Paper 236-2012

Implementing Control Selection Using Hash Tables: A Case Study

Craig Kasper, Manitoba Health, Winnipeg, Manitoba, Canada

ABSTRACT

This paper discusses a program written to perform matched control selection from a research population, using hash tables to eliminate the need for a large temporary data set, and allowing the process to run significantly faster as a result. It describes the development of the program from the point of initial need to the completion of the working program, and discusses how insights into the control selection process enabled the transformation of the program from using conventional methods to one using dynamic programming.

INTRODUCTION

It is a fact of the programmer's life that successfully acquiring skill in one area invariably results in opportunities or even responsibilities in another. As a result, skilled programmers often find themselves continually facing new problems and challenges.

It is also a fact of the programmer's life that the programmer's toolbox is continually improving and expanding. These new or improved tools enable new solutions to both new and old problems and challenges.

This paper is a case study which explores a new set of problems and challenges I was faced with recently – the selection of controls for case patients in epidemiological research – and will be of interest to the academic SAS® user looking for information on implementing control selection or related population sampling techniques.

This paper also explores how a relatively new tool – the hash object, which was introduced in SAS 9.1 – proved to be a pivotal part of an innovative solution to these problems. As such, this paper will be of interest to programmers who have an understanding of the hash object and its use, but may be uncertain of how best to apply it. As it discusses approaches to programming, this paper may also be of interest to programmers new to the hash object, but it is not intended to be a primer on the subject. If the reader is looking for an introduction to the hash object, the reader may be interested in the papers by Dorfman et al. and by Eberhardt listed in the recommended reading.

CONTROL SELECTION: WHAT IS IT?

A matched control study is a type of research study where researchers compare patients under study to other individuals selected from the general population who have specific similarities to that patient. The patient under study is called the case patient; the individuals chosen for comparative study are called the control patients. Case/control studies may utilize multiple control patients for every case patient to enable broader conclusions to be drawn, or to increase the level of certainty in regards to the results of the study.

A case/control study design is often highly useful for epidemiological research, as it allows researchers to investigate specific conditions, factors and outcomes, while controlling for the effects of demographic factors such as age and gender. The factors for which the researcher wishes to control in a case/control study design are used to specify the criteria used to match controls to case patients. So, for example, if a researcher wishes to control for the effects of gender, the criterion will specify that all control patients selected must be of the same gender as their case patient. The effects of age are usually controlled for by specifying that a control patient must have been born within a range of days before or after the case patient, and so on. Control patients are randomly selected from all members of the general population who have been identified as meeting the matching criteria.

THE STARTING POINT

The story of this program begins when this paper's author was first assigned the request to perform control selection on behalf of the researchers for a specific project. A program using a standard approach to this task was supplied for this purpose. Not long after being assigned the original request, a second request which would also require control selection was assigned to this paper's author, with the expectation of similar requirements to be ongoing.

The program provided was designed to perform the task in two stages. The first stage of the program would generate a data set containing every potential match between a case and a control as defined by the matching criteria, and assign a random number to each case-control combination. Once this stage was finished, the second stage of the program would select the appropriate number of control patients per case by selecting the control patients whose assigned random numbers were the highest.

This program was a very good starting point, and even though this program had been implemented on a different installation and operating environment, adapting the program to run in our in house environment was not a problem.

AN ASIDE: WHAT ABOUT PROC SURVEYSELECT?

It should be mentioned here that if the control selection process involves criteria for which distinct strata can be identified across discrete variables, the control selection process can be handled with the use of PROC SURVEYSELECT, a SAS procedure designed specifically for stratified random selection.

Some matching criteria, however, are not possible to implement directly with PROC SURVEYSELECT, because matching criteria do not always stratify the potential controls into unique or distinct categories. Some of the criteria for our project were like this.

These kinds of criteria do not mean that PROC SURVEYSELECT cannot be used. When an intermediate data set is created which contains all permissible case-controls pairings, like the data set the created by the program I received, every set of potential controls for a case can be treated as a separate stratum, and PROC SURVEYSELECT can be used. For the project on which we were working, using PROC SURVEYSELECT for the selection would not have made much of a difference, as it turned out.

THE SPANNER IN THE WORKS

When program provided select the controls corresponding to each case was first run, a problem emerged, as every attempt to run the program failed. Because of the sizes of the case and population data sets, the program would use up all of the storage allocated to it in the process of creating the interim data set which would hold all of the potential matches, and the process would need to be terminated.

Subdividing the problem was one possible approach, though not a particularly simple one, and not one that could necessarily be generalized and placed into a macro for repeated use. This was troubling, as we knew there would be more data requests requiring control selection; while the problem needed to be solved for this specific project, it would also need to be solved over and over again – with the attendant re-writing of code over and over again – unless a general approach to control selection which avoided the interim data set could be found.

The program provided for the task was not creating the interim data set without purpose; the interim data set was being created, and the problems were accordingly being encountered, because joining case and population data is necessary to perform the matching part of the process. Further to this, conventional SAS programming techniques for joining or merging tables via DATA steps or PROC SQL require the data sets involved to be explicitly defined as either input data sets or output data sets. Even DATA steps which have the same input and output data set specified in the DATA and SET statements work around this by writing the output data set to a temporary file, then replacing the input data set when the output data set has been successfully written. Because the process was designed to feed the output of one step into the input of the next set, it did not appear that any conventional SAS techniques for performing the join would be helpful here.

Because the problem - the use of the interim data set – was an immediate consequence of any table join which used conventional programming techniques, it became pretty clear that conventional programming techniques would be unlikely to result in achieving the goal of a re-usable, macro-suitable program which worked within the limited amount of resources available. Fortunately, recent versions of SAS include hash table support through the SAS's implementation of the hash object.

WHAT IS A SAS HASH TABLE?

In SAS, a hash table is a special type of temporary SAS data set designed to be created, accessed, and controlled by a DATA step program. SAS has added special objects designed to manage hash tables and to navigate through them, and associated methods for these objects. These objects have been integrated into the DATA step program language to allow hash tables to be created and used within a data step.

A hash table is composed of two different elements: key variables and data. If you think of a hash table as a city, key variables function as the street address which allow you to find the building where all of the specific data for that key can be found. Locating any data requires only its address – the key values which correspond to it; the hash object's.check() method will establish whether there is data stored at that location, while the .find() method will navigate there. SAS also provides the hash iterator object to simplify navigation, either by traversing multiple records in order (using the .prev() and .next() methods), by finding records by key (using the .setcur() method), or both.

From an efficiency perspective, hash tables will normally be the best choice for storing temporary data, because the hash table data is stored in memory rather than on disk. They also can offer significant advantages over previous techniques for data access at the record level, such as using SET or MODIFY with the POINT or KEY options to access data sets on disk; the most notable of these advantages may be the ability to combine reading, updating and adding records in a single method of data access. Data sets may be loaded into a hash object from disk, or saved from a hash object to disk; as a result of this, use of the hash object is often a suitable alternative to other record access techniques when a process is performing a large amount of disk access.

HASH TABLES AND THE PROGRAM DATA VECTOR

Hash tables have a special relationship with a DATA step's program data vector, or PDV - the set of all variables available to the DATA step. This relationship is special for two reasons. Firstly, records in a hash have the values of their variables (for the current hash table record) made available to the program data vector. Secondly, records in a hash table are immune to changes to the values in the program data vector unless the programmer instructs SAS to update the current hash table record.

Here is how this translates into the execution of a data step:

During any given pass through a data step, values are given to variables in the PDV in various ways. Values may be assigned directly, read into variables in the PDV as the result of a SET statement, updated behind-the-scenes if they are SAS-supplied (such as _n_, and the first. and last. values for BY variables), or even set to missing if the variable is referenced before any value has been supplied to it. Variables in a data step which are connected to a hash table can and will have their variables set in another way: when the hash table's current record is changed, the values in the connected data step variables are updated with the values from the hash table's new current record. Because these variables are connected to Program Data Vector variables, SAS always synchronizes the PDV versions of these variables to match the hash table record's values whenever a new hash table record is navigated to.

Conversely, variable values within the hash table itself are not updated as a result of a variable assignment, set statement, or any other update to a variable value within the data step itself. Values in specific hash table record can be explicitly overwritten with all values from the connected variables within the data step by using the hash object's .replace() method, but the only way that values in a hash table can be modified is by use of this or other hash object methods.

As with other DATA step programming techniques, hash table instructions and actions can be performed in whatever pattern is useful to the programmer. Accordingly, a hash table can be read from, written to, or navigated into whenever the programmer wishes to, with whatever order, combination, or frequency is most suitable.

CAN HASH TABLES HELP HERE?

Hash tables give the programmer fine-grained control over table access. This access gives the programmer considerable flexibility in joining and updating records. We can leverage the hash table's flexibility in regards to data access to successfully perform both the joining portion of the control selection process and the selection portion of the process within a single data step, provided that we can figure out how to get the joining process to play nicely with the selection process, and vice versa.

Simply using the hash table instead of an interim data set, however, will normally result in resource problems even more quickly than with creating an interim data set, because most systems on which SAS runs have more available storage on disk than in memory. Since the tool – hash tables – does not directly match the proposed process – create an interim dataset – the method will need to be modified if we intend to leverage the tool. Indeed, there is a general principle at work here: when considering the use of a new programming tool or technique, we need to avoid the assumption that the new tool and the current method will mesh well. Instead, we need to examine the fit closely, and potentially rethink our approach either to our choice of tools or our preferred process if they do not seem to work well together. Here, the problems with the existing process rendered it expendable, so it was clearly worth attempting to create a new approach using the powerful abilities hash tables provide.

AN INSTRUCTIVE FAILURE

With all of the above in mind, the search was on for assumptions which may have been made by previous implementations but which may not be required by a new implementation. The first insight arrived at was that there was nothing specific about the problem which required all of the controls to be selected simultaneously. If the controls were selected separately for each case in sequence, there would be no need to retain any data, other than the data corresponding to a single case's candidate controls, at any given time. If we only need to join one case to the population at a time, we do not need to retain all of the eligible matched records, because only the eligible records for the current case matter.

The fact that each case could have its controls selected separately suggested that the entire control selection step for a single case could correspond to a single iteration of a data step. The information used to match controls to a new case could be loaded at the beginning of the data step, the case could be test-matched to all members of the population during the mid-step processing, and the control selected could be finalized at the end. Loading the entire population data set into a hash table would be sufficient to facilitate this. With the population data loaded into the hash table, the DATA step could access it when and however it was needed in order to identify matches and select controls. The control selection process could then be performed entirely within a single data step.

At a conceptual level, a control selection process such as the one detailed above would look something like this:

```
data match_controls;
    if _n_=1 then do;
        [load population into hash table]
        end;
set cases;
[traverse entire population, identifying eligible controls for the case]
[select chosen control(s) for the current case from eligible controls]
run;
```

With a view towards just getting the required deliverable completed, a working program was created which loaded the population into a hash table, and traversed the population to select controls for each case in sequence, but it wasn't pretty. There were problems which prevented it from being a successful solution to the general problem. The most substantial problem was that the population data set was large enough that that the size of the individual hash records had to be reduced to 28 bytes in order to reduce the amount of memory needed to load the entire selection population enough to fit the amount of memory available. Reducing the record size required some ingenuity; for example, separate variables whose range of values allowed it were combined into a single variable to eliminate the extra two bytes a separate variable would require. Not only did this result in a program which was hard for other programmers to understand, but any other combination of different criteria could conceivably have made it impossible to implement in this way within the resources available.

While this program was a failure, however, it was by no means a total loss. For one thing, the hash table technology had proved itself, and I had a better working understanding of it to bring forward into any future attempts to solve the problem. It had also established that there was at least one different way of doing things which could succeed, which suggested that there might be more. Indeed, if anything, it suggested that the changes made to the way of performing the task had not been radical enough.

Beyond that, the fact that the failure was due to a problem with a specific and identifiable cause – it simply wasn't practical on an ongoing basis to rely on loading the population data set into memory – provided a clearer picture of what a successful approach would look like. A successful general solution would not load the larger of the two data sets into memory – the population data set – but would instead load the smaller one into memory – the cases data set. Was this even possible? Yes, it was – but it required one more insight to make it work.

A SURPRISING APPROACH LEADS TO SUCCESS

Any approach which would load the cases into memory, but would not load the potential controls, would be limited to accessing the information about one potential control at a time. For this sort of approach to work, we need to be able to consider potential controls in near-isolation. I say "near isolation" as RETAINing variables, for example, would allow us to access information from previous records. We already know that we cannot know which potential control is the selected control unless we have access to information on all of the other possible controls to do so.

Here is where having the right insight comes in: we may not be able to identify the selected control without examining all other potential controls, but we can identify a potential control which will not be selected by comparing one potential control to another. One of the two potential controls will have a randomly assigned value which is smaller than the other control's. If we can come up with a way to assign controls that lets us determine potential controls exclusively by comparing one potential control to another, we can avoid having to identify all of the potential controls for any one case simultaneously. In practice, we can. As long as we always remember the "winner" of the previous comparison to be pitted against the next "challenger", we will eventually have identified the potential control with the highest randomly assigned number.

To make this scheme work, we will need to support it in two key ways. Firstly, we will need to remember the current "winner" for each case as we make our way through the population data set. We already have the information about each case stored in our hash table, so we can store the current winner in the hash table (with the rest of the information about the case) easily enough. This simplifies our requirements even more, as we do not need to work out a scheme to RETAIN information from previous DATA step iterations. Secondly, the evaluation of whether a potential control is a match will need to be integrated into the process, because the matching status of a potential control will no longer be determined in advance or remembered after it has been compared to the current winner.

Since this approach was arrived at, it has been implemented successfully, validated, and used to provide requested deliverables, both for the original request and subsequent ones. For the purpose of discussion as well as for the availability of the SAS user community, a version of the code which performs the control selection using this algorithm has been provided in an appendix at the end of this paper.

DIGGING INTO THE CODE

The code has been provided in the form of a macro which can easily be brought into a program using the %INCLUDE statement. The provided code is also capable of generating random test data from a set of Canadian postal codes for demonstration purposes.

The macro code which has been wrapped around the actual selection code is, for the most part, straightforward, with several parameters being used as direct substitutions into the control selection code. A pair of %DO loops are used to handle the support for selected multiple distinct controls in a single pass; the first one ensures that distinct variables are created for each distinct control in the data set which will be loaded into the hash object, while the second generates the code within the actual selection process which handles the control selection for each distinct control separately.

The macro does include one slightly unusual programming technique in allowing the user to specify actual program code to be injected into the code generated by the macro in three of the parameters. A couple of these parameters are designed to receive comparison clauses which are substituted directly into IF … THEN statements, while a third parameter is designed to receive the name of a macro which is called by the control selection macro to generate the data step code used to determine whether a potential control is an eligible match. This macros-calling-macros approach is more or less necessary here, as the matching criteria may easily be too complicated to allow the SAS code which evaluates matches to be generated automatically.

The main DATA _NULL_ step which performs the actual selection makes use of the LINK and RETURN statements to organize the code into smaller routines which make the code easier to read and understand. In particular, the handleSinglePopRec routine separates the program code that is run to perform the combined join-and-select process for a single population record over the complete set of cases from the code to iterate through the population.

In practice, the implementation performs all parts of table joining manually. All possible joined rows are generated because the handleSinglePopRec routine is run once for every population record, and cycles through all appropriate case records. In the course of cycling through all appropriate population/case combinations, each joined pair is evaluated for inclusion by using the eligibility check provided by the user, and the output data is conditionally saved based on whether it can be eliminated as a control.

The inclusion or elimination of a case-control pairing is worth a closer look. If the generated case-control pair is eligible to be selected, the program generates the random number for that pairing which will be used to determine whether that control is selected. If the pairing is the first eligible pairing, it is considered to be the provisionally selected case-control pairing by default; otherwise, it has its random number compared to that of the provisionally selected pair. If the generated pairing can be eliminated because its generated random number is too low, no further operations are necessary, and the next pair can be considered. If the generated pairing has a higher random number, the important values for the generated pairing (in particular, the unique identifier and its random number for comparison) are placed into the variables shared with the hash table, and the replace() method is used to save the values for the new provisionally chosen control.

This ability to both read and save data within the same process is not only very helpful in allowing us to avoid the interim dataset, it is essential for the approach taken here. Because the hash table provides us with a massive short-term memory to work with, we can remember enough about each case's control assignments that the control assignment is essentially performed in parallel, rather than separately in sequence. And it is because we are not limited to performing the control selection operations separately in discrete sequences that we are now able to perform the control selection operations in the order that works for us – that is, by considering one population member at a time. This interleaving of parallel operations will not be needed for every problem, but it is nonetheless a powerful tool which may be useful in other situations.

In regards to efficiency, the set of all possible joined rows is generated by traversing through every row of case data in the hash table once for every population record. As a result of this, both input data sets have every record read from disk exactly once: the cases data set when it is read into the hash table, and the population data set has each record read in order because of the SET statement. Thus, very little reading from disk is required, which reduces both resource usage and the program's run time.

One simple but significant optimization has been allowed for in the code. If the cases data set is in sorted order by one of the fields used for the matching process, a pair of comparison expressions may be included. If these comparison expressions are included, the data step uses them to perform a quick search over the sorted data set to identify a subset of the cases data set records which will contain all case records which will match the potential control. Because cases which fall outside of this subset are known to not be eligible, they do not need to be processed at all. If the comparisons are chosen wisely, 75-90% of the processing which is done otherwise for a given potential control can be eliminated. As a result, run time is further reduced when these comparison expressions are provided.

Further improvements to the macro are undoubtedly possible. In particular, the need for the program to run in a SAS 9.1 environment has resulted in a few workarounds for useful functionality available in SAS 9.2, most notably the .setcur() functionality. This is why the _position variable is created in the temporary data set: it is a way of enabling navigation to specific records without some of the conveniences SAS provides for hash iterators since SAS 9.2.

One more aspect of the selection process is worthy of discussion, and that is the requirement that a specific case's selected controls be different from one another. (Control selection requirements can be more stringent than this, and require that no control be selected for more than one case; the approach taken here would need additional adaptation to be used in this case.) The use of random selection does not necessarily prevent a control from being randomly selected twice for the same case if multiple controls per case are being selected per case, because each control is being selected independently. In fact, if controls are matched sufficiently closely to cases, and if the number of cases is high enough, a truly randomized selection process almost guarantees that some controls will be selected twice.

This program minimizes the likelihood of that happening by generating a random number once for each case-control pairing, regardless of how many controls are being selected. The random number for comparison is generated is in the range of 0 to N where N is the total number of controls. After each control is selected, the number for comparison is incremented by 1. If adding 1 results in a number for comparison which is out of range, N is subtracted to place the number back into the 0 to N range. This has the effect of placing the number for comparison in every distinct range from 0...1 to N-1...N exactly once for each case-control pairing being generated as part of the process. The transformation has the effect of making a specific pairing more or less likely to be selected at various points in the process without causing the overall likelihood of a given case-control pairing being selected overall to be non-random (because the same process is being applied to all pairings). Because a case-control pairing is much more or less likely to be selected for a given control, the likelihood of repeat selections is greatly reduced. If repeat selections do occur, they can sometimes be eliminated by changing the random number seed; conversely, an additional control can be included in the selection process, and only used when needed to replace a duplicated control elsewhere.

CONCLUSION

The development of this macro has proven to be quite instructive to its author. I've gained experience with and a better understanding of hash tables, as well as how they can be leveraged to improve program efficiency and resource usage, and unlock new programming techniques. I've also seen firsthand how trying new things as a programmer can be useful even when they don't initially work, as long as your analysis allows you to learn from failed attempts. I trust that these aspects of my experience, as well as others, will be useful and informative to you in your SAS programming in the future.

ACKNOWLEDGMENTS

The author would like to thank Peter Eberhardt for detailed feedback on the contents of this paper, Arthur Tabachneck and Shelley Derksen for additional feedback, and the other Beyond the Basics presenters for their support and encouragement.

RECOMMENDED READING

- Hash Crash and Beyond – Dorfman, P., Shajenko, L., and Vyverman, K. (SAS Global Forum paper 037-2008, available at http://www2.sas.com/proceedings/forum2008/037-2008.pdf)
- The SAS® Hash Object: It's Time To .find() Your Way Around – Eberhardt, P. (SAS Global Forum paper 168-2011, available at http://support.sas.com/resources/papers/proceedings11/168-2011.pdf)

CONTACT INFORMATION

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

Name: Craig Kasper
Enterprise: Government of Manitoba, Manitoba Health
Address: 4th Floor, 300 Carlton St.
City, State ZIP: Winnipeg, Manitoba, Canada R2C 1Z7
Work Phone: 204-788-6354
E-mail: craigkas@mts.net

Appendix 1 – the SelectControls macro and sample code

The most current version of this source code will also be available at the SASCommunity.org page for this paper, at http://www.sascommunity.org/wiki/Implementing_Control_Selection_Using_Hash_Tables:_A_Case_Study.

```
/*  Generate sample data sets for demo purposes from a postal codes table. */
data work.test_pop_for_cc(keep=postal_code fsa age gender uniqueid)
     work.test_cases_for_cc(keep=case_postal_code case_fsa case_age case_gender
case_uniqueid);

     retain uniqueid 0;
     set work.pcodes(keep=postal_code);
       if _n_=1 then do;
          _ctrl_randseed=round(rand('UNIFORM')*2147483647,1);
           _ctrl_randseed_str=put(_ctrl_randseed,10.);
          call streaminit(_ctrl_randseed);
          put "NOTE: stream initialized with seed " _ctrl_randseed_str;
          end;
     case_age=round(rand('UNIFORM')*70,1)+20;
     case_gender="M";
     if rand('UNIFORM')*2.1 > 1 then gender="F";

     uniqueid+1;
     fsa=substr(postal_code,1,3);
     case_postal_code=postal_code;
     case_fsa=fsa;
     case_uniqueid=uniqueid;
     output work.test_cases_for_cc;

     age=case_age;
     gender=case_gender;
     output work.test_pop_for_cc;

     popcount=round(rand('UNIFORM')*50,1)+25;
     do iterate=1 to popcount;
          age=round(rand('UNIFORM')*70,1)+20;
          gender="M";
          if rand('UNIFORM')*2.1 > 1 then gender="F";
          uniqueid+1;
          output work.test_pop_for_cc;
     end;
run;

proc sort data=work.test_cases_for_cc;
       by case_age case_uniqueid;
run;

%macro SelectControls(TempDataset=,CaseDataSet=,CaseID=,PopDataSet=,PopID=,
CtrlCount=,EligibilityCheck=,Output=,CaseBeforeCheck=,CaseAfterCheck=,RandSeed=);

/* This is a trimmed down program version.  Parameter-checking code and other
conveniences have been omitted in order to save space.
Parameters:
    TempDataset: The macro creates one temporary data set.  Use this parameter to
specify it by name.
    CaseDataSet: Use this parameter to specify the data set containing case patients.
    CaseID: Use this parameter to specify the name of the variable in the case patient
data set which contains the patient's unique identifier.  This identifier should be
created or chosen to identify the case patient within the population.
    PopDataSet: Use this parameter to specify the name of the population data set.
    PopID: The macro requires the name of the variable in the population data set
which contains the unique identifier for each individual in the population.
    CtrlCount: The total number of controls to generate for each case patient.
```

```
    EligibilityCheck: The name of a macro which contains data step code to determine
whether a member of the population is eligible to be a control for a specific case.
    Output: The name (including library name if needed) of the data set where the
output of case-control assignment will be saved.
    CaseBeforeCheck: an optional comparison clause which, if provided, is used to find
a starting point for cycling through the cases later than the beginning of the table,
when appropriate.  If this clause is provided, the case data set must be sorted by all
variables used in the comparison.  The comparison should be chosen to return true if
all of the cases for which a population member is eligible for matching will be found
later in the cases data set, and false otherwise.  Comparisons provided for this
parameter should use the two-letter comparison operators (ge/le/eq and the like)
instead of the symbolic ones (>, <, = and combinations of them).
    CaseAfterCheck: an optional comparison clause which, if provided, is used to find
an ending point for cycling through the cases later than the beginning of the table,
when appropriate.  The comparison should be chosen to return true if all of the cases
for which a population member is eligible for matching will be found earlier in the
cases data set, and false otherwise.  The same prerequisites (sorted data set, etc.)
that apply to the CaseBeforeCheck parameter apply to this one too.
    RandSeed: Random number generator seed.  Set this to 0 if you want the random
number generator seed to be generated using the current system time, or to a number
between 1 and 2147483647 to use a pre-chosen random number generator seed for
replicability purposes.  If you leave the RandSeed parameter blank, the program will
randomly generate a random number generator seed and print it to the log.  */

/* Prerequisites:
- case and population datasets, and the temporary data set's library, must exist.
- It is strongly recommended that the cases data set and the population data set
  should not have any variable names in common.
- Certain additional variables are created in the temporary data set and during the
  assignment process.  Variables with the same variable names may be overwritten.
  All additional variables all have names which start with an underscore character;
  variable names which do not start with an underscore will not be overwritten.  */

/* Step 1: prepare the data set which will be loaded into the hash object.  This data
set contains all of the case data set's fields plus fields to contain all of the
temporary data, as well as the controls selected by the process. */

data &TempDataset.;
     set &CaseDataSet.;
         _position=_n_;
/* Add variables to hold the current provisionally selected control's ID and its
generated random number + a flag indicating whether no control has been selected yet.
Each of these variables is created for every distinct control to be selected.  */
     %do cnum=1 %to &CtrlCount.;
        length _CtrlID&cnum. 8. _CtrlRandnum&cnum. 8. _CtrlAssigned&cnum. 4.;
        _CtrlID&cnum.=.;
        _CtrlRandnum&cnum.=.;
        _CtrlAssigned&cnum.=0;
     %end;
     _numeligibleforselection=0;
run;

/* the main selection process follows. */
data _null_;

/* The line below pre-creates the variables for the cases data in the Program Data
Vector so that they will not be explicitly created before loading the cases into the
hash object without actually SETting the data table.  */
     if 1=0 then set &TempDataset.;

if _n_=1 then do;
    /* Initialization: handle seeding the random number generator */
    if "&randseed."="" then do;
```

Implementing Control Selection Using Hash Tables: A Case Study, continued

```
            _ctrl_randseed=round(rand('UNIFORM')*2147483647,1);
            _ctrl_randseed_str=put(_ctrl_randseed,10.);
        end;
    else do;
        _ctrl_randseed_str="&randseed.";
        _ctrl_randseed=input(_ctrl_randseed_str,10.);
         end;
    call streaminit(_ctrl_randseed);

    put "NOTE: stream initialized with seed " _ctrl_randseed_str;
    ttime=time(); put "NOTE: Control assignment started at " ttime time.;

    /* Initialization: prepare our hash object and an iterator. */
    declare hash _hcontrol(dataset:"&TempDataset.", ordered:'a');
    _hcontrol.definekey("_position");
    _hcontrol.definedata(All:'YES');
    _hcontrol.definedone();

    declare hiter _hsearch;
    _hsearch= _new_ hiter('_hcontrol');
end;

/* Detect the end of the population data set and exit manually when we hit it. */
if endOfSet=1 then do;
    _hcontrol.output(dataset:"&output.");
    stop;
    end;

/* Start of the main loop: */
     SET &PopDataSet. end=EndOfSet;
/* For each coverage record, handle the match-and-select process: */
     link handleSinglePopRec;

/* To show progress, send the log _n_ and a timestamp every 20,000 records. */
     if mod(_n_,20000 )=0 or EndOfSet=1 then do;
          ttime=time(); put "NOTE: " _n_ " records processed at " ttime time.;
          end;
return;  /* end of main body of data step */

handleSinglePopRec:

/* an optimization: find better start and end points for traversing through cases */
    _rc=_hsearch.last();
    _numcases=_position;

   _findstart=-1; /* -1 means "Starting point not found" */
   _findend=_numcases;

   do _position= _numcases to 1 by -int(_numcases/25);
        _hcontrol.find();
/* If comparison have been provided allows us to eliminate records at the start and
end of the cases set: */
        %if "&caseBeforeCheck." ne "" %then %do;
        if &caseBeforeCheck. and _findstart eq -1 then _findstart=_position;
        %end;
        %if "&caseAfterCheck." ne "" %then %do;
        if &caseAfterCheck. then _findend=_position;
        %end;

        end;

    _findstart=abs(_findstart);
/* Convert -1 ("not found") to starting position 1, making no change otherwise */
```

```
    do _position=_findstart to _findend by 1;
        rc=_hcontrol.find();

/* check to see if the population member is eligible to be a control for the case */
        eligibility=0;
        link eligibilityCheckStep;

/* if so, update the count of eligible population members for the control */
        if eligibility=1 and &popID. ne &caseid. then do;
           _numeligibleforselection+1;
           _hcontrol.replace();
           comparisonVal=rand('UNIFORM') * &CtrlCount;

/* For every distinct control being assigned: */
           %do cnum=1 %to &CtrlCount.;
/* For the case we are examining here, if there is no candidate control patient
assigned (in other words, if the assigned flag is 0), the current population member is
assigned by default. */
                if _CtrlAssigned&cnum.=0 then do;
                    _CtrlAssigned&cnum.=1;
                    _CtrlRandnum&cnum.=ComparisonVal;
                    _CtrlID&cnum.=&popID.;
                    _hcontrol.replace();
                    end;
/* Otherwise, replace the previous candidate control patient and random number with
the current candidate/random number if the current random number is higher.  */
                else if (comparisonVal gt _CtrlRandnum&cnum.) then do;
                    _CtrlRandnum&cnum.=ComparisonVal;
                    _CtrlID&cnum.=&popID.;
                    _hcontrol.replace();
                    end;
                else do;
/* otherwise do nothing. */
                    end;
/* Rotate the random number generated by 1 */
                comparisonval+1;
                comparisonval=mod(comparisonval,&CtrlCount.);
            %end;
            end;  /*** if eligible ***/
        end;  /*** do while ***/
return;

eligibilityCheckStep:
       %if "&eligibilityCheck." ne "" %then %do;
       %&eligibilityCheck.;
       %end;
return;

run;
%mend;

%macro CheckElig();
/* match criteria: born within 5 years of each other, and living in the same FSA*/
       if abs(case_age - age) le 5 and fsa eq case_fsa then eligibility=1;
%mend;

option mprint;
%SelectControls(TempDataset=work.tempdata,CaseDataSet=work.Test_cases_for_cc,
caseid=case_uniqueid, PopDataSet=work.test_pop_for_cc,Popid=uniqueid,CtrlCount=5,
eligibilityCheck=CheckElig, output=work.ctrl_sample, CaseBeforeCheck=(case_age-age) le
-6,CaseAfterCheck=(case_age-age) ge 6, randseed=13572468);
Option nomprint;
```