Paper SD04

Creating Data-Driven Data Set Names in a Single Pass Using Hash Objects

Jack Hamilton, Kaiser Foundation Health Plan, Oakland, California

ABSTRACT

Although it's not usually good SAS® coding practice, external requirements sometimes demand that you create a series of SAS data sets, each with a name derived from a data value (such as STATES.CA containing data for California, STATES.KY containing data for Kentucky, and so forth, created from a master table containing data for all states). A common solution is to make two passes through the input data: one to collect the different values needed for the names, and another to reread the input and write the output data sets. This paper presents a different method, using the new data step Hash Object to create multiple output data sets with a single pass through the input data. Although this method is not suitable for all occasions, it is a useful tool to have in your arsenal. The use of a DOW loop to simplify BY-group processing is also discussed.

INTRODUCTION

This paper describes how to create multiple output datasets, with names based on data values, using a single pass through the input data set. The (no longer) new data step hash objects are used to store and output records. A DOW loop is used to simplify BY-group processing. The techniques that would have been used in older versions of SAS are also briefly described.

This paper is divided into 8 sections:

- Description of the problem to be solved
- Sample data
- A quick description of the solution
- The DOW loop
- Using hash objects
- Putting it all together
- Older methods of achieving the same results, and timing comparisons
- Conclusion

I will create a new version of this paper approximately a week after the conference. It will include answers to questions asked during the session, and will contain a more detailed description of the DOW loop. Please look for it at <www.excursive.com/sas>. You will also be able to download the SAS code used in this paper.

DESCRIPTION OF THE PROBLEM TO BE SOLVED

It is sometimes desirable to take an existing SAS data set (which I'll call the master data set) and create multiple output data sets from it, with the names of the output data sets based on a value in the master data set. In this paper, I will use a master data set containing a US state postal code, and create a separate output data set for each state.

This is not something that you will usually need to do if you are processing your data entirely in SAS – in that case, you would typically use BY-groups, possibly in combination with indexes, to operate on your data one BY-group at a time.

The need for separate data sets most often arises when the data sets are going to be exported to another program which doesn't have the sophisticated processing capabilities of SAS. You might also need separate data sets in SAS when using macros which have not been written with BY-groups in mind, or when using PROCS that don't handle BY-groups in the way you would like.

SAMPLE DATA

In the main part of this paper I will use a small data set with three different values of the state code, and a few numeric variables that we don't really care about (they're there because most real-life data sets contain more than one variable). There aren't many observations in the data set, to make it easier to show what's happening. The Appendix uses a much larger data set.

In the sample, all of the data with a particular state code are in consecutive observations, but the state codes themselves are not in any particular order.

Here's the code to create the sample data set:

```
data have;
   do state = 'KY', 'CA', 'MS';
      do b = 1 to 3;
         do c = 200, 100;
            output;
         end;
      end;
   end;
   stop;
run;
```

Here are the first few rows of data:

Data Set HAVE

Obs	state	b	c
1	KY	1	200
2	KY	1	100
3	KY	2	200
4	KY	2	100
5	KY	3	200
6	KY	3	100
7	CA	1	200
8	CA	1	100
9	CA	2	200
10	CA	2	100
11	CA	3	200

What we want is three output data sets, named (for example) ex1_KY, ex1_CA, and ex1_MS, with each output data set containing all and only the observations for its state code.

A QUICK DESCRIPTION OF THE SOLUTION

The solution presented in this paper is to loop through the master data set. When the first observation for a state code is reached, a hash object will be created. Subsequent observations add entries to the hash object. When the last observation for a state is read, the data in the hash object will be written to an appropriately named output data set, and the hash object deleted. This process is repeated for each group of data.

Here's the code:

```
data _null_;

   declare hash group;

   group = _new_ hash(ordered: 'ascending');
   group.definekey('_unique_Key');
   group.definedata('State', 'b', 'c');
   group.definedone();

   do until (last.state);
      set have;
      by state notsorted;
      _unique_key + 1;
      group.add();
   end;

   group.output(dataset: 'ex1_' || state);

   rc = group.delete();

run;
```

THE DOW LOOP

One unusual thing you might notice about the solution is the *DO UNTIL (LAST.state)* loop in the middle of the data step. Rather than using the standard SAS technique of one iteration through the data step per observation, this loop lets us use one iteration per BY-group. This makes handling the group start and end conditions much easier. The standard technique would require putting the hash initialization and termination code inside IF/THEN/DO groups, which is longer, less efficient, and (in my opinion) harder to follow.

According to SAS-L folklore, the DOW loop was first discovered by Ian Whitlock – hence the name, which is an abbreviation for ***D***o loop ***O***f ***W***hitlock. It was then publicized and promoted by Paul Dorfman on SAS-L and at user group meetings. I wish I had discovered it, because I think it's (a) brilliant, and (b) obvious once you think about it, but alas, I can't take credit.

Two sources for more information about the DOW loop are Paul's paper *The Magnificent Do*, available from <http://www.devenezia.com/papers/other-authors/sesug-2002/TheMagnificentDO.pdf>, and Ron Fehd's paper *Do Which? Loop, Until or While? A Review Of Data Step And Macro Algorithm*, available from <http://www2.sas.com/proceedings/forum2007/067-2007.pdf>.

The DOW loop may be obvious once you think about it, but it's not obvious at first glance, so an explanation is warranted. Here is the program above with some PUTLOG statements added for tracing, and only one state selected (to make the log shorter), followed by the corresponding SAS log. Line numbers have been added for easy reference (1-18 for the program, 101-112 for the log). The two papers mentioned above will also help you to understand what's happening.

```
1   data _null_;
2      declare hash group;
3      putlog 'At top';
4      group = _new_ hash(ordered: 'ascending');
5      group.definekey('_unique_Key');
6      group.definedata('state', 'b', 'c');
7      group.definedone();
8      do until (last.state);
9         set have (where=(state = 'KY'));
10        by state notsorted;
11        _unique_key + 1;
12        group.add();
13        putlog _unique_key=2. @17 state= b= c= +1 last.state= +1 _n_= ;
14     end;
15     putlog 'At bottom';
16     group.output(dataset: 'ex2_' || state);
17     rc = group.delete();
18  run;

101 At top
102 _unique_key=1   state=KY b=1 c=200  last.state=0  _N_=1
103 _unique_key=2   state=KY b=1 c=100  last.state=0  _N_=1
104 _unique_key=3   state=KY b=2 c=200  last.state=0  _N_=1
105 _unique_key=4   state=KY b=2 c=100  last.state=0  _N_=1
106 _unique_key=5   state=KY b=3 c=200  last.state=0  _N_=1
107 _unique_key=6   state=KY b=3 c=100  last.state=1  _N_=1
108 At bottom
109 NOTE: The data set WORK.EX2_KY has 6 observations and 3 variables.
110 At top
111 NOTE: There were 6 observations read from the data set WORK.HAVE.
112       WHERE state='KY';
```

So what's happening here? For each BY-group, we initialize a hash object (we'll cover the hash functions later). Then we write each observation to the hash using the loop. Finally, after we've processed all the observations for a group, we write out the hash and delete it.

- When SAS starts processing the data step for the first time, lines 1-7 are executed, printing line 101.
- When line 8 is executed for the first time, the DO condition is ignored because it's in an UNTIL loop, which will always execute at least once.
- When line 9 is executed, data from the first observation is brought into the Program Data Vector. Line 11 increments a counter, which provides a unique value for the hash. Because it's incremented when a read is done, it also shows us the current observation number in the input data set. At line 13, some information about the current state is printed.
- When line 14, the end of the loop, is reached, execution returns to line 8. The loop in lines 8-14 is repeated 5 times (for a total of 6 executions), printing lines 102-107.
- After the sixth time line 9 is executed, LAST.STATE becomes true. SAS always does a lookahead read, and knows that the value of STATE will be different the next time the input data set is read. The UNTIL condition is now false, so execution continues at line 15. Line 108 is printed, and a new iteration of the data step will begin, as shown by line 110.

USING HASH OBJECTS

You can think of a hash object as being like an array, with a few important differences:

- You can create and delete hash objects, or change their size, during data step execution. Arrays are fixed in size when the data step is compiled.
- The data in hash objects are always referenced through functions (also called methods). Array elements can be referenced directly.
- The hash object can read and write SAS data sets directly. Arrays must be explicitly populated using code. (This papers demonstrates writing data sets from hash objects, but not reading data sets directly into hash objects.)
- Arguments to the hash object functions are usually expressions evaluating to names of data sets or variables, as strings, rather than the data sets or variables themselves. This means that you can dynamically calculate, among other things, the name of the output data set created by a hash object.

  There's no equivalent capability with arrays, but there are some other data step functions that accept variable names as strings. One example is VLABELX('MYVAR') compared to VLABEL(MYVAR).

This paper uses the DECLARE statement plus 7 of the hash object methods: _new_ hash(), definekey(), definedata(), definedone(), add(), output(), and delete(). I will explain each of these in turn.

Keep in mind that I'm not describing everything that can be done with these functions (or methods) – I'm describing only what's needed for this paper. The online documentation for the hash objects is quite good, and describes everything. For SAS 9.1.3, start at *Using the Hash Object* in the Language Reference Concepts manual <http://support.sas.com/onlinedoc/913/getDoc/en/lrcon.hlp/a002585310.htm> and *DATA Step Object Attributes and Methods* in the Language Reference Dictionary <http://support.sas.com/onlinedoc/913/getDoc/en/lrdict.hlp/a002588142.htm>.

DECLARE HASH

DECLARE is a relatively new statement in SAS. It is used to tell SAS that a variable is a component object of some kind. The statement `declare hash group;` tells SAS that the variable GROUP is a hash object. The other types of component object are hash iteration objects and ODS objects. The other two types of data step variables, numeric and character, don't require (or allow) the DECLARE statement.

The DECLARE statement doesn't actually do anything at execution time – it just tells the compiler to expect more information later.

NEW HASH()

The _NEW_ statement is what actually creates a hash object. The statement `group = _new_ hash(ordered: 'ascending');` tells SAS to create a new hash object and assign it to the variable GROUP, which was previously declared as a hash object.

The arguments to component object methods have a different syntax than what you're used to seeing in the data step. Traditional functions have positional parameters: the first argument always has a particular meaning, as does the second, and so on. Object methods can be used in this way, but some methods also allow keyword parameters: you can specify arguments in any order, and you can omit arguments without coding placeholder parentheses, as long as you specify a keyword for every parameter.

HASH takes several possible arguments, but we're using only one, the ORDERED argument. This tells SAS that the hash object should return data in key order. If you don't use the ORDERED parameter, the data will be returned in whatever arbitrary order pleases SAS at the moment, which might or might not be the order you want. We want the output data sets to be in the same order as the master data set, so we specify ORDER: 'ascending'. The _UNIQUE_KEY variable is incremented by1 for each input observation, so it's a unique key, and in the same order as the input.

DEFINEKEY()

DEFINEKEY is used to tell SAS what the key, or ordering, variables are for the hash. The variables are listed by name. `group.definekey('_unique_Key');` tells SAS that there's one key variable, _UNIQUE_KEY. The key must be unique in the hash object – you can't have two entries with the same values for the key variable(s).

DEFINEDATA()

DEFINEDATA is used to tell SAS what variables should be included in the hash object. `group.definedata('State', 'b', 'c');` tells SAS to include variables State, B, and C. Notice that the _UNIQUE_KEY variable is *not* specified: in a hash, you can order by a variable that's not included in the data (it is unusual to have a key variable that's not also a data variable, but it does happen).

Instead of calling DefineData once with three variable names, I could have called it three times with one variable name each time. The result is the same, but one way or the other might be more convenient in your code. The example uses hard-coded variable names, but you could calculate them at execution time if you needed to do so.

DEFINEDONE()

DEFINEDONE is used to tell SAS that you have finished defining the key and data variables for the hash. Because you can call DefineData and DefineKey multiple times, SAS has no way to know when you're finished, so you have to tell it. DefineDone doesn't take any parameters; it's simply `group.definedone();`.

ADD()

ADD is used to add a new entry to the hash object. You've already told SAS which variables to use (UNIQUE_KEY for the key variable, and STATE, B, and C for the data variables), so `group.add();` is all you need. The current values for these variables are added to a new entry in the hash object.

You can explicitly list the values you want to add using KEY: and DATA: parameters, but that's not needed in this case.

OUTPUT()

OUTPUT tells SAS to write the values in the hash object to a SAS data set. You can specify either a literal or an expression, so you can calculate the output data set name at execution time (that's what makes this whole thing work). `group.output(dataset: 'ex1_' || state);` writes a different data set for each value of the STATE variable.

DELETE()

DELETE tells SAS that you've finished using the hash object. When you execute `group.delete();`, the memory used by the hash object is freed, and any subsequent references to it will result in an error. You can create another hash object using the _NEW_ statement.

PUTTING IT ALL TOGETHER

Now we can look at and understand the complete program:

```
1   data _null_;
2
3      declare hash group;
4
5      group = _new_ hash(ordered: 'ascending');
6      group.definekey('_unique_Key');
7      group.definedata('State', 'b', 'c');
8      group.definedone();
9
10     do until (last.state);
11        set have;
12        by state notsorted;
13        _unique_key + 1;
14        group.add();
15     end;
16
17     group.output(dataset: 'ex1_' || state);
18
19     rc = group.delete();
20
21  run;
```

Lines 3-8 are executed once for each by group. They create and define the hash object that will be used to hold the data.

Lines 10-15 loop through all the observations for a single BY-group. The code used to control the loop is what is commonly called a DOW loop. Each input observation is added to the hash object using the ADD method. A new variable, _UNIQUE_KEY, is used as the key for the hash; this allows us to keep the order of the input data.

Line 17 is executed after all the observations for a BY-group have been processed. It writes the data in the hash object to a SAS data set whose name is dynamically calculated.

Finally, Line 19 deletes the hash object and frees the memory it used.

Note: there are two important restrictions on the use of this technique: the data for a group must be consecutive in the input data set, and the data for a group must fit into memory – hash objects are not stored on disk if they get too large (this is unlike PROC SUMMARY, for example, which will write internal data to disk if there's not enough memory).

OLDER METHODS OF ACHIEVING THE SAME RESULTS, AND TIMING COMPARISONS

Older versions of SAS required using code-that-writes-code to create data sets with data-driven names. Two methods are shown here, but there are others.

Both methods create a data step which looks like this (for the sample data):

```
data ex3_CA ex3_KY ex3_MS;
   set have;
   select(state);
      when ('CA') output ex3_CA;
      when ('KY') output ex3_KY;
      when ('MS') output ex3_MS;
      otherwise;
   end;
run;
```

PROC SQL AND MACRO VARIABLES

In this technique, PROC SQL is used to create macro variables which resolve to the needed data step code:

```
proc sql noprint;
   select distinct
            'ex3_' || state,
            cats('when (', quote(state), ') output ex3_', state, ';')
   into
            :DATASETS separated by ' ',
            :WHENS    separated by ' '
   from
            have;
quit;

data &DATASETS;
   set have;
   select(state);
      &WHENS
      otherwise;
   end;
run;
```

Although this technique is less concise and may take more time to execute than the hash object technique, it (and the following method) does have three major advantages:

- It can be used in older versions of SAS that don't have hash objects.
- The data in a group don't have to be consecutive in the input.
- The data in a group don't have to fit into memory.

These techniques will take longer to run, because they require at least two passes through the input data set. The hash object technique requires only one. Depending on your data, this might be an important consideration.

Late addition: the source code file available on my web site contains an enhanced version of the hash object technique which allows an unsorted input file.

DATA STEP AND %INCLUDE FILES

In this technique, a data step is used to create source code files which are then %INCLUDEd:

```
filename source1 temp;
filename source2 temp;

data _null_;

   do until (last.state);
      set bigtest end=eod;
      by state notsorted;
   end;

   file source1;

   if _n_ =1 then
      put 'data ';

   put 'ex6_' state;

   if eod then
      put ';';

   file source2;
   put 'when (' state $quote4. ') output ex6_' state ';';

run;

%include source1;
   set bigtest;
   select(state);
      %include source2;
      otherwise;
   end;
run;
```

You could achieve similar results using CALL EXECUTE or CALL SYMPUT instead of creating source code files.

TIMING

I used the hash object method, the proc sql method, and the data step method on a large data set (2.21 million records) containing more numeric variables, and some large character variables:

```
data bigtest;
   do state = 'NH', 'NJ', 'NM', 'NY', 'IA', 'NV', 'OK', 'HI', 'AR', 'TX',
              'PW', 'DE', 'AL', 'WI', 'RI', 'FM', 'VI', 'VA', 'OR', 'UT',
              'MI', 'WA', 'ID', 'IN', 'MP', 'DC', 'KY', 'MD', 'AZ';
      do b = 1 to 300;
         do c = 1 to 300;
            d = ranuni(0); e = ranuni(0); f = ranuni(0); g = ranuni(0);
            h = put(b, roman10.); i = put(c, roman10.);
            length j k l $400.; j = ' '; k = ' '; l = ' ';
            output;
         end;
      end;
   end;
run;
```

Technique	Windows		z/OS		
	CPU Time	*Elapsed Time*	*CPU Time*	*Elapsed Time*	*EXCPs*
Hash object with DOW	13.54	2:22.32	23.27	1:53.34	23,870
Hash Object without DOW	13.26	3:28.29	24.02	2:23.81	23,877
SQL + data step	48.97	12:28.66	42.69	5:43.68	87,046
Data step + data step	31.01	8:46.33	18.25	4:58.04	60,904

As shown in the z/OS columns, the use of hash objects can greatly reduce both I/O and CPU usage, resulting in decreased elapsed time.

The enhanced version of the hash program which works with an unordered input data set uses slightly more CPU and elapsed time than the hash object without a DOW, but less than the older methods. It requires sufficient memory to store the entire input table and the largest output table in memory at the same time.

CONCLUSION

The use of the DOW loop and hash objects provides a fast and effective alternative to older methods of creating data sets with data-driven names.

CONTACT INFORMATION

Jack Hamilton
Kaiser Foundation Health Plan
1950 Franklin Street
Oakland, California 94612
+1 510 987-1556
Jack.Hamilton@kp.org
jfh@alumni.stanford.org
www.excursive.com/sas