

Paper SBC-120

Utilizing Hash Tables to Obtain Matched Post-Hoc Control Populations

Elayne Reiss, University of Central Florida, Orlando, FL

Jeffrey Reiss, Independent Statistical Consultant, Lake Mary, FL

ABSTRACT

In education, we are frequently asked to conduct post-hoc studies to measure the effectiveness of a particular student instructional program. In other words, those implementing the program have not identified an appropriate control group for comparison purposes. Some studies target a very specific type of student with a particular background academic and demographic profile, so the creation of a closely matched control group—after the fact—is essential to quality data comparison.

This paper will explore a simple yet handy use of hash tables 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. We will also include the data preparation requirements and sampling procedures (through PROC SURVEYSELECT) to provide a holistic view of the analytical process.

INTRODUCTION

Statisticians working in any social science area, such as education, can be faced at any time with a post-hoc study request intending to analyze the effectiveness of some program or intervention. In some cases, simply measuring the growth or decline of a factor among program participants can appropriately address the research questions. In other situations, the end results of program participants must be compared to those of an appropriate control group to help measure program success.

When the participant-control design structure is selected as most appropriate from an analytical perspective, what can a statistician do upon the realization that the individuals who launched the program never identified a suitable control group? In the case of some programs, participants must meet a stringent set of requirements for entry, whether they relate to demographic, academic, or other performance-type criteria. For these situations, selecting a control comparison group is not a difficult task. Since all participants in the program had a specific set of entrance requirements, control groups would have to meet this same set of entrance requirements as well.

Not all program evaluation studies are created equal, however. Some programs are more self-selective than others, which yield participants who may share a general common trait (i.e. “living in a low-income area” or “do not mesh well with a traditional high school environment”) but may or may not share such a concentrated distribution of sub-traits, such as standardized test performance or school attendance rate. Fortunately, when the target variables involve metrics collected for all individuals in a population, regardless of participant status, a little bit of SAS® programming can resolve the issue quickly, easily, and accurately. Hash tables, a programming feature made widely available in SAS Version 9, gives an analyst the power to overcome the post-hoc issue of finding an appropriate control group through one-to-one matching on a set of key traits.

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 link between the resulting tables and the sampling method that leads to analysis will be provided thereafter.

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 dataset and use it as a key in order to find similar values in a destination dataset much more quickly 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 dataset "sample." The only variable in the dataset, *sku*, is defined as the lookup key, utilizing the values found in dataset "sampleset." After the definition of the hash table, the DATA step searches through all of the observations in dataset "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 datasets must contain the key variables with identical names. For instance, if *sku* is actually named *productsku* in dataset "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 dataset "productlisting" that meet the criteria found in another dataset 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 dataset "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" dataset. This is a requirement for the hash table to work properly when variables from the hash table are added to the destination dataset; otherwise, SAS will generate an error when running the code. 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 (**find**) essentially ignores the data variable values in its search process. Variables declared under **defineData** are also available for use within any additional DATA step processing of the dataset. Once the DATA step finishes, the same items from the hashing dataset, "sampleset," will be in the resulting sample dataset, "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 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, SAS 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 datasets is sorted in a unique way that does not reference the key. A programmer would first have to sort it by the key, run the merge, conduct any necessary dataset 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 dataset with a matching hash key value. In the case of the retail examples, if the search yields a matching value, that observation is recorded into the destination dataset along with any other non-key variables defined in the hash declaration. At the same time, the natural order of the dataset is preserved, saving time on multiple sorts.

ADDING AND REMOVING OBSERVATIONS

Hash tables also serve as a dynamic method to 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 have the ability to add and remove entries from the hash table. From a practical sense, this is a simple way to maintain an ever-changing dataset. Returning to retail store example, consider that every week, products are added and removed from the master product list to reflect actual weekly store sales figures. Through add and remove procedures, this process becomes a simple task, as demonstrated below.

```
data productlisting;
    input sku $ price 7.2 description $ 16-37;
datalines;
MZT4567 200.99 Panaphonics TV
DSE5382 45.66 Blendco Toaster
Q745Q 35.39 Q-Tech 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 Q-Tech 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” dataset, the hash table, serves as the main database of products. Likewise, the “changes” dataset 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” dataset. As for the DATA step, it processes through a null dataset because all change information is contained within dataset “productlisting.” The hash declaration within the null dataset is standard, addressing the fact that all three variables in the hash dataset should be used to update the “changes” dataset, marked by the **set** statement. The *done* variable marks the end of the dataset. 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 remove it from the hash table, if 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” dataset is reached, variable *done* will be set to true and the hash output statement is executed to save the changes. This output statement is crucial; otherwise, all the processing in the null DATA step would be lost. Afterwards, any sorting methods will need to be reapplied to the “productlisting” dataset because all newly added values are automatically placed at the beginning of the dataset.

Despite all the benefits of hash tables, they contain one major flaw—memory usage. Because hash tables place an entire dataset in memory, not having enough 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 SAS 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 datasets for a specific study.

CREATING MATCHING POST-HOC CONTROL POPULATIONS: THE ISSUE

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 the 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 historically proven to be significant factors that can differentiate academic performance between any two given students. The dataset 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.

The initial proposed method involved a looping procedure through a DATA step that would, for each of the 9 different criteria, compare each student in the alternative school population with each student in the zoned school population. The process would mark each potential member of the control group with an indicator showing that the student was a potential match on a given variable or set of variables. With so many variables, both continuous and discrete, as well as the large number of students from which to select a subset, this method would have been presumably cumbersome and processor-intensive. The hash table method served as a far more efficient method to carry out these goals.

DATASET 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 dataset to maximize the number of resulting observations.

PREPARATION

For this situation, one of the stipulations for the control dataset to indicate a match with a student in the alternative school was that their math and language arts GPA's 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 GPA's 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 DATA step above, while somewhat basic in nature, outputs every possible combination of math and language arts GPA's, 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 dataset serves as a good way to minimize the amount of necessary redundant code. Once all of the combinations of math and language arts GPA's are created, the PROC SORT code is run with the NODUPKEY option to eliminate any duplicate combinations. Once again, datasets used for creating hash tables should be as lean as possible to minimize memory usage.

	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		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

The image above 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 dataset 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.

DATASET 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 nominal 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 dataset is that a student's cumulative GPA in the zoned school population must be within $\pm .25$ 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 dataset 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;

```

The main difference between this situation and any of the previous examples is that a continuous variable is being utilized as a key value. Hash table keys can only be discrete values, so requesting a match for a continuous range poses a minor problem. In order to work through this potential issue, a workaround is necessary.

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 **definekey**, 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 dataset. If not, the DATA step continues until its next find. Once this process is completed, a fully functioning control dataset 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 dataset has all or mostly continuous keys, another method should be pursued.

SAMPLING

Thanks to hash table programming, we were able to create a formatted dataset representing a more appropriate control population by which we could compare the academic performance of students in the alternative school with that of very similar students in traditional schools. However, the process may pull multiple matching students for the control group for one alternative student, while pulling only one matching student for another. Large samples are helpful when available, but for this particular application, we would rather pare down the control population to ensure that this comparison group is not weighed unfairly, as it is still somewhat of a "convenience sample."

Our main concern for fair sampling in this case regards student grade. Students undergo very different experiences in 9th grade than they do in 10th grade, so it is important to ensure that the representation of 9th grade to 10th grade students in the control sample appropriately mirrors the representation in the alternative school group. The SAS PROC SURVEYSELECT function allows this process to occur quickly and easily.

```

proc sort data = source.questhash2;
    by grade;
run;

proc surveyselect data = source.questhash2
    method = srs
    n=(65 23)
    seed=1016
    out=ctrlsamp0607 noprint;
    strata grade;
run;

```

The alternative school population consisted of 65 9th grade students and 23 10th grade students. Armed with this information, the PROC SURVEYSELECT statement allowed us to take a simple random sample from the hash table-prepared control population to yield exactly 65 9th grade students and 23 10th grade students. Although using stratification such as the type conducted by this procedure removes some of the specificity of the matched observations uniquely selected by the hash table, a comparison of the demographics of the sampled control set and the alternative school population showed near-identical matches.

As a word of caution when using the **strata** statement, SAS requires the dataset to first be sorted in the order of the stratification variable. Additionally, the dataset from which the procedure samples must contain at least as many observations per stratum as specified in the **n=** statement. Otherwise, SAS will produce a message stating that there were insufficient observations for the stratum. Finally, it is essential to ensure that the dataset being sampled contains the exact strata as desired. For example, this instance requires a sampling of 9th and 10th grade students. If a stray 8th grade student slipped into the dataset for whatever reason (impossible in this particular hash table selection-based dataset, but possible in other applications), the PROC SURVEYSELECT code would expect to select 65 8th grade students and 23 9th grade students, since it will automatically select the lowest value in the dataset for the first stratum. Frequent dataset checks are imperative for quality output.

CONCLUSION

The above example demonstrates how hash table programming through SAS can serve as an extremely efficient method for creating an appropriate post-hoc control dataset, particularly when data have been collected for an entire population. Matching program participants on any group of key demographics or performance variables to a non-participant population 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 in cases such as ours. Coupled with careful dataset preparation, hash table programming helped us turn a potential post-hoc study nightmare into a successful, stress-free statistical analysis.

REFERENCES

- P. Dorfman, G. Snell. *Hashing Rehashed*. SUGI 27, Orlando, FL, 2002.
- E. Muriel. *Hashing Performance Time with Hash Tables*. SAS Global Forum 2007, Orlando, FL, 2007.
- J. Secosky, J. Bloom. *Getting Started with the DATA Step Hash Object*. SAS Global Forum 2007, Orlando, FL, 2007.

CONTACT INFORMATION

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

Elayne Reiss
University of Central Florida
University Analysis and Planning Support
12424 Research Parkway
Orlando, FL 32826
(407) 882-0267
uaps@mail.ucf.edu
<http://uaps.ucf.edu>

Jeffrey Reiss
Independent Statistical Consultant
jeff10@bellsouth.net

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.