Hashing performance time with Hash tables

Elena Muriel, Amadeus Software Limited

ABSTRACT

With SAS® 9 two new types of object become available in the data step environment. They are called hash tables and hash iterator objects, and their performance speed will amaze anybody having to merge, sort or perform table lookups with large data sets.

This paper includes a basic discussion on the concept of hash tables and the theory that lies behind them. As a hash table is an "associative array", we will discover that some advanced programmers have already been benefiting from hash tables in previous versions of SAS.

The main section will be a more practical approach into the definition and use of hash tables in SAS 9. Through simple code examples we will illustrate the new methods and properties that are available and how to use them to radically improve your performance time. We will learn how to make use of the iterator object to order data and summarise information without the need for a SAS procedure.

The later part of this paper will include a comparison between some traditional lookup methods (e.g.: formats, merge and SQL) and hash tables to try and convince any leftover sceptics of the full power of hash tables.

INTRODUCTION

Hash tables are not a new concept, as they have been extensively used in other programming languages (such as C++ and Microsoft .NET languages C# and VB) for performing fast table lookups.

Until recently, hash tables have been a pretty unknown concept for the SAS programmer. With the introduction of SAS 9, SAS Institute has added this powerful built-in feature into one of its basic programming structures, the data step environment.

So why are hash tables different to other existing lookup methods? Because the search they perform is based on direct addressing instead of the traditional comparison methods. What this means is that instead of having to iterate through all the observations in order to find a match (as happens with merge statements, SAS indexes and formats), the new structure already knows where to find the information and simply reads it.

This information is physically stored in temporary arrays, which is why hash tables are also known or referred to as "associative arrays". Hash tables are then the first truly runtime dynamic, memory-resident Data step structure.

This idea is not totally new to the SAS data step environment as other methods are currently available for using the direct access approach. By using SAS arrays and direct file access with the POINT= option an advanced programmer can create very fast and efficient lookup methods and also their own hash tables.

In the next few sections we will describe the evolution of hash tables in SAS and detail its current use in SAS 9.

CONCEPT OF HASH TABLES

THEORY

The concept behind a hash table is a temporary array holding in memory associate keys with some values (also referred to lookup values). In a merge situation, an associate key refers to the variables present in the data sets to be combined, which fully identifies an observation. In this situation the lookup values refer to the satellite information we are trying to add.

Nowadays, searching methods based on direct addressing techniques are referred as "hashing", although in reality the term hashing denotes just one of the direct addressing techniques.

On the next section we will find out how to start using direct addressing techniques and how we can build our own hash tables without using the new built-in feature.

DIRECT ADDRESSING

Most traditional methods are comparison based methods so for example, if we need to find a specific value contained in a SAS data set we will have to search through all its observations. Consider the following example in which we have a simple data set containing a key and lookup values:

hash.sas

```
data key_value;
  input key value $;
  cards;
1 a
2 b
3 c
4 d
5 e
6 f
7 g
8 h
9 i
run;
```

Normally if we are interested in retrieving the lookup information for the key value 7, we will need to create a program following the structure:

hash.sas

```
%let find=7;

data _null_;
  set key_value;
  if key=&find then do;
    put 'Found it in iteration '
      _n_ 'and has value ' value;
    stop;
  end;
run;
```

This program searches all values contained in the data set until it finds a valid match using standard conditional processing. Only when a match has been found can the data step processing stop.

```
Log - (Untitled)
227  data _null_;
228    set key_value;
229    if key=&find then do;
230      put 'Found it in iteration '
231        _n_ 'and has value ' value;
232      stop;
233    end;
234  run;

Found it in iteration 7 and has value g
NOTE: There were 7 observations read from the data set WORK.KEY_VALUE.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Looking through the log file we can see that only after the 7th iteration the data step is able to find a match for our value and stop.

In using a direct addressing method, what we are eventually looking for is to achieve the same result (obtain the lookup value for key 7), but without having to read the previous 6 observations (or using a sequential read).

In the following example we are going to use the values from the key variable to create a temporary array named *key_array*. This array will hold the lookup information, and the position where to store them will be based on the associated key value.

```
hash.sas
data _null_;
  array key_array(9) $ _temporary_
    ('a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i');
  if key_array(&find) ne '' then
    put 'Found it too! but this time at ' _n_
        'and has value ' key_array(&find);
  else put 'It is not there';
run;
```

The *key_array* will therefore hold the following information:

Element	1	2	3	4	5	6	7	8	9
Value	a	b	C	d	e	f	g	h	I

The use of key variables to denote the position the lookup values hold in an array is the key to direct addressing. By using this new method we don't need to loop through the data set finding the right value, as in this case we go and grab the value stored in the 7th position of the array.

```
Log - (Untitled)
235  data _null_;
236    array key_array(9) $ _temporary_
237      ('a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i');
238    if key_array(&find) ne '' then
239      put 'Found it too! but this time at ' _n_
240          'and has value ' key_array(&find);
241    else put 'It is not there';
242  run;

Found it too! but this time at 1 and has value g
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

As shown on the log window, we can retrieve the same value but in this case the data step has only performed one iteration.

This method for direct accessing information is also called key-indexing.

KEY-INDEXING

Lets progress by using another easy example. In this case we have a data set containing a list of all students in a class, and a second data set containing all those students who have passed their maths A level exam.

```
hash.sas
data students;
  input student_id name $;
  cards;
1 John
2 Mary
3 Peter
4 Charlie
5 Sarah
6 Kate
7 Matthew
8 David
9 Clare
run;
```

```
hash.sas
data a_results;
  input student_id result $;
  cards;
3 A
5 B
6 A+
8 C
run;
```

On the previous direct addressing example, in order to retrieve the information I first had to define the temporary array with its lookup values hard coded. As this is not a very realistic situation we need to replace this step with a dynamic load of all the lookup values into an array.

This is achievable by using do loops and using the END= option in order to flag when the last observation of the table has been reached. The next section of data step code uses a sequential read to populate the *grades* array with the results information.

```
hash.sas
  array grades(9) $  _temporary_;
  do until(eof);
    set a_results end=eof;
    grades(student_id)=result;
  end;
```

By using the student_id values, we can populate the *grades* array as:

Element	1	2	3	4	5	6	7	8	9
Value			A		B	A+		C	

In order to perform a successful lookup, we now read all the values from the students data set and using the student_id key variable retrieve the adequate lookup information from the *grades* array.

hash.sas

```
data students;
  array grades(9) $  _temporary_;
  do until(eof);
    set a_results end=eof;
    grades(student_id)=result;
  end;

  do until(eof2);
    set students end=eof2;
    result=grades(student_id);
    output;
  end;
run;
```

This is called direct addressing by using the key-indexing technique. We use the value of the student_id variable as an index to that table, giving us the desired results:

VIEWTABLE: Work.Students

	student_id	result	name
1	1		John
2	2		Mary
3	3	A	Peter
4	4		Charlie
5	5	B	Sarah
6	6	A+	Kate
7	7		Matthew
8	8	C	David
9	9		Clare

This is probably one of the fastest and simplest methods of performing a table lookup, outperforming MERGE statements by a factor of 5 **[1]**.
But although this method is very powerful in many cases, it also has a couple of major limitations, these being memory size and index size.
Memory becomes an issue when the lookup data to be added to the array gets larger and larger.
But the major limitation in this case is index size. Imagine using this method when the key variable (in this case student_id) has 12 digits, as many bank account numbers may have. The memory requirement will scale up as SAS will need to hold in memory 1 to 999,999,999,999 positions with its correspondent lookup values.

Lets see what would happen if we modify the student_id's to hold 12 digits values

hash.sas

```
data students;
  input student_id name $;
  cards;
100000000001 John
200000000001 Mary
300000000001 Peter
400000000001 Charlie
500000000001 Sarah
600000000001 Kate
700000000001 Matthew
800000000001 David
900000000001 Clare
run;
```

and also update the a_results data set

hash.sas

```
data a_results;
  input student_id result $;
  cards;
300000000001 A
500000000001 B
600000000001 A+
800000000001 C
run;
```

If we now execute the above key-indexing code, but increase the size of the array that holds the information so it can include the 12 digit student id number,

hash.sas

```
data students;
  array grades(100000000001:9999999999999) $
    _temporary_;
  do until(eof);
    set a_results end=eof;
    grades(student_id)=result;
  end;

  do until(eof2);
    set students end=eof2;
    result=grades(student_id);
    output;
  end;
run;
```

what we find when we run our example is the following error message in the log window:

Log - (Untitled)

```
ERROR: An unknown, abnormal error has occurred. This step is being terminated.
NOTE: The SAS System stopped processing this step because of errors.
407  data students;
408    array grades(100000000001:9999999999999) $
409      _temporary_;
410    do until(eof);
411      set a_results end=eof;
412      grades(student_id)=result;
413    end;
414
415    do until(eof2);
416      set students end=eof2;
417      result=grades(student_id);
418      output;
419    end;
420  run;
```

This error message is produced because the computer has run out of memory in the process of loading all the array variables.

Now key-indexing can be extremely useful and fast when we are dealing with "small" numbers to hold our array positions, but if we want to use this method for "big" numbers we start finding problems. So where do we go from here? I'm afraid the only answer is hashing!

HASHING

In these sorts of cases Hashing can save the day. We need to obtain a number which is lower than the huge combination presented above. Hashing is the process of converting a long-range key (can be either numeric or character) into a smaller-range integer value using a mathematical algorithm or function call HASH.

As seen on the previous example too, although the student_id numbers are quite large most of them are also empty. SAS needs to load a huge number of array elements although only 9 of them are populated.

We need to reduce the number of empty buckets or positions present in the array. This is achieved by using a function that provides us with a smaller integer value for the student_id.

Hash functions can get awfully complicated but for our student example we can actually create our own one. We could use a division to shorten the number and the ROUND function to make sure that only integer values will be used to populate positions in the array:

Hash Function=round(student_id/100000000000,1)

And by applying this function to the key value

Key value → hash function → new array address

We will finally obtain

900000000001→ round(900000000001/100000000000,1) → 9

a much reduced number which will be used as the address for the lookup value in the array.

Using this hash function we can now update the previous code

hash.sas

```
data students;
  array grades(9) $ _temporary_;
  do until(eof);
    set a_results end=eof;
    grades(round(student_id/100000000000,1))=result;
  end;

  do until(eof2);
    set students end=eof2;
    result=grades(round(student_id/100000000000,1));
    output;
  end;
run;
```

which now has no memory problems and can produce the lookup table

Log - (Untitled)

```
283  data students;
284  array grades(9) $ _temporary_;
285  do until(eof);
286    set a_results end=eof;
287    grades(round(student_id/100000000000,1))=result;
288  end;
289
290  do until(eof2);
291    set students end=eof2;
292    result=grades(round(student_id/100000000000,1));
293    output;
294  end;
295  run;

NOTE: There were 4 observations read from the data set WORK.A_RESULTS.
NOTE: There were 9 observations read from the data set WORK.STUDENTS.
NOTE: The data set WORK.STUDENTS has 9 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

So eventually we can create an array with 9 elements that holds the desired information and then it will work as we saw before in our key-indexing example when the student_id only contained a 1 digit number.

But what will happen if we have two students with id numbers 100000000001 and the other one with 100000000002? By applying our hash function the position to store the information in the array is the same (in this case position 1). These situations are also referred to as collisions.

COLLISION THEORY

Although it is not the aim of the paper to discuss the theory behind collisions we can briefly mention the two approaches we are likely to find:

Linear Probing is the first and simplest approach available. This method works by simply looking for the next free space available in the array to fit in the information. When a collision takes place and a bucket has already been populated with the lookup values, the linear probing method will check if the previous bucket is empty. If it is free it will store the information in it. If on the other hand the previous bucket is also populated it will check the one before, and so on until it finds an empty one. If it reaches the beginning of the table without an empty space it will start searching again from the last position of the array.

If we are half way through loading an array with its lookup values as:

Element	1	2	3	4	5
Value				Value1	Value2

and the value returned by the hash function tells us that next position to use is 5, Linear probing will work as:

Element	1	2	3	4	5
Value			**Value3**	Value1	Value2

First it tries to include the value at position 5. As it is already in use it moves to position 4, which is also in use so it moves to position 3. As this position is empty it will include the lookup value in this bucket.

As a rule of thumb, linear probing performs best if about half of all nodes in the table are left empty and its performance rapidly deteriorates as the table gets fuller.

Separate chaining is another method of dealing with collisions and it is a more complete one. In this method a second array is introduced to hold information regarding which positions had a collision and where the collision information has been stored.

So considering the same example, if we have a look at a hash table half way through loading information:

HASH ARRAY

Element	1	2	3	4	5
Value				Value1	Value2

With separate chaining there will be a second array holding information regarding which positions have already been filled. In this case I am calling that array 'Position'

POSITION ARRAY

Element	1	2	3	4	5
Value	.	.	.	0	0

The 0 denotes that a position has already been taken in the array and a missing value denotes that it is still available.

If now we have another element which needs to be loaded and the hash function returns position 5, the separate chaining method will first check the Position array. If it finds that the space is already taken, it will search backwards for the nearest empty position. In this example that will be position 3. It will change the value of bucket 3 from . to 0 (to denote that the position has been taken) and modify the value of bucket 5 (the original collision bucket) to refer to the bucket number holding

the secondary information, leaving us with an updated array as:

POSITION ARRAY

Element	1	2	3	4	5
Value	.	.	0	0	3

And

HASH ARRAY

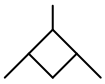
Element	1	2	3	4	5
Value			**Value3**	Value1	Value2

Although you can create your own hash tables by identifying a hash function that works well enough for the type of data you are trying to access and deals with collision situations, the coding around it is not trivial and requires considerable thought. For anybody interested in creating hash tables before SAS 9 and who would like a better understanding of the collision theory please refer to paper **[3]** for examples.

Luckily for most of us, hash tables in SAS 9 include an inbuilt hash function and also a facility to deal with collisions.

Collisions in SAS 9 are solved using the AVL (Adelson-Volsky & Landis) approach. The key is passed to a hash function which returns a bucket number where the data can be found. When multiple keys hash to the same bucket, the key pairs are stored in an AVL tree. AVL trees are binary trees holding the lookup information and we can imagine them as:

HASH ARRAY

Element	1	2	3
Value			

HOW TO CREATE HASH TABLES IN V9

Now we are familiar with some of the background information and evolution of hash tables, this section will use SAS 9 terminology to create hash tables in the data step environment.

Following on from the student example we will introduce some of the new syntax for creating a hash table in SAS 9.

hash.sas

```
data results;
  set a_results point=_n_; ①
  declare hash ht (hashexp:16, dataset:'a_results');②
  ht.defineKey('student_id'); ③
  ht.definedata('result'); ④
  ht.definedone();⑤

  do until (eof);
    set students end=eof;⑥
    if ht.find()=0 then output results;⑦
  end;
  stop;
run;
```

To define and use the hash table the following steps are identified:

First we need to load the lookup table into memory. But now, instead of manually coding the temporary array we use the SAS 9 notation.

SAS needs to load the properties of the variables present in the lookup data set. I could include a LENGTH statement to specify the properties or read the properties with a SET statement.
We are going to use the SET statement, as it requires less coding and has fewer chances of making a mistake. There are 3 different statements that we can use to read the variable properties:

```
set a_results point=_n_;
*or;
set small(obs=1);
*or;
if 0 then set small;
```

but it will not work on:

```
set small(obs=0);
```

② We then declare the hash table by using the DECLARE statement (you can also use the short version DCL) followed by the word HASH to denote hash table (you can also include the word ASSOCIATIVEARRAY) followed by the name of the hash table you want to create, in this case 'ht'.

In brackets after the name of the hash table we can also include two arguments:

- DATASET. Name of the data set holding information to load
- HASHEXP. Is the hash object's internal table size, where the size of the hash table is 2^n. The maximum allowed is 16, and it means 2^{16} buckets. Note that the hash table size is not equal to the number of items that can be stored as one bucket will hold an AVL tree.

③ We can then declare the key variables using the DefineKey method. For this example we only have one key variable (student_id), but if we had any more all we would have to do is list them separated by a comma. The DefineKey method can accept character variables as SAS will convert the text into a number that later will be used to populate the hash table.

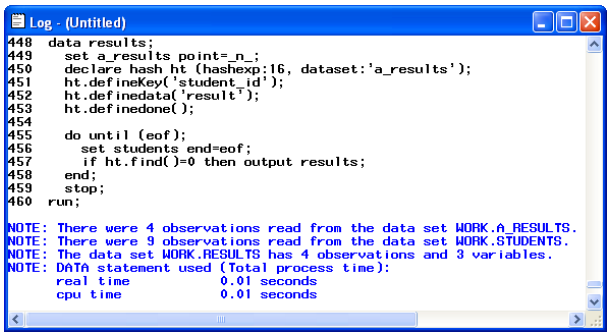
④ Use the DefineData method to declare which variables are going to be the lookup values. Again if more than one value is required list them separated by a comma.

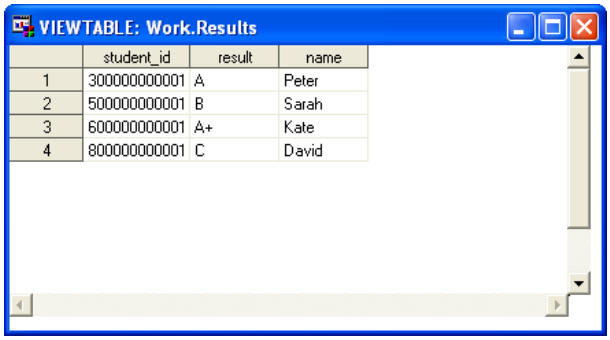
⑤ And once the properties of the data have been defined, we need to close the definition of the hash table by using the DefineDone method.

In the second section of the data step code the direct accessing takes place.

⑥ We read the observations coming from the students data set.

⑦ And using the Find method to access the hash object, obtain the lookup information. The Find method returns the value of 0 when if finds a match and it is only for these cases that we output the information.





What happens if our key variable that creates the hash table is not unique? The default behaviour when creating the hash table is that only the first value will be saved for the lookup, and any other occurrences will be ignored as that bucket is already populated.
There are not many options to change this behaviour, although we can use the Replace method if we want to keep the information provided by the last occurrence of a given key.

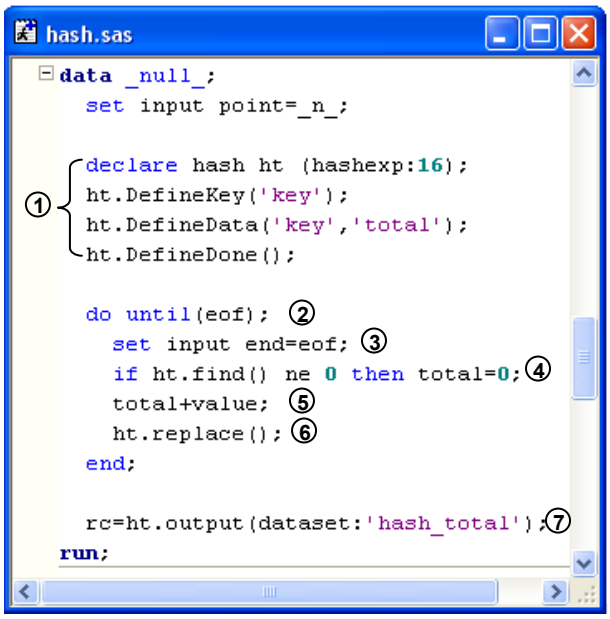
V9 hashing does not provide a mechanism for storing and/or handling duplicate keys with different lookup values in one and the same hash table. This difficulty can be circumvented by creating a secondary key, making the entire composite key unique.

When coding your hash tables there are a few more methods that can be used:

- Add: Obtains key and lookup values from the table and loads information into the hash object. If you don't specify a data set argument when defining the table the items will need to be loaded.
- Remove: Removes key and lookup values from the hash object table
- Check: Searches for a key value in the hash object and if found returns the value of 0
- Output: Saves the hash object into an output data set
- Replace: Replaces the lookup values with the latest read for a duplicate key value

USING HASH TABLES FOR SUMMARISATION

Hash tables can also be used for data summarisation. In this example we have a data set called 'input' which contains a key variable (key) and a numeric variable (value), which we want to summarise. The following code shows an example of the use of hash tables for summarising data:



① First we declare our hash table 'ht', but in this case note that we don't populate the hash table with any values. This is because 'ht' will hold the results of summarising the information of *value* for each *key*. As the only information I want to keep in the output data set is the *key* and the *total*, I only need to define *key* in the DefineKey method, and *key* and *total* (new variable holding the result) for DefineData.

② Start the loop that will read the observations until it reaches the last observation of the data set.

③ Read the observation from the input data set.

④ Use the Find method to initialise the *total* variable to zero every time the key value changes (every subgroup gets initialised). Remember that the Find method returns the value of 0 when it finds a match for the key and 1 if it doesn't find any more values.

⑤ Use accumulator statement to add the values

⑥ Use the Replace method so for each unique key value only the last calculated total value gets added into the ht hash table.

⑦ And finally use the Output method to write the contents of the 'ht' hash table into the output data set called 'hash_total'.

A hash method approach to data summarisation can be much more efficient that using summarisation procedures such as proc summary or proc means.

The advantages become more noticeable when we have to calculate totals over classification variables holding a high number of subgroups.

HASH ITERATOR

The second new type of SAS 9 object available in the data step environment is the hash iterator object. The hash iterator enables you to retrieve the hash object data in forward or reverse order key (so the results are ordered!).

To use it we first need to declare the iterator object. Use a DECLARE statement as previously seen for the hash table but now declare your new object as HITER. Include a name for the object and specify the hash table associated with it.
(Note: An iterator object will always have to be declared after a hash table)

For our test student data we have rearranged the observations present in the students data set to see our ascending ordered result.

hash.sas

```
data students;
  input student_id name $;
  cards;
200000000001 Mary
400000000001 Charlie
500000000001 Sarah
100000000001 John
800000000001 David
600000000001 Kate
700000000001 Matthew
900000000001 Clare
300000000001 Peter
run;
```

The new code using the iterator operator will be:

hash.sas

```
data sorted_students;
  set students point=_n_;
  declare hash ht (dataset:'students',
                   hashexp:16,
                   ordered: 'a' );
  declare hiter hi ('ht');
  ht.defineKey('student_id');
  ht.definedata('name','student_id');
  ht.definedone();

  do rc=hi.first() by 0 while (rc = 0);
    output;
    rc=hi.next();
  end;
  stop;
run;
```

① Note that apart from declaring the hash object table with the 'dataset' and 'hashexp' arguments, we have added a new parameter called *ordered*. The new *ordered* parameter can take the values "a", "y" or "ascending" to denote ascending mode, and "d" or "descending" for descending mode.

② Declare the iterator operator 'hi' and associate it with the hash table 'ht'

③ To sort the table we use the First method. This method allows us to access the element with the smallest key value from the array. Once we have found the first ordered observation, we output it to the SAS data set, and use the Next method to retrieve the next element. The Next method will obtain values in ascending order until we run out of elements and the result of fetching the next item returns the value of 1.

Notice the inclusion of the key variable student_id also in the DefineData method for the hash table. As the do loop performs its iteration, the Next method goes through the values defined as lookup values but without updating the value for the key variable. By adding the key variable into the DefineData method both sets of information get updated.

The iterator operator also has the following methods:

- Prev(): To obtain the previous value from a hash table
- Last(): To obtain the last value of a hash table.

So to order the data in descending mode we will need to use something like:

hash.sas

```
data d_sorted_students;
  set students point=_n_;
  declare hash ht (dataset:'students',hashexp:16,ordered: 'a' );
  declare hiter hi ('ht');
  ht.defineKey('student_id');
  ht.definedata('name','student_id');
  ht.definedone();

  do rc=hi.last() by 0 while (rc = 0);
    output;
    rc=hi.prev();
  end;
  stop;
run;
```

In the above case the code is using the inverse methods Last and Prev, but we could also use the First and Next methods and change the argument *ordered* in the hash table definition to 'descending'.

PERFORMANCE TIMES COMPARISON

The advantage of using hash tables and iterators is their performance time.

If we are interested in improving our time when performing table lookups have a look at the following results table:

Method	Time
Merge statement with index generation	8:37.52 mins
Merge statement and 2 proc sorts	6:16.84 mins
SQL	6:40.98 mins
Key=	1:17.06 mins
Formats	23.86 secs
Hash table	**32.34 secs**

These tests were performed on a 1 million record data set (containing 109 variables) and using a lookup table with 100.000 key values with 1 satellite variable. Tests were executed on Windows XP, Pentium 4 2.80 GHz and 512 MB of RAM.

As illustrated on the results table hash tables and formats are just a lot faster that any other of the traditional comparison methods.

Although using formats lookup methods is actually faster than hash tables for this given example, using formats has the limitation of only being able to define one variable as the key variable, whereas using hash tables the key variables specified can be multiple.

If the data we are trying to combine contains multiple keys hash tables is our best approach. There is still a considerable gap between using a hash method and other direct access such as KEY= option.

If we are comparing times for sorting information, the results obtained are:

Method	Time
Proc sort	5:20.48 mins
Iterator object	2:10.09 mins

Again quite an astonishing difference between the two methods in which the use of hash tables can more than half the time taken by the traditional method for ordering data.

When hash tables are used for data summarisation, the results obtained:

Method	Time
Hash table	4.46 secs
Proc summary	10.88 secs

Also reflecting the full power of using hash tables.

CONCLUSION

It has already been proven that direct addressing techniques are really efficient when performing table lookups. We have also seen that some of these techniques can be used for other areas such as data summarisation and data ordering.

Before SAS 9 only a few advanced programmers were able to use these extremely useful techniques. Although achievable, programmers had to use some very advanced code, which had to cover all possible eventualities (including the painful process of dealing with collisions).

With SAS 9, hash tables are made available using a much more simplified approach, as they are now fully integrated in the data step environment. The incorporation of easy to use methods that allow us to obtain maximum benefit from hash tables and iterators has to be a winning approach (as there is also minimum pain involved when it comes to code it!).

Hopefully the performance results obtained will convince users of the benefits and put hash tables in the spotlight so their use will increase within the SAS community.

REFERENCES

[1] “Table Lookup by Direct Addressing: From V8 to V9” by Paul M. Dorfman
http://www/nesug.org/html/Proceedings/nesug03/hw/hw002.pdf
[2] “Data Step Programming Using the Hash Objects” by Paul M. Dorfman and Lessia S. Shajenko
http://www.nesug.org/html/Proceedings/nesug04/pm/pm06.pdf
[3] “Hashing Rehashed” by Paul M. Dorfman and Gregg P. Snell
http://www2.sas.com/proceedings/sugi27/p012-27.pdf
[4] “The DATA step in Version 9: What’s New?” by Jason Secosky
http://support.sas.com/rnd/papers/sugi27/dsv9-sugi.pdf
[5] SAS 9 Online Help
http://support.sas.com/onlinedoc/913/docMainpage.jsp

CONTACT INFORMATION

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

Author Name	Elena Muriel
Company:	Amadeus Software Limited
Address:	Mulberry House, 9 Church Green, Witney, Oxon OX28 4AZ
Work Phone:	+44 (0) 1993 848010
Email:	Elena.Muriel@amadeus.co.uk
Web:	www.amadeus.co.uk