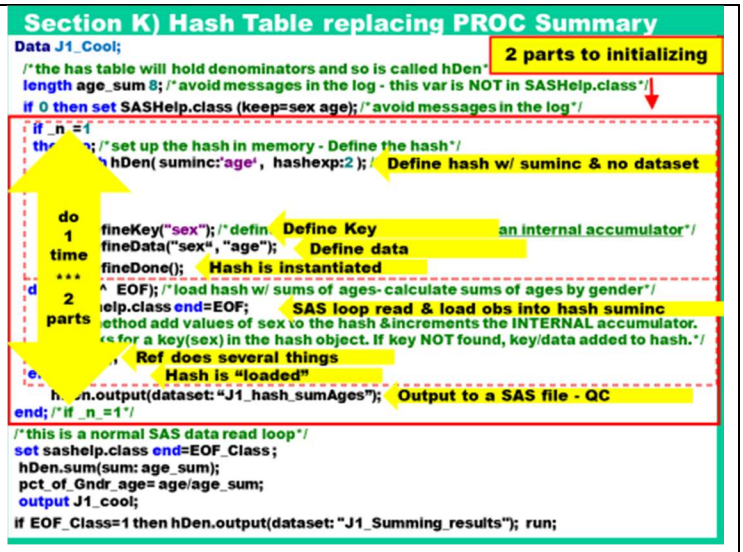

Figure 28

If Ren does find that key in the hash object, it simply adds the value of age to the internal summation inside the hash object. The internal summation is called age_sum.

Figure 29 shows the first load of data into the hash object.

The red international not sign indicates that ref did not find M as a key value in the hash object.

Therefore ref created an entry in the hash object and added Alfred's 14 years to the internal summary.

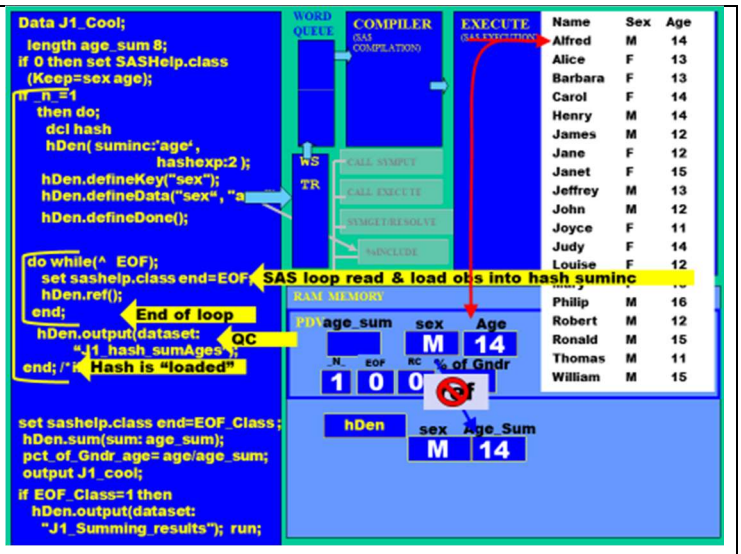

Figure 29

Figure 30 shows processing of the third observation from SAS HELP.class.

Ref did find an entry for F and did not need to create another hash element to hold females.

At this stage and simply had to add Barbara's 13-years to Alice's 13 years.

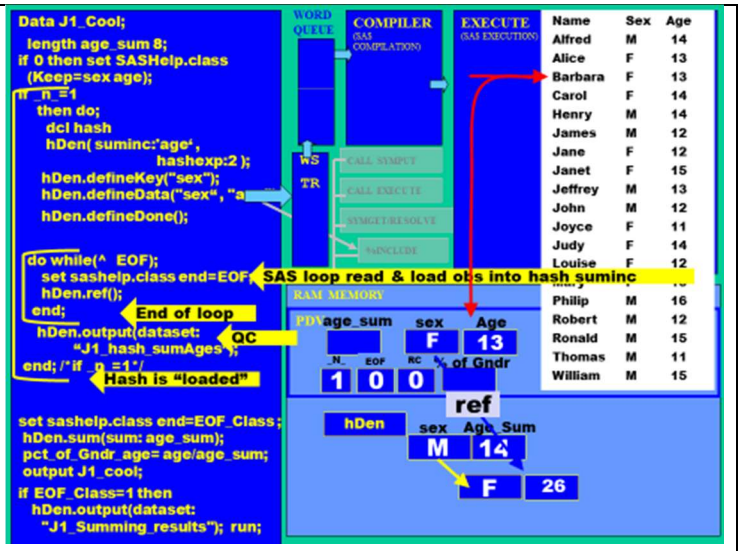

Figure 30

Figure 31 shows the processing of William's data.

At this point, SAS has completed the execution of all the code inside the if _N_=1 block.

This code never executes again and the rest of the logic is going to be determined by the code at the bottom of the input stack.

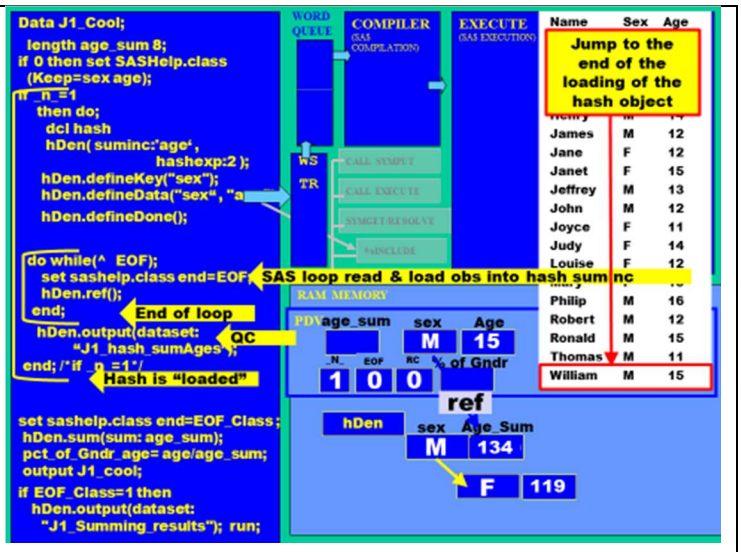

Figure 31

Figure 32 illustrates the processing of the second row of data through the loop at the bottom of the input stack. We read Alice's information into the PDV.

hDen.sum returns the automatic summary field from the hash object into a variable on the PDV that is named age_sum.

We calculate percentage of gender and use an ordinary SAS output to send this PDV to a data set called J1_cool.

If we have processed all of the data and EOF_class equals one then we use a hash object output method to output the whole hash object to a data file called J1_summing_results.

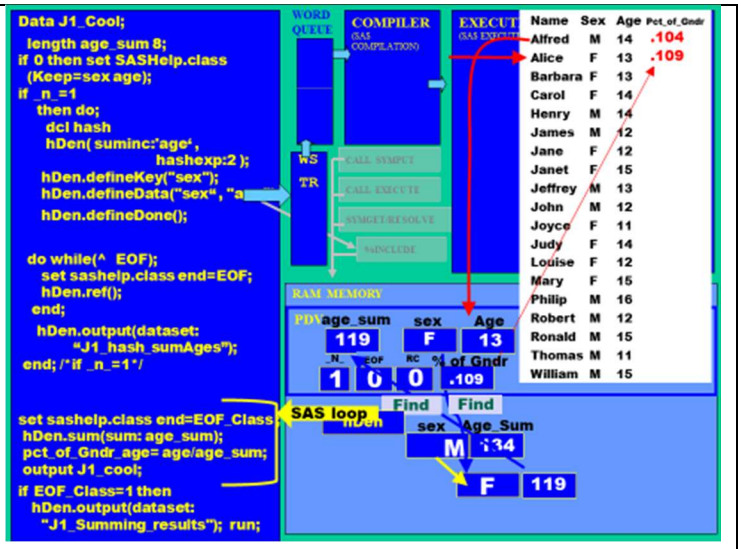

Figure 32

15) HASH TABLE SUMMARIZING SALES REP ACTIVITY

As observed before examples with unusual business logic are more difficult to understand. Hash tables will be used to create the report below. The report summarizes the visits of sales representatives on different types of doctors and might be another "odd and confusing" report.

The client wanted to have all of the doctor activity put on one row in a report. The report contains the sales reps name and some characteristics about their visit on the doctor. The report tells whether the doctor is a "high writer" a "medium writer" or a "low writer" and the date of the visit

```
Data L1_SalesCalls;
  infile datalines truncover firstobs=2;
  /*Sales rep*Doctor Tier*Visit Date*/
  input @1 Rep $char6.
        @9 tier $char5.
        @15 Date & MMDDYY8. ;
datalines;
12345678901234567890
Ahmed   Hi_Rx 01/02/20
Bo      MedRx 02/02/20
Ahmed   Hi_Rx 02/02/20
Carlos  LowRx 03/02/20
David   LowRx 02/02/20
Bo      MedRx 03/02/20
Carlos  LowRx 04/02/20
David   LowRx 04/02/20
Ahmed   LowRx 05/02/20
;
run;
```

Please say that again

Obs	Rep	Hist
1	David	LowRx 02/02/20 -- LowRx04/02/20
2	Carlos	LowRx 03/02/20 -- LowRx04/02/20
3	Bo	MedRx 02/02/20 -- MedRx03/02/20
4	Ahmed	Hi_Rx 01/02/20 -- Hi_Rx02/02/20 -- LowRx05/02/20

Figure 33

Figure 33 shows the raw data and the resulting report.

Figure 34 shows the code required to create the above report. Note two solid red boxes/

The top red box defines the structure of the hash table. This, as is usual with hash tables, only executes one time.

The rest of the code (second solid red box) executes in the context of a typical SAS loop- read and has two components.

The code in the solid red box in the bottom of the figure executes one time for every observation in the source data set.

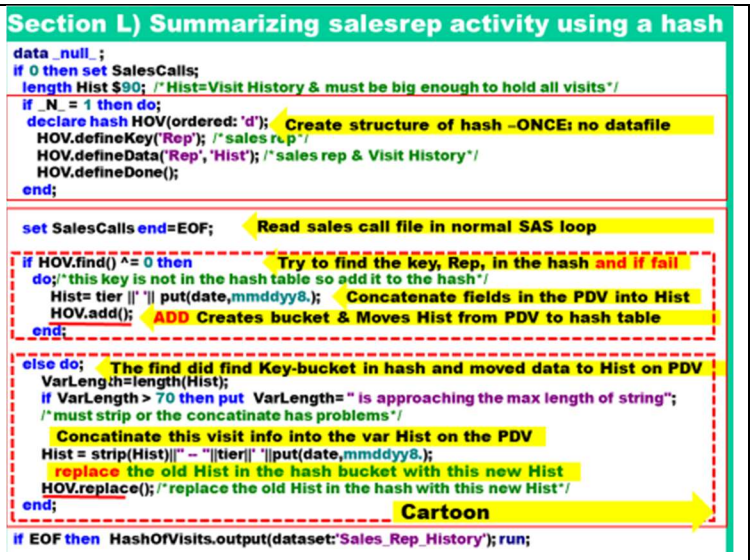

Figure 34

Please note that, in the _N_=1 block of code (top solid red box), the hash table was defined but not loaded. The hash table is loaded in the bottom solid red box.

The code in the bottom solid red box has two components – one component executes if the sales rep is already in the hash table and the other component executes if the sales rep is not in the hash table.

A key is read in from SalesCalls and a find method is used to pull data back from the hash table into the PDV.

The top dotted red box executes if the find was NOT successful. The bottom dotted red box executes if the find was successful. Note that I have a bit of code that checks to see if the variable HIST has been overloaded. That is not important in this context what is just a bit of defensive coding.

Figure 35 illustrates the next step in the process- processing Ahmed's first row of data..

If there was no occurrence of that key in the hash table information in the tier and date columns are concatenated and loaded into HIST.

Then the add method creates an entry in the hash table to hold the values of hist.

This process is repeated every time there is a new value of the key variable – a new value for the variable REP.

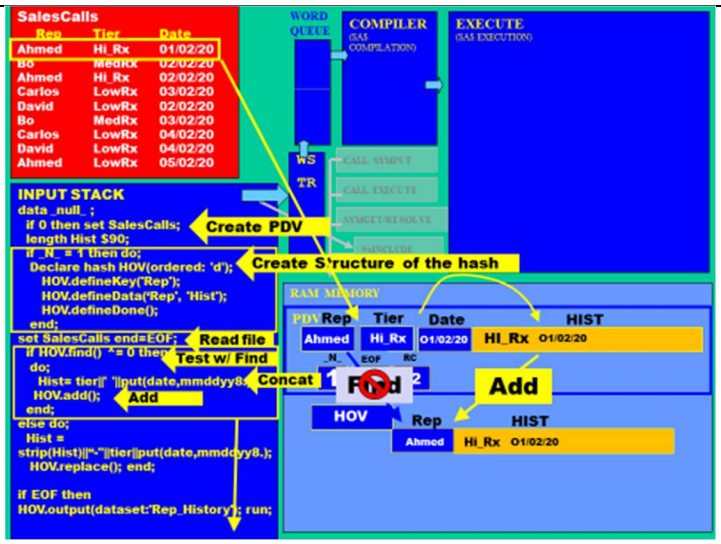

Figure 35

Figure 36, unfortunately, shows the state of the system at several different times in one static image.

Ahmed is loaded into rep because it's read from the source file.

HOV.find() checks to see if Ahmed is in the hash table and he is. At this time the hash table only contains the black part of the string (see figure 35).

The find takes the black part of the and loads it back into HIST in the PDV.

Next the concatenation of the information in Tier and date variables happens (creating a string that is shown in red). That is added to the end of the black string in HIST on the PDV.

st

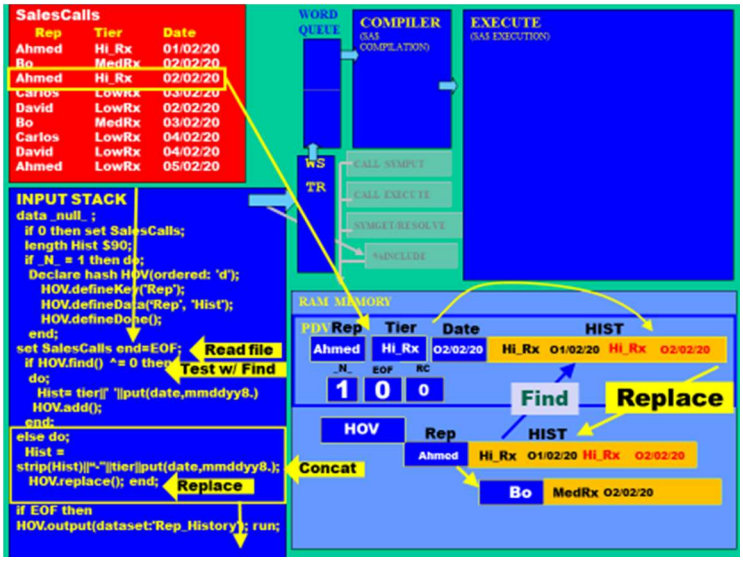

Figure 36

Finally; the replace method is used to load the string from the HIST variable back into the hash table. This method replaces the value in the hash table with the value in that data out variables on the PDV.

16) USING A HASH TABLE AS PART OF A THREE WAY JOIN

Figure 37 shows how hash tables can be used in a three-way join.

Imagine that we want to add sophomore and junior GPAs to a list of students. Imagine that the name of the students is not going to be duplicated and can be used as a key.

Remember, that the problem with normal SAS merges is that the data must be sorted and sorting takes time.

This technique can join three data sets without sorting any of them.

The code is shown in figure 37 and you can see that it has two major parts.

The first part is inside an if _N_=1 block and loads the two hash objects.

```
Section M) A three Way Merge
Data N3_3_Way
if 0 then set sashelp.class   N1_So_GPA      N2_Jr_GPA;   Create PDV
if _N_ =1 then                               Initialize one time
 do;   declare hash H_So(dataset:'N1_So_GPA');
             H_So.defineKey('Name');
             H_So.defineData("Name" , 'So_GPA');          Load Soph GPAs into hash – one time
             H_So.defineDone();
       H_So.output(dataset: "N4_Soph");

       declare hash H_Jr(dataset:'N2_Jr_GPA');
             H_Jr.defineKey('Name');
             H_Jr.defineData("Name" , N2_Jr_GPA');        Load Jr GPAs into hash – one time
             H_Jr.defineDone();
       H_Jr.output(dataset: "N4_Jr");
 end; /*if _N_ =1 then*/
   set sashelp.class;                        Normal SAS loop here – read each observation

   rc_Soph = H_So.find();                    Try to Find/pull GPA from H_So
      if rc_Soph NE 0 then So_GPA=0;         If not found set to 0
   rc_Jr =H_Jr.find();                       Try to Find/pull GPA from H_Jr
      if rc_Jr NE 0    then Jr_GPA=0;        If not found set to 0
run;
```

Figure 37

The second section of code is controlled by a normal “SAS loop read”.

That “SAS loop read” reads an observation and executes two find methods.

The first find method tries to get a sophomore grade added to the PDV.

The second find method tries to add a Junior grade to the PDV.

As was done before the return code monitors the success of the find.

If we do not have a sophomore, or junior grade, in the hash table we enter a zero in the PDV for those variables.

```
Section M) A three Way Merge
Data N3_3_Way
if 0 then set sashelp.class   N1_So_GPA      N2_Jr_GPA;   Create PDV
if _N_ =1 then                               Initialize one time
 do;   declare hash H_So(dataset:'N1_So_GPA');
             H_So.defineKey('Name');
             H_So.defineData("Name" , 'So_GPA');          Load Soph GPAs into hash – one time
             H_So.defineDone();
       H_So.output(dataset: "N4_Soph");

       declare hash H_Jr(dataset:'N2_Jr_GPA');
             H_Jr.defineKey('Name');
             H_Jr.defineData("Name" , N2_Jr_GPA');        Load Jr GPAs into hash – one time
             H_Jr.defineDone();
       H_Jr.output(dataset: "N4_Jr");
 end; /*if _N_ =1 then*/
   set sashelp.class;                        Normal SAS loop here – read each observation

   rc_Soph = H_So.find();                    Try to Find/pull GPA from H_So
      if rc_Soph NE 0 then So_GPA=0;         If not found set to 0
   rc_Jr =H_Jr.find();                       Try to Find/pull GPA from H_Jr
      if rc_Jr NE 0    then Jr_GPA=0;        If not found set to 0
run;
```

Figure 38

CONCLUSION

The hash table is a useful addition to any SAS programmer’s tool belt. It’s especially helpful in situations where sorting of data must be avoided.

REFERENCES

Axelrod,2018, “HASH Beyond Lookups –Take Another Look” proceedings of the SAS global Forum 3125-2019 , Dallas Texas

Dorfman, and Henderson, 2017 “Beyond Table Lookup: The Versatile SAS® Hash Object” proceedings of the SAS global Forum 821-2017 , Orlando Florida

Dorfman and Vyverman, 2006, Data Step Hash Objects as Programming Tools proceedings of the SAS Users Group 21, 241-31, SanFrancisco CA

Dorfman, 2016, Data Step Hash Objects as Programming Tools Proceedings of the SAS global Forum 10200-2016, LasVegas Nevada

Dorfman and Henderson 2015: Data Aggregation Using the SAS Hash Object
SAS community.coghttp://www.sascommunity.org/wiki/Data_Aggregation_Using_the_SAS_Hash_Object

SAS 9 Hash Object Tip Sheet http://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf

Lavery, “An Animated Guide: Power Merges: The format table lookup,” NESUG 16, September, 2003, Washington, DC.

Lavery, “An Animated Guide: Speed Merges with Key Merging and the _IORC_ Variable” WUSS 2003

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Russ Lavery
Contractor
russ.lavery@verizon.net