

This Works on So Many Levels: Building Summary Tables in SAS® with Hash Programming

David V. Cole, Harvard Pilgrim Health Care Institute

ABSTRACT

Hash programming in SAS allows users to store and retrieve information from in-memory objects. While often thought of first as a means for more efficient table lookups, hash programming has much broader applications, in large part because it is used within DATA-step processing.

The creation of summary tables is a common task in SAS. While hash can be used to create simple summary tables via straightforward code, the programming techniques can be extended and combined with other DATA-step functionality to generate multi-level summaries within a single data step from one pass through the data.

This paper discusses the considerations behind using hash to construct various types of summaries and provides code to demonstrate how to use the power of hash in SAS to create flexible, efficient code for complex tasks.

INTRODUCTION

So you want to create a multi-level summary table in SAS.

You want to write and maintain a minimal amount of code while allowing for any number of summary levels. You only want to summarize certain levels for specified combinations of variables, and you may need to add or take away levels as the requirements of the summary evolve. You want code that will work on small and large datasets, and you want results that are easy to read and easy to process. In other words, you want a solution that is compact, flexible, and scalable.

SAS provides many tools for summarizing data. You could use PROC FREQ, PROC MEANS, PROC SQL, DATA step, or some combination of any or all of these tools. You probably have a go-to method already, one you are comfortable with and can code quickly. With this paper, I am proposing another potential solution: hash programming.

Hash programming in SAS has many uses beyond efficient joins with lookup tables. Data can flow in both directions. In other words, not only can information be retrieved from a hash table, it can also be added or replaced, and then the entire hash table can be output to a dataset when processing is complete. By combining these capabilities with DATA-step processing, I will show you how to create the compact, flexible, and scalable solution that you wanted.

METHOD

Let's begin with the basic method for creating a simple summary count with hash programming, which I've adapted from Dorfman and Henderson's 2015 paper on data aggregation [1]. Define a hash object to store running counts by a specified key variable, and then process each record in the input dataset to retrieve, recalculate, and update the running counts in the hash. When the input dataset has been fully processed, output the hash object to a dataset.

To build the code, start with data _null_ since you only want to output the hash table. Load descriptors from the input dataset and define a count variable so that all necessary variables are available to the hash. Declare an ordered hash object to produce a sorted summary table. Define the hash key with the specified key variable, and define the hash data with the key and count variables.

Use a DOW loop [2] to process the input dataset. Reset count to missing. Find the key in the hash to retrieve the running count, recalculate the count, and replace the key in the hash to update running count. Be aware that, if the key does not already exist in the hash, the .REPLACE() method will add a new record to the hash. After all records in the input dataset have been processed, output the hash table to a dataset.

The following is a template for the basic method:

```
* Begin with data _null_ to output only the hash object;
data _null_;
* Load descriptors from the input dataset...;
if 0 then set <input-dataset>;
* ...and define a summary count variable for use in the hash object;
length count 5;

* Declare an ordered hash object...;
dcl hash <hashname> (ordered:'Y');
*...with variable you want to summarize as hash key...;
<hashname>.definekey( '<key-variable>');
*...plus the key and count variables as hash data;
<hashname>.definedata('<key-variable>', 'count');
<hashname>.definedone();

* Use a DOW loop to process the input dataset;
do until (eof);
set <input-dataset> end=eof;
* Initialize count to missing;
call missing(count);
* Find the key in the hash to retrieve the running count value;
rc=<hashname>.find();
* Recalculate count;
count=sum(count,1);
* Replace the key in the hash to update the running count value;
rc=<hashname>.replace();
end;

* When finished with the input dataset, output the hash;
<hashname>.output(dataset:'<output-dataset>');
run;
```

With just this basic framework, we already have all of the essential elements. In the demonstration I will show how to modify and expand this method until we can produce a multi-level summary table with multiple statistics.

EXAMPLE SCENARIO

To demonstrate the method, I first need some data. I have chosen a scenario that draws from my background in public health and medical research. Suppose we have data on individuals who have received at least one dose of a fictional vaccine with a two-dose schedule, i.e., a primary dose and a booster dose. Given this scenario, we may wish to answer many different questions, such as:

- How many individuals received at least one dose?
- What is the average age of individuals receiving their primary dose?
- How many individuals get a booster dose?
- What is the average spacing between primary and booster doses?

We also want any or all of these answers to be stratified by other information such as the demographics and health characteristics of the individuals receiving vaccinations.

Our test dataset is named Have. The unique row represents each individual who received at least one dose of the vaccine. The dataset includes administration dates of primary and booster doses, along with other information on geographic region, age at primary dose, and indicators for pre-existing health conditions. Although I use a small number of variables in this example, you can imagine a dataset with

many more variables for additional information on demographics, health characteristics, dose administration, costs, adverse events, vaccine effectiveness, etc.

Here is a list of variables and attributes in the Have dataset:

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Format
5	age	Num	3	
3	boostdate	Num	4	DATE9.
6	cond1	Num	3	
7	cond2	Num	3	
8	cond3	Num	3	
2	dosedate	Num	4	DATE9.
4	geo	Char	1	
1	uniqueid	Num	5	

Table 1. From PROC CONTENTS output for the Have dataset

And here are the first 10 records in the Have dataset:

	uniqueid	dosedate	boostdate	geo	age	cond1	cond2	cond3
1	1	19OCT2021	09DEC2021	C	37	1	1	0
2	2	20NOV2023	.	B	46	0	0	0
3	3	03FEB2022	13MAR2022	C	54	0	0	0
4	4	19MAY2024	15JUL2024	A	39	0	0	0
5	5	24JUL2021	15SEP2021	A	58	0	0	0
6	6	07DEC2022	27JAN2023	C	35	0	0	0
7	7	06JUL2022	.	C	39	1	0	0
8	8	27NOV2022	.	C	26	0	0	0
9	9	14JUN2021	25JUL2021	B	52	1	0	0
10	10	13DEC2022	.	B	31	0	0	0

Table 2. First 10 records of randomly generated data in the Have dataset

Please see the Appendix for code used to create randomly generated data for the Have dataset.

DEMONSTRATION

We will start with the simple case of calculating summary counts for a single variable and then build up to a multi-level summary with multiple descriptive statistics.

For one-way summary counts of a single variable, we can fill in the code template from the Methods section using the details from our example scenario. Enter Have as the input dataset. Name the hash "ht" for hash table, just to keep things simple. To summarize the geo variable to get counts of individuals by geographic region, enter geo as both key and data. And since this is a one-way summary, name the output dataset "Want_OneWay".

Here is the code to create one-way summary counts of one variable:

```
* Begin with data _null_ to output only the hash object;
data _null_;
  * Load descriptors from the input dataset...;
  if 0 then set Have;
```

```

* ...and define a summary count variable for use in the hash object;
length count 5;

* Declare an ordered hash object...;
dcl hash ht(ordered:'Y');
*...with variable you want to summarize as hash key...;
ht.definekey( 'geo');
*...plus the key and count variables as hash data;
ht.definedata('geo','count');
ht.definedone();

* Use a DOW loop to process the input dataset;
do until (eof);
  set Have end=eof;
  * Initialize count to missing;
  call missing(count);
  * Find the key in the hash to retrieve the running count value;
  rc=ht.find();
  * Recalculate count;
  count=sum(count,1);
  * Replace the key in the hash to update the running count value;
  rc=ht.replace();
end;

* When finished with the input dataset, output the hash;
ht.output(dataset:'Want_OneWay');
run;

```

And here is the Want_OneWay output dataset:

	geo	count
1	A	422
2	B	294
3	C	180
4	D	104

Output 1. Want_OneWay output dataset from the code to create summary counts of one variable

To expand our summary of geographic region to include two of the pre-existing conditions, we only need to add the cond1 and cond2 variables to the hash key and data. Everything else remains the same, though we will use a different name for the output dataset to avoid overwriting the one-way summary. The resulting table contains summary counts by all combinations of geo, cond1, and cond2.

Here is the code to create summary counts of multiple variables:

```

* Begin with data _null_ to output only the hash object;
data _null_;
  * Load descriptors from the input dataset...;
  if 0 then set Have;
  * ...and define a summary count variable for use in the hash object;
  length count 5;

  * Declare an ordered hash object...;
  dcl hash ht(ordered:'Y');
  *...with variable(s) you want to summarize as hash key...;
  ht.definekey( 'geo','cond1','cond2');
  *...plus the key and count variables as hash data;
  ht.definedata('geo','cond1','cond2','count');

```

```

ht.definedone();

* Use a DOW loop to process the input dataset;
do until (eof);
  set Have end=eof;
  * Initialize count to missing;
  call missing(count);
  * Find the key in the hash to retrieve the running count value;
  rc=ht.find();
  * Recalculate count;
  count=sum(count,1);
  * Replace the key in the hash to update the running count value;
  rc=ht.replace();
end;

* When finished with the input dataset, output the hash;
ht.output(dataset:'Want_NWay');
run;

```

And here is the Want_NWay output dataset:

	geo	cond1	cond2	count
1	A	0	0	225
2	A	0	1	23
3	A	1	0	163
4	A	1	1	11
5	B	0	0	135
6	B	0	1	23
7	B	1	0	117
8	B	1	1	19
9	C	0	0	99
10	C	0	1	14
11	C	1	0	57
12	C	1	1	10
13	D	0	0	58
14	D	0	1	5
15	D	1	0	38
16	D	1	1	3

Output 2. Want_NWay output dataset from the code to create N-way summary counts of multiple variables

Note that, as the output dataset name reflects, we can only get N-way summary counts through this method. We no longer have totals by geographic region regardless of the two pre-existing conditions, nor do we have totals by either one or both of the pre-existing conditions. This is because the basic method has a big limitation: since all key variables are used to retrieve and replace information in the hash table, we can only calculate one summary level per hash table. No matter how many variables we add to the key, we can only ever produce an N-way summary.

We can overcome this limitation by taking a more general approach to the key variables. Instead of adding every variable we need to summarize, we instead use only two variables that, when combined, serve as the natural key for any specified combination of one or more variables. For this example, name these variables "level" and "sublevel", where level is assigned with the name(s) of the variable(s) being summarized and sublevel with the value(s) of the variable(s).

Back to our example scenario. To create a multi-level summary table, we need to define our general key variables level and sublevel. Be sure to define level and sublevel as character variables with sufficient

length to avoid truncation, and then we can replace geo, cond1, and cond2 with level and sublevel in the key and data definitions:

```
ht.definekey( 'level','sublevel');
ht.definedata('level','sublevel','count');
```

Before we try to retrieve the key from the hash, we need to assign values to level and sublevel. For our first summary level, assign level='geo', i.e., the variable's name, and sublevel=geo, i.e., the variable's value. From here the process remains the same. Reset count to missing. Find the key in the hash to retrieve the running count, recalculate the count, and replace the key in the hash to update the running count.

To add another summary level, simply repeat the entire process using different variables. To include levels with two or more variables, we can use * (asterisk) as delimiter for both the variable names and values. For our second summary level, repeat the entire process for level='geo*cond1' and sublevel=catx(*,geo,cond1), and for the final level, repeat again for level='geo*cond1*cond2' and sublevel=catx(*,geo,cond1,cond2).

Here is the code to create multi-level summary counts:

```
* Begin with data _null_ to output only the hash object;
data _null_;
  * Load descriptors from the input dataset...;
  if 0 then set Have;
  * ...and define key and count variables for use in the hash object;
  length level sublevel $20 count 5;

  * Declare an ordered hash object...;
  dcl hash ht(ordered:'Y');
  *...with variable(s) you want to summarize as hash key...;
  ht.definekey( 'level','sublevel');
  *...plus the key and count variables as hash data;
  ht.definedata('level','sublevel','count');
  ht.definedone();

  * Use a DOW loop to process the input dataset;
  do until (eof);
    set Have end=eof;
    * Assign values to the key variables;
    level="geo";
    sublevel=geo;
    * Initialize count to missing;
    call missing(count);
    * Find the key in the hash to retrieve the running count value;
    rc=ht.find();
    * Recalculate count;
    count=sum(count,1);
    * Replace the key in the hash to update the running count value;
    rc=ht.replace();
    * Assign values to the key variables;
    level="geo*cond1";
    sublevel=catx('*',geo,cond1);
    * Initialize count to missing;
    call missing(count);
    * Find the key in the hash to retrieve the running count value;
    rc=ht.find();
    * Recalculate count;
    count=sum(count,1);
    * Replace the key in the hash to update the running count value;
```

```

rc=ht.replace();
* Assign values to the key variables;
level="geo*cond1*cond2";
sublevel=catx('*',geo,cond1,cond2);
* Initialize count to missing;
call missing(count);
* Find the key in the hash to retrieve the running count value;
rc=ht.find();
* Recalculate count;
count=sum(count,1);
* Replace the key in the hash to update the running count value;
rc=ht.replace();
end;

* When finished with the input dataset, output the hash;
ht.output(dataset:'Want_MultiLevel_1');
run;

```

And here are the first 15 records from the Want_MultiLevel_1 output dataset:

	level	sublevel	count
1	geo	A	422
2	geo	B	294
3	geo	C	180
4	geo	D	104
5	geo*cond1	A*0	248
6	geo*cond1	A*1	174
7	geo*cond1	B*0	158
8	geo*cond1	B*1	136
9	geo*cond1	C*0	113
10	geo*cond1	C*1	67
11	geo*cond1	D*0	63
12	geo*cond1	D*1	41
13	geo*cond1*cond2	A*0*0	225
14	geo*cond1*cond2	A*0*1	23
15	geo*cond1*cond2	A*1*0	163

Output 3. First 15 records from the Want_MultiLevel_1 output dataset from the code to create multi-level summary counts

You can see the counts are summarized by only the specified combinations of variables. We have counts for geo alone because that was specified as a level by itself. We do not have counts for cond1 or cond2 alone because we did not specify either of them as separate levels. We have counts for geo*cond1 and geo*cond1*cond2 but not for geo*cond2 or cond1*cond2.

This generalized approach also lets us keep the summary table narrow because we no longer need to include all of the original variables. That is, no matter how many different variables from the input dataset we include across our summary levels, the summary table will always contain only the level, sublevel, and count variables.

This code has become very repetitive, and it will only become more so as we add more and more summary levels. Fortunately, we can replace these redundancies with some relatively simple code. If we assign a macro variable LIST_LEVELS with a space-delimited list of our summary levels, we can then replace repetitive code with a couple of iterative DO loops.

With the first loop, process each level in LIST_LEVELS. Instead of assigning level with a hard-coded value, we can use the SCAN() function to find the *i*-th element of LIST_LEVELS. Then we can use another iterative DO loop to assign sublevel by processing each variable named in level. We can use

CATX() to concatenate the asterisk-delimited values into sublevel, but if we simply used the SCAN() function to find the j -th element of level, we would get the variable name instead of its value. Instead, we need to nest the SCAN() function inside the VVALUEX() function, which returns a character string with the formatted value of the variable named in function's argument. If the variable being processed is numeric, VVALUEX() will convert the value to character using the variable's assigned format without requiring the PUT() function.

Now that we have assigned our key variables, we return to familiar territory. Reset count to missing. Find the key in the hash to retrieve the running count, recalculate the count, and replace the key in the hash to update the running count. Finally, output the hash to a dataset.

Here is the flexible code to create multi-level summary counts:

```
* Assign macro variable LIST_LEVELS with desired summary levels;
%let LIST_LEVELS=geo geo*cond1 geo*cond1*cond2;

* Begin with data _null_ to output only the hash object;
data _null_;
  * Load descriptors from the input dataset...;
  if 0 then set Have;
  * ...and define key and count variables for use in the hash object;
  length level sublevel $20 count 5;

  * Declare an ordered hash object...;
  dcl hash ht(ordered:'Y');
  *...with variable(s) you want to summarize as hash key...;
  ht.definekey( 'level','sublevel');
  *...plus the key and count variables as hash data;
  ht.definedata('level','sublevel','count');
  ht.definedone();

  * Use a DOW loop to process the input dataset;
  do until (eof);
    set Have end=eof;
    * For every specified summary level...;
    do i=1 to countw("&LIST_LEVELS",' ');
      * ...assign level from the LIST_LEVELS macro variable...;
      level=scan("&LIST_LEVELS",i,' ');
      * ...reset sublevel to missing...;
      call missing(sublevel);
      * ...and assign sublevel with values of all variables in the level;
      do j=1 to countw(level,'*');
        sublevel=catx('*',sublevel,vvaluex(scan(level,j,'*')));
      end;
      * Initialize count to missing;
      call missing(count);
      * Find the key in the hash to retrieve the running count value;
      rc=ht.find();
      * Recalculate count;
      count=sum(count,1);
      * Replace the key in the hash to update the running count value;
      rc=ht.replace();
    end;
  end;

  * When finished with the input dataset, output the hash;
  ht.output(dataset:'Want_MultiLevel_2');
run;
```

And here are the first 15 records from the Want_MultiLevel_2 output dataset. Note that these results are identical to those in Output 3 but have been produced with cleaner, more flexible code:

	level	sublevel	count
1	geo	A	422
2	geo	B	294
3	geo	C	180
4	geo	D	104
5	geo*cond1	A*0	248
6	geo*cond1	A*1	174
7	geo*cond1	B*0	158
8	geo*cond1	B*1	136
9	geo*cond1	C*0	113
10	geo*cond1	C*1	67
11	geo*cond1	D*0	63
12	geo*cond1	D*1	41
13	geo*cond1*cond2	A*0*0	225
14	geo*cond1*cond2	A*0*1	23
15	geo*cond1*cond2	A*1*0	163

Output 4. First 15 records of the Want_MultiLevel_2 output dataset from the flexible code to create multi-level summary counts

Now that we are familiar with the method, we can expand it even further. First, we can take advantage of data-step processing by calculating new variables and using them in our summary levels. Say we want counts by year of primary dose and the number of weeks between primary and booster doses. We need to add new summary levels to LIST_LEVELS using new variables year and spacing. We define and assign those variables during processing, and then the remaining processing is the same because of the flexibility we introduced in the previous step.

So far, we have worked with summary levels that have no missing values, which admittedly is not very realistic. But now, because boostdate is missing for individuals who received the primary dose but not the booster, we need a way to handle missing values in the spacing variable. As I noted earlier, the VVALUEX() function returns a formatted value, so we could simply use the default missing value ".", or we could apply the more descriptive value "NoBoost" with a custom format. And since we are already using a custom format, we can also format the non-missing spacing values to be more descriptive and to enable a better sort order:

```
proc format;
  picture spacingfmt
    low-high='99Weeks'
    other='NoBoost'
  ;
quit;
```

We can also include new summary statistics. Since we already have count, we can calculate the mean age if we first calculate the sum as a running total similar to how we calculate count. We could recalculate the mean as each input record is processed, but because each calculation within the nested do-loops for every level and sublevel for every input data record will add to the processing time, we should find a way to calculate the mean for each record in the hash only after the entire input dataset has been processed. To do this, we need to use the hash iterator.

As a refresher, the hash iterator iterates through the records in a hash object without using a key. Different methods can be used to specify the iteration order, where to begin, etc. For our purposes, the hash iterator allows us to simply process each record in the hash object, from first to last. As each record is processed, we can calculate the mean and replace the key in the hash object to add the mean to our summary table, all before we output the hash to a dataset.

Now that we are using the hash iterator, we can add a new statistic that we could not otherwise calculate and that PROC MEANS cannot produce: percent of total. To do this, we first need to add an overall level to our multi-level summary. Instead of adding this new level to LIST_LEVELS, we can hard-code it by looping from i=0 instead of i=1. Within the loop, we reset sublevel to missing, and when i=0, we assign level="_overall_". For all other iterations of i, we use the same methods to assign level and sublevel from LIST_LEVELS.

Because our hash object is ordered, and level is the first variable added to the key, the "_overall_" level will be the first record in the hash object since all other variable names begin with an alphabetic character. Knowing this, we can assign count from the "_overall_" record to a new variable total that will hold the overall count, allowing us to calculate percent of total for each record as we iterate through the hash object.

Now we can walk through the necessary modifications in the code. Begin by defining the percent variable and adding it to the hash data. After the hash object has been defined, declare the hash iterator, which I've named "hit" to represent hash iterator, and enter 'ht' as the argument since we want to iterate through the ht hash object:

```
dcl hiter hit('ht');
```

Then after the DOW loop to process the input dataset, add a new step to process the hash iterator using a DO-WHILE loop with the argument hit.next()=0. Using the .NEXT() hash iterator method opens and processes the first record in the hash object and continues on to the next record until none are left. Note that no keys are involved because the hash iterator does not use key lookups for processing. Instead, it simply iterates through each record in the hash object. Think of this as a sort of analog to the SET statement for a dataset, but because the hash object is not a dataset, we use the hash iterator instead. When the first record is processed, which we already know contains the "_overall_" level, assign a new temporary variable total with the value of count. Then we can calculate percent=total/count and mean=sum/count and replace the key in the hash to update percent and mean. When this process is complete, we output the hash object to a dataset.

Here is the code for the multi-level, multi-statistic summary:

```
* Assign custom format for the dose spacing variable;
proc format;
  picture spacingfmt
    low-high='99Weeks'
    other='NoBoost'
  ;
quit;

* Assign macro variable LIST_LEVELS with desired summary levels;
%let LIST_LEVELS=geo geo*cond1 geo*cond1*cond2 year year*spacing;

* Begin with data _null_ to output only the hash object;
data _null_;
  * Load descriptors from the input dataset...;
  if 0 then set Have;
  * ...and define key and statistic variables for use in the hash object;
  length year spacing 3 level sublevel $20 count sum 5 mean percent 8;
  * ...and apply formats;
  format spacing spacingfmt. mean 4.1 percent percent8.1;

  * Declare an ordered hash object...;
  dcl hash ht(ordered:'Y');
  *...with variable(s) you want to summarize as hash key...;
  ht.definekey( 'level', 'sublevel');
  *...plus the key and statistic variables as hash data;
  ht.definedata('level', 'sublevel', 'count', 'sum', 'mean', 'percent');
```

```

ht.definedone();
* Declare a hash iterator for the ht hash object;
dcl hiter hit('ht');

* Use a DOW loop to process the input dataset;
do until (eof);
  set have end=eof;
  * Assign new variables for use in the summary;
  year=year(dosedate);
  call missing(spacing);
  if not missing(boostdate) then spacing=floor((boostdate-dosedate)/7);
  * To summarize an overall level and requested levels...;
  do i=0 to countw("&LIST_LEVELS", ' ');
    * Reset sublevel to missing;
    call missing(sublevel);
    * Assign the overall level...;
    if i=0 then level='_overall_';
    * ...then the levels requested in LIST_LEVELS;
    else do;
      * ...assign level from the LIST_LEVELS macro variable...;
      level=scan("&LIST_LEVELS", i, ' ');
      * ...reset sublevel to missing...;
      call missing(sublevel);
      * ...assign sublevel with values of all variables in the level;
      do j=1 to countw(level, '*');
        sublevel=catx('*', sublevel, vvaluex(scan(level, j, '*')));
      end;
    end;
    * Initialize statistics to missing;
    call missing(of count sum);
    * Find the key in the hash to retrieve the running statistics;
    rc=ht.find();
    * Recalculate statistics;
    count=sum(count, 1);
    sum=sum(sum, age);
    * Replace the key in the hash to update the running statistics;
    rc=ht.replace();
  end;
end;

* Use hit hash iterator to process all records in the ht hash;
do while(hit.next()=0);
  * Assign total with overall count;
  if level='_overall_' then total=count;
  * Calculate new statistics;
  percent=count/total;
  mean=sum/count;
  * Replace the key in the hash to add the new statistics;
  ht.replace();
end;

* When finished with the input dataset, output the hash;
ht.output(dataset: 'Want_MultiLevel_3');
run;

```

And here are the first 15 records of the Want_MultiLevel_3 output dataset:

	level	sublevel	count	sum	mean	percent
1	_overall_		1000	42341	42.3	100.0%
2	geo	A	422	17601	41.7	42.2%
3	geo	B	294	12537	42.6	29.4%
4	geo	C	180	7868	43.7	18.0%
5	geo	D	104	4335	41.7	10.4%
6	geo*cond1	A*0	248	10688	43.1	24.8%
7	geo*cond1	A*1	174	6913	39.7	17.4%
8	geo*cond1	B*0	158	6760	42.8	15.8%
9	geo*cond1	B*1	136	5777	42.5	13.6%
10	geo*cond1	C*0	113	5084	45.0	11.3%
11	geo*cond1	C*1	67	2784	41.6	6.7%
12	geo*cond1	D*0	63	2623	41.6	6.3%
13	geo*cond1	D*1	41	1712	41.8	4.1%
14	geo*cond1*cond2	A*0*0	225	9750	43.3	22.5%
15	geo*cond1*cond2	A*0*1	23	938	40.8	2.3%

Output 5. First 15 records of the Want_MultiLevel_3 output dataset from the code to create a multi-level, multi-statistic summary

And here is another excerpt from the Want_MultiLevel_3 output dataset to show the year and year*spacing levels, noting the formatted values for spacing:

	level	sublevel	count	sum	mean	percent
30	year	2021	287	11908	41.5	28.7%
31	year	2022	296	12600	42.6	29.6%
32	year	2023	277	12063	43.5	27.7%
33	year	2024	140	5770	41.2	14.0%
34	year*spacing	2021*05Weeks	5	217	43.4	0.5%
35	year*spacing	2021*06Weeks	34	1350	39.7	3.4%
36	year*spacing	2021*07Weeks	76	3197	42.1	7.6%
37	year*spacing	2021*08Weeks	62	2529	40.8	6.2%
38	year*spacing	2021*09Weeks	21	915	43.6	2.1%
39	year*spacing	2021*10Weeks	4	148	37.0	0.4%
40	year*spacing	2021*11Weeks	1	20	20.0	0.1%
41	year*spacing	2021*NoBoost	84	3532	42.0	8.4%
42	year*spacing	2022*05Weeks	4	203	50.8	0.4%
43	year*spacing	2022*06Weeks	22	933	42.4	2.2%
44	year*spacing	2022*07Weeks	66	2835	43.0	6.6%

Output 6. First 15 records for the year and year*spacing levels in the Want_MultiLevel_3 output dataset from the code to create a multi-level, multi-statistic summary

I hope this demonstration has already begun to inspire more ideas for modifying and expanding the method. Maybe you want to calculate summary statistics for more than one quantitative variable, or maybe you only want to specify which statistics to calculate for different variables. In public health research, for example, we often need counts of health outcomes and sums of eligible person time to calculate outcomes per unit of person time. Or perhaps you want a summary table that includes all valid values, even if some values do not appear in the data, for which you could create a shell table first to use as the hash object to populate when processing the data. Hopefully, the more comfortable you get with new programming methods, the more new ways you can find to use them.

DISCUSSION

The hash method for creating multi-level summary tables has many advantages. We can use a single DATA step, so the code is compact, easy to read, and easy to modify. Because we use hash programming within a DATA step, we do not need to pre-sort or pre-process the data. The resulting summary table is also compact and readable because it is narrow instead of broad, vertical instead of horizontal. With DO loop processing, it is easy to process new summary levels, and each level only adds new rows but not new variables to the summary, keeping the results compact and readable. With the hash iterator, we can make more efficient calculations and expand the available descriptive statistics.

However, this method also has disadvantages. It cannot easily calculate complex statistics like standard deviation. Though new values and variables can be calculated within the same DATA step, each new calculation and variable adds to the processing space and time. Though this solution is scalable relative to the number of records in the input dataset, it is still limited by available memory at execution, which in turn can limit the number of summary levels, the length of the level values, and the number of unique values per sublevel. Depending on these factors, the method may be less efficient than others.

My purpose is not to compare methods to declare a clear winner or a "best" solution, but to take one example, perhaps the most obvious alternate to the hash method is PROC MEANS. It is highly optimized to summarize large amounts of data and can most likely produce a summary faster than the hash method. It can easily calculate more complicated statistics that the hash method may or may not be able to handle, even with additional programming. But PROC MEANS may not be able to do everything by itself. It may require pre-processing with a DATA step to produce values to be summarized, and the data may need to be sorted to take advantage of certain features. Although you can select specific combinations of variables to summarize with the TYPES statement, the output dataset still contains all variables being summarized across all levels, so you may need to further process the output to make your summary more readable. Consider the advantages and disadvantages of both when choosing your solution.

CONCLUSION

When I first learned to program in SAS, I practiced new skills by using SAS to recreate processes that I had already developed with other tools. This is still an effective way for me to learn and expand my programming skills, except now I also use new-to-me SAS programming techniques to replicate results created with more familiar SAS methods. In this paper I have explored hash programming methods to present a compact, flexible, and scalable solution to a common problem and hope I have added something useful to your SAS programming toolbox. Practice may not necessarily make perfect, but practice can make progress when the results are easy to verify.

REFERENCES

1. Dorfman, Paul M., Henderson, Don. 2015. "Data Aggregation Using the SAS Hash Object." Proceedings of the SAS Global Forum 2015. Dallas, TX. Available at <https://support.sas.com/resources/papers/proceedings15/2000-2015.pdf>.
2. Dorfman, Paul M., Vyverman, Koen. 2009. "The DOW-Loop Unrolled." SAS Global Forum 2009, Beyond the Basics, paper 038-2009, Washington, DC. Available at <https://support.sas.com/resources/papers/proceedings09/038-2009.pdf>.

ACKNOWLEDGMENTS

I would like to thank Lauren Zichittella for encouragement and support in writing this paper; Sarah Woodruff for sharing experience and knowledge during abstract and paper development; Neesha Nathwani for providing helpful feedback on the paper; and the Harvard Pilgrim Health Care Institute for supporting my work and attendance at this conference.

CONTACT INFORMATION

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

David V. Cole
Department of Population Medicine, Harvard Pilgrim Health Care Institute
Boston, Massachusetts
david_cole@hphci.harvard.edu

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX

To produce a test dataset for the example scenario to analyze individuals receiving one or both doses of a fictional vaccine, I started with a macro variable IDCOUNT to specify the number of records to generate, making it easy to create multiple datasets to test performance of the method based on the size of the input data. I also assigned macro variable RANDSEED to specify a seed for random number generation in a CALL STREAMINIT routine so that the data can be reproduced.

Then I used different variations of the RAND() function to randomly generate data. I used RAND('integer') to assign dosedate and age uniformly across their respective ranges. I used RAND('bern') to determine the proportion of individuals receiving a booster dose, and then I used RAND('normal') to randomize the number of days between primary and booster doses across a normal distribution. I used RAND('table') to assign individuals to geographic regions of different sizes, and for the pre-existing conditions, I again used RAND('bern') to vary the prevalence of each condition among the vaccinated individuals.

Here is the code to create randomized test data for the example scenario:

```
%let IDCOUNT = %eval(10**3);
%let RANDSEED = 81;

data Have;
  length uniqueid 5 dosedate boostdate 4 geo $1 age  cond1-cond3 3;
  format dosedate boostdate date9.;
  call streaminit(&RANDSEED);
  do uniqueid=1 to &IDCOUNT;
    call missing(of dosedate boostdate geo age preexist: );
    /* Randomly assign data values */
    dosedate=rand('integer', "01JAN2021"d, "30JUN2024"d);
    if rand('bern', 0.7) then do;
      boostdate=floor(rand('normal', 56, 7))+dosedate;
    end;
    geo=scan("A B C D", rand('table', 0.4, 0.3, 0.2, 0.1));
    age=rand('integer', 20, 64);
    cond1=rand('bern', 0.4);
    cond2=rand('bern', 0.1);
    cond3=rand('bern', 0.0001);
    output;
  end;
run;
```