Paper 111-2011

Using Hash Tables to Obtain Matched Post-Hoc Control Populations

Jeffrey Reiss, University of Central Florida, Orlando, FL
Elayne Reiss, University of Central Florida, Orlando, FL

ABSTRACT

In social science and other settings, analysts are often asked to conduct post-hoc studies to measure the effectiveness of a particular program or treatment. In other words, those implementing the program have not identified an appropriate control group for comparison purposes, although it may be crucially necessary in order to conduct an effective analysis of the study. For example, in some educational studies, an intervention may target a very specific type of student with a particular academic background and demographic profile, so the creation of a closely matched control group—created after the fact—is essential to quality data comparison.

This paper explores a straightforward yet handy use of the hash table, a data structure available in Base SAS®, to turn a large database of mostly "mismatched" potential control group observations into a much more comparable sampling population. This code can accommodate as many baseline comparative factors as necessary, both continuous and discrete.

INTRODUCTION

Within Base SAS, there are numerous methods to traverse and match data sets. One of the more recently added methods, hash tables, was introduced in SAS Version 9 as a way to expedite data set navigation—at the cost of memory usage. While the hash table's speed is unmatched, it does have a downfall of only being able to match on keys consisting of discrete data points. If a data range or continuous variable constitutes a key for matching, hash tables lose their usefulness due to their underlying structure.

By using some simple code manipulations, however, the hash table can account for these situations without sacrificing any accuracy. After providing a general overview of hash tables, this paper will elaborate on two particular examples from social science research where hash tables were utilized to provide a tightly matched control group population. The resulting populations can serve as inputs to a larger study utilizing other sampling and analytical methods.

AN OVERVIEW OF HASH TABLES

Hash tables were introduced in a user-friendly fashion in SAS 9 as a more versatile method to manipulate data. They take an entire source data set and use it as a key in order to find similar values in a destination data set in a much quicker fashion than other SAS data manipulation methods. For example, suppose that a retail company would like to obtain specific sales figures from a handful of items amidst a large database of products. The DATA step that would produce these figures is as follows:

```
data sampleset;
    input sku $;
datalines;
MZT4567
ABC3956
QQQ4587
;

data sample;
    if _n_ = 1 then do;
           declare hash h(dataset: "sampleset");
           h.defineKey('sku');
           h.defineDone();
    end;
    set productlisting;
           if h.find() = 0 then output;
run;
```

Within this simple example, the hash table is declared on the first iteration of the DATA step defining data set "sample." The only variable in the data set, *sku*, is defined as the lookup key, utilizing the values found in data set "sampleset." After the definition of the hash table, the DATA step searches through all of the observations in data set

“productlisting” to find any items that match any of the contents of variable *sku*, the hash lookup key. In order for this process to work, however, both data sets must contain the key variables with identical names. For instance, if *sku* is actually named *productsku* in data set “productlisting,” it must be first renamed to *sku* in the SET statement in order for the observation matching to work. The above code creates a sample of values contained in the data set “productlisting” that meet the criteria found in another data set without using extra DATA steps or PROC SORT code.

The previous example described a hash table in its simplest form. At this stage, it does not demonstrate much of a reduction in code beyond the elimination of some PROC SORT code. To build upon this example, suppose that the retail company wants to take the items found in data set “sampleset” and assign them a sale price.

```
data sampleset;
    input sku $ sale 5.2;
datalines;
MZT4567 100.99
ABC3956 37.50
QQQ4587 356.33;

data sample;
    format sale 5.2;
    if _n_ = 1 then do;
           declare hash h(dataset: “sampleset”);
           h.defineKey(‘sku’);
           h.defineData(‘sale’);
           h.defineDone();
    end;
    set productlisting;
           if h.find() = 0 then output;
run;
```

In this example, only two lines of code were added to the DATA step. The first line is a simple format statement for the sale variable that was added to the “sample” data set. This is a requirement for the hash table to work properly when variables from the hash table are added to the destination data set; otherwise, SAS will generate an error when running the code due to possible format mismatching. The second line is the official definition of the *sale* variable as a data variable within the hash table. The hash data variable function DEFINEDATA acts differently from the hash lookup key function DEFINEKEY in that the FIND function only passes DEFINEDATA values along instead of matching upon them, as it would with DEFINEKEY values. Variables declared under DEFINEDATA are also available for use within any additional DATA step processing of the data set. Once the DATA step finishes, the same items from the hashing data set “sampleset” will be in the resulting sample data set “sample” and will also contain the key’s corresponding value from the *sale* variable.

WHY USE HASH TABLES?

Considering that explicit hash table functions are a fairly recent addition to SAS, it is important to answer the question of why hash tables may be a better method to utilize than a combination of any other pre-existing functions. While no programming method is ever perfect, hash tables have a number of pros that outweigh the cons.

SORTING POWER

The first major advantage that has been shown through the aforementioned retail examples is the lack of a need to pre-sort the data. In the MERGE/BY setup, the code compares two lists of variables, but they have to be sorted first in order for the DATA step to run. This may be troublesome if one of the data sets is sorted in a unique way that does not reference the key. A programmer would first have to sort the data set by the key, run the merge, conduct any necessary data set cleanup, and then re-sort it into the original order. Because the hash table runs strictly off memory, its functions simply create a hash address location for each key value when searching. Likewise, when the FIND function is called upon, it searches through the list of hash keys in order to match the current observation in a given data set with a corresponding hash key value. In the case of the retail examples, if the search yields a matching value, that observation is recorded into the destination data set along with any other non-key variables defined in the hash declaration. At the same time, the natural order of the data set is preserved, saving time on multiple sorts.

ADDING AND REMOVING OBSERVATIONS

Hash tables also serve as a dynamic method for data storage. They can become as large or as small as the programmer wishes. This is where two functions come into service: ADD and REMOVE. As their names imply, the functions can add and remove entries from the hash table. From a practical sense, this is a simple way to maintain an ever-changing data set. Returning to the retail store example, consider the situation where every week, products are added and removed from the master product list to reflect actual weekly store sales figures. Through the ADD and REMOVE functions, this process becomes a simple task, as demonstrated in the following code segment.

```
data productlisting;
    input sku $ price 7.2 description $ 16-37;
datalines;
MZT4567 200.99 Panaphonics TV
DSE5382  45.66 Blendco Toaster
Q745Q    35.39 Toastronic Blender
;

data changes;
    input adrem:$1 sku $ price 7.2 description $ 19-40 ;
datalines;
r MZT4567 200.99  Panaphonics TV
a MZT4567 150.99  Panaphonics TV
r Q745Q   35.39   Toastronic Blender
a B4UHVAM 4500.99 Sorny 45" Plasma TV
;

data _null_;
    format sku $7. price 7.2 description $20.;
    if _n_ = 1 then do;
           declare hash h(dataset: 'productlisting');
           h.defineKey('sku');
           h.definedata( 'sku', 'price', 'description');
           h.defineDone();
    end;
    set changes end=done;
           if adrem='r' then do;
                  if h.find()=0 then h.remove();
           end;
           else if adrem='a' then do;
                  if h.find()^=0 then h.add();
           end;
           if done then h.output(dataset:'productlisting');
run;
```

The “productlisting” data set, the hash table, serves as the main database of products. Likewise, the “changes” data set contains the products that are to be added and removed as indicated by the *adrem* variable. Technically, only the *sku* and *adrem* variables are needed for product removals, but from a quality control standpoint, all of the item information should be contained within the “changes” data set. As for the DATA step, it processes through a null data set because all change information is contained within data set “productlisting.” The hash declaration within the null data set is standard, addressing the fact that all three variables in the hash data set should be used to update the “changes” data set, marked by the SET statement. The *done* variable marks the end of the data set. If an observation is marked with an ‘r,’ indicating removal, then SAS will check to see if that observation is in the hash table and will subsequently remove it from the hash table if the observation is located. If an observation is marked with an ‘a,’ indicating an addition, then SAS will check that the observation is not already in the hash table and add it. The FIND function serves as an error trap to ensure proper execution. Finally, once the end of the “changes” data set is reached, variable *done* will be set to ‘true’ and the hash OUTPUT function is executed to save the changes. This OUTPUT function is crucial; otherwise, all the processing in the null DATA step would be lost. Afterwards, any sorting methods will need to be re-applied to the “productlisting” data set because all newly added values are automatically placed at the beginning of the data set.

Despite all the benefits of hash tables, they contain one major flaw—memory usage. Because hash tables place an entire data set in memory, not having enough memory may be problematic. While a new computer with over two gigabytes of memory may not have a problem, an older machine may face some memory issues when using large hash tables. When this situation occurs, the programmer will have to settle for a slower, less memory-intensive way to handle the data. Now that the basics have been explained, the following sections will showcase how hash tables can quickly solve the more complex problems faced within the realm of creating post-hoc matching control data sets for a specific study.

MATCHING RANGES OF VALUES AND CONTINUOUS VARIABLES

To demonstrate how ranges of values and continuous variables can be properly matched, consider the following situation. We were faced with the task of comparing quantifiable academic and behavioral progress of students who enrolled in an alternative high school program to that of students who continued their enrollment in their zoned high schools. Enrollment in this alternative high school was largely self-selective, so while the population of these students shared the general trait that they were having some academic difficulties in their zoned schools, not all students necessarily held a grade point average (GPA) below a certain level or held failing scores on state standardized tests.

With this type of research, it would be physically impossible to track a particular student's progress through a year or two of the alternative high school, then bring the student back to a zoned high school and compare progress for the next year or two. By nature of how students progress through schooling, maturation effects would quickly invalidate results, nor do we have the power to reverse time and eliminate a student's academic and behavioral maturation effects completely. Therefore, in order to answer the question, "Would the alternative school students have made more or less academic and behavioral progress had they been enrolled in their zoned schools," we needed to attempt to provide a "matching" student to each alternative school student.

Based upon prior research conducted within this particular school district, we listed a set of variables that have been historically proven as significant factors that can differentiate academic performance between any two given students. The data set preparation goal involved finding a matching student for every unique student in the alternative high school population based upon this set of factors. We needed to find a way to achieve this goal efficiently, particularly when the control population for sampling contained 30,000 students and 9 different criteria (both continuous and discrete in nature) were necessary for matching.

DATA SET 1: HANDLING VARIABLES WITH ORDINAL RANGES

When working with both continuous and discrete variable matching, an appropriate control sample is obtained through running some preparations on the control population data set to maximize the number of resulting observations.

PREPARATION

For this situation, one of the stipulations for the control data set to indicate a match with a student in the alternative school was that their math and language arts GPAs needed to be within 0.5 of a point of the actual value, on a 4-point scale. In the code below, the *markla1* variable indicates language arts GPA, while the *markmath1* variable indicates mathematics GPA. However, these variables are ordinal, since a yearly GPA in a single subject can only fall within 0.5-point increments on a range of 0 to 4. Therefore, determining every possible combination of subject GPAs does not result in hundreds of observations with ordinal scores such as these.

```
data cclc&year;
    set source.cclc_&year;
    output;
    if markla1 ^= . and markmath1 ^= . then do;
           markmath1 ++.5;
           output;
           markmath1 ++ -1;
           output;

           markmath1 ++.5;
           markla1 ++.5;
           output;
           markmath1 ++.5;
           output;
           markmath1 ++ -1;
           output;
```

```
            markmath1 ++.5;
            markla1 ++ -1;
            output;
            markmath1 ++.5;
            output;
            markmath1 ++ -1;
            output;
    end;
run;
proc sort data = cclc&year nodupkey;
    by schoolnum grade ethniccode freemeals pr_mastr pr_mastm markla1 markmath1;
run;
```

The preceding DATA step, while somewhat basic in nature, outputs every possible combination of math and language arts GPAs based upon the scores of the students in the alternative school data set. The additions and subtractions of 1 and 0.5 accommodate the ± 0.5 span from the actual value, which results in a total range of 1.

The &YEAR references are present since this code is contained within a SAS macro. This analysis was requested for a range of school years, so the use of macro code to perform the same operations on each separate year's data set serves as a good way to minimize the amount of necessary redundant code. Once all of the combinations of math and language arts GPAs are created, the PROC SORT code is run with the NODUPKEY option to eliminate any duplicate combinations. Once again, data sets used for creating hash tables should be as lean as possible to minimize memory usage.

VIEWTABLE: Work.Cclc0405

	FIRSTNAME	LASTNAME	markla1	markmath1
33	NIKITA		2.50	1.50
34	NIKITA		2.50	2.00
35	NIKITA		2.50	2.50
36	NIKITA		3.00	1.50
37	NIKITA	Starting Point -->	3.00	2.00
38	NIKITA		3.00	2.50
39	NIKITA		3.50	1.50
40	NIKITA		3.50	2.00
41	NIKITA		3.50	2.50

Figure 1. Data Set Displaying Possible Discrete Combinations of Variables

Figure 1 provides an example of the end result for the *markla1* and *markmath1* variables. The original observation for this particular student, as indicated by the arrow, was a language arts GPA of 3 and a math GPA of 2. The routine written through the above code creates every combination of the ± 0.5 range. Since the language arts GPA can range from 2.5 to 3.5 and the math GPA can range from 1.5 to 2.5, each combination is depicted in the data set as a possible value set for the hash table, as will be described in the next subsection.

THE HASH TABLE

The hash code for this situation follows closely to the code found within the retail example described earlier.

```
data cclc&year.hash;
    if _n_ = 1 then do;
            declare hash h(dataset: "cclc&year");
            h.definekey('schoolnum', 'grade', 'ethniccode', 'freemeals',
                    'pr_mastr', 'pr_mastm', 'markla1', 'markmath1');
            h.definedone();
    end;
```

```
    set source.control_&year;
    if h.find()=0 then output;
run;

proc sort data = cclc&year.hash ;
    by schoolnum grade ethniccode freemeals pr_mastr pr_mastm markla1 markmath1;
run;
```

While this code may appear simple, this solution shows off another benefit of the hash table—the key size. The key in this case consists of eight different variables. Without the use of hash tables, obtaining the same results would have likely required far more convoluted coding solutions through DATA step loops. It should be noted, however, that this method should be restrained to smaller data sets as the process triples its size.

DATA SET 2: HANDLING VARIABLES WITH CONTINUOUS RANGES

Since continuous variables technically have an infinite range, they must be treated differently within the hash table. Unlike their discrete counterparts, it is largely impossible to create a table containing all combinations of a continuous variable. Therefore, extra preparation is required.

PREPARATION

For this situation, one of the stipulations for entry to the control data set is that a student's cumulative GPA in the zoned school population must be within ± .25 to be considered a "match" to any given student in the alternative high school. Unlike the previous example, a cumulative GPA can fall anywhere in this continuous range, so the data must be set up differently.

```
data quest; /*prep for continuous*/
    set source.quest (rename=(hszone=school));
    gpalow = pr_gpa-.25;
    gpahi = pr_gpa + .25;
run;
```

Compared to the preparation code for Data Set 1, this setup is quite simplistic. Two new variables are created for each GPA to represent their lowest point and highest point. The duplicate observation-removing PROC SORT was not run because in this scenario, there is considerably more variation and less of a chance a duplicate would exist. If desired, this solution may also be applied to the previous example that handled discrete values. Choosing this method may be considered more efficient, as it clearly requires far less code.

THE HASH TABLE

The hash table code for this situation follows closely to the final retail example from the hash table overview section.

```
data source.questhash2;
    format gpalow gpahi 6.3;
    if _n_ = 1 then do;
           declare hash h(dataset: "quest");
           h.definekey('school', 'pr_grade', 'grade', 'ethnic', 'frl',
                  'py_rmast', 'py_mmast');
           h.definedata('gpalow', 'gpahi');
           h.definedone();
    end;
    set source.control ;
    if h.find()=0 then do;
           if gpalow <= pr_gpa <= gpahi then do;
                  output;
           end;
    end;
run;
```

Within this example, the hash table's DEFINEDATA function is utilized to reference the high and low values defined during the data preparation stage within the variables *gpalow* and *gpahi*. While not explicitly part of the key, they collectively serve as a pseudo-key of sorts for the GPA variable. After SAS finds an observation that matches all of the discrete key values given by the DEFINEKEY function, the continuous GPA variable is then checked to determine if it is between the low and high values, which still exist within the hash table through the DEFINEDATA function. If the GPA variable is within the specified range, that observation is output to the control data set. If not, the DATA step continues until its next find. Once this process is completed, a fully functioning control data set is available to complete the analysis. If multiple continuous variables are present and required for the key value, a simple collection of nested IF-THEN statements will provide the desired results. Keep in mind, however, that the hash table needs at least one discrete variable to function properly. If a data set has all or mostly continuous keys, another method should be pursued.

CONCLUSION

This paper demonstrated how hash table programming through SAS can serve as an extremely efficient method for matching data sets and does not require a cumbersome set of sorts, merges, and loops, which can become even more cumbersome upon the addition of extra variables. The functions that contribute to hash table dynamics are straightforward and versatile, and although they only explicitly allow for the use of discrete variables as keys, a little bit of creative coding can help to bypass this issue.

REFERENCES

Dorfman, P., Snell, G. "Hashing Rehashed." 2002. *Proceedings of the SAS Users Group International 27th Conference.* Orlando, FL. Available at http://www2.sas.com/proceedings/sugi27/Proceed27.pdf.

Muriel, E. "Hashing Performance Time with Hash Tables." 2007. *Proceedings of the SAS Global Forum 2007 Conference.* Orlando, FL. Available at http://www2.sas.com/proceedings/forum2007/039-2007.pdf.

Secosky, J., Bloom, J. "Getting Started with the DATA Step Hash Object." 2007. *Proceedings of the SAS Global Forum 2007 Conference.* Orlando, FL. Available at http://www2.sas.com/proceedings/forum2007/271-2007.pdf.

CONTACT INFORMATION

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

Jeffrey Reiss
University of Central Florida
Operational Excellence and Assessment Support
12424 Research Pkwy, Suite 225
Orlando FL 32826
Phone: (407) 882-0261
jreiss@ucf.edu

Elayne Reiss
University of Central Florida
University Analysis and Planning Support
12424 Research Pkwy, Suite 215
Orlando FL 32826
Phone: (407) 882-0267
ereiss@ucf.edu