Getting Started with the DATA Step Hash Object

Jason Secosky, SAS Institute Inc., Cary, NC
Janice Bloom, SAS Institute Inc., Cary, NC

ABSTRACT

A time consuming part of many SAS® programs is looking up a value from one data set in another data set. SAS 6 lookup methods, like SET with KEY= or a format, are good for many applications. However, in SAS 9, there is a better tool, the DATA Step hash object. The hash object provides a fast, easy way to perform lookups without sorting or indexing. This paper introduces the hash object, examines a few common hash object methods, compares SAS 6 techniques with the hash object, and builds rules of thumb for when to apply this new technology to your programs.

INTRODUCTION

Looking for a value in one data set when given a value from another is a core operation for many SAS programs. Of critical interest is the data associated with the value being looked for. For instance, given a customer id, one would like to retrieve a customer's address or other information from a data set.

With many lookup mechanisms, the value being looked for is called a "key". The "key" provides a means to access the data, just as a key for a lock provides a means to access what was locked. In SAS, there are several key lookup mechanisms: the DATA step SET statement with the KEY= option, the PUT and INPUT functions using a format or informat, match-merging with a MERGE statement, and other methods. The hash object is a new method in SAS 9 which solves several execution slowdowns with existing lookup methods.

The hash object is an in-memory lookup table accessible from the DATA step. A hash object is loaded with records and is only available from the DATA step that creates it. A hash record consists of two parts: a key part and a data part. The key part consists of one or more character and numeric values. The data part consists of zero or more character and numeric values.

Once a hash object is loaded with records, a lookup occurs by passing a key to the hash object's FIND method. If a record with the particular key is found, the data part of the record is copied into DATA step variables. In addition to being able to add and find records, there are methods to replace records, remove records, and output records to a data set.

Programs benefit from using the hash object for the following reasons:

- Key lookup occurs in memory, avoiding costly disk access.
- When a key lookup occurs, only a small subset of the records are searched.
- The key and data parts of a record can consist of more than one value, removing the need to format and concatenate values to construct the key and data parts.
- The hash object allocates memory as records are added. That is, the hash object only allocates as much memory as it needs and the number of records that can be stored is only limited by the amount of memory available to SAS.
- When loading a hash object from a data set, the data set need not be sorted or indexed.

The first section of this paper introduces the functionality of the hash object with an example that loads a hash object with records and performs lookup operations without getting into the gory details of hashing. Next, adding, replacing, and outputting records are covered, followed by common messages and missteps. Memory considerations and a comparison of the hash object with existing lookup methods is covered at the end of the paper in order to help guide design decisions.

HASH OBJECT LOAD AND LOOKUP

A common operation with a hash object is to load a small data set into the hash object and perform lookups using keys loaded from a large data set. Neither data set is required to be sorted or indexed. Not having to sort or index shortens execution time. When records are added to the hash object, the keys must be unique. Ensuring keys are unique may require including more than one variable in the key.

The small program below demonstrates how a hash object is declared, loaded, and how lookups occur. The first DATA step creates a data set that contains participant names, their gender and drug treatment code. The second

DATA step creates a data set holding patient weights. We would like to merge the name, gender, and treatment with the weight data for an analysis that occurs later. The third DATA step uses a hash object to merge the treatment for each patient with their weight data.

```
data participants;
  input name $ gender:$1. treatment $;
datalines;
John     M Placebo
Ronald   M Drug-A
Barbara  F Drug-B
Alice    F Drug-A
;

data weight(drop=i);
  input date:DATE9. @;
  do i = 1 to 4;
    input name $ weight @;
    output;
  end;
/* For brevity, only two dates are listed below */
datalines;
05May2006 Barbara 125  Alice 130  Ronald 170  John 160
04Jun2006 Barbara 122  Alice 133  Ronald 168  John 155
;

data results;
  length name treatment $ 8 gender $ 1;

  if _N_ = 1 then do;
    declare hash h(dataset:'participants');
    h.defineKey('name');
    h.defineData('gender', 'treatment');
    h.defineDone();
  end;

  set weight;
  if h.find() = 0 then
    output;
run;

proc print data=results;
  format date DATE9.;
  var date name gender weight treatment;
run;
```

Output:

Obs	date	name	gender	weight	treatment
1	05MAY2006	Barbara	F	125	Drug-B
2	05MAY2006	Alice	F	130	Drug-A
3	05MAY2006	Ronald	M	170	Drug-A
4	05MAY2006	John	M	160	Placebo
5	04JUN2006	Barbara	F	122	Drug-B
6	04JUN2006	Alice	F	133	Drug-A
7	04JUN2006	Ronald	M	168	Drug-A
8	04JUN2006	John	M	155	Placebo

During the first iteration of the implicit loop of the third DATA step, when _N_ is 1, the program creates and loads the hash object. There are four steps to define a hash object.

1. **Create and name the hash object.** The DECLARE statement creates an object. DECLARE can be abbreviated as DCL. To create a hash object, the keyword HASH is specified after DECLARE. In order to manipulate the hash object in the code, it must be given a name. The name is given after the keyword HASH. In this case, the name is H. Initialization options are specified within the parentheses. The DATASET: argument tag is specified to load the hash object from a data set, PARTICIPANTS.
2. **Specify key variables.** The DEFINEKEY method is used to specify one or more key variables. Key variables are DATA step variables that contribute to the key part of a hash record. At least one key variable must be specified; otherwise key lookups could not occur because there would be no key values to search. When a method is called that requires a key, like FIND, the values in the key variables are used to perform the lookup.
3. **Specify data variables.** The DEFINEDATA method is used to specify zero or more data variables. Data variables are DATA step variables that contribute to the data part of a hash record. When a lookup method, like FIND, successfully finds a key, the data variables are filled with the data part of a hash record.
4. **Complete the definition.** The definition of a hash object is concluded by calling the DEFINEDONE method. When DEFINEDONE is called, DEFINEKEY and DEFINEDATA can no longer be called. If DATASET: was specified on the DECLARE statement, the data set is loaded into the hash object.

Now that the hash object is created and loaded, we can lookup keys to retrieve data from the hash object. Lookups are typically performed with the FIND method. FIND uses the values of the key variables to determine if a record with this key exists in the hash object. If a record exists, the data part is copied into the variables specified when DEFINEDATA was called. FIND returns zero to indicate a key was found, otherwise a non-zero value is returned.

In this program, we read an observation from WEIGHT with the SET statement. The observation contains a date, name, and weight. The FIND method determines if the hash object contains a record with a key that is the same value as NAME. If a record is found, the variables GENDER and TREATMENT are filled and FIND returns a zero. At this point, the treatment and weight data have been combined, so an observation is output to the data set RESULTS by using the OUTPUT statement.

If a record is not found, FIND returns a non-zero value, the program returns to the start of the implicit DATA step loop, the next observation is read from WEIGHT, and another hash object lookup occurs.

This program demonstrates the steps required to create and load a hash object. It also shows how to perform a lookup with a hash object. The MERGE statement could have been used to write an equivalent program. In order for MERGE to operate, the two input data sets must be sorted by or have an index on the key variable, NAME. In many applications, either one or both of the data sets are too large to be sorted within a required time window. The hash object is able to perform the merge without any explicit sorting or indexing.

ADD, REPLACE, AND OUTPUT

This section elaborates on three more hash object methods, ADD, REPLACE, and OUTPUT. The ADD method adds records to a hash object. When ADD is called, the values of the key and data variables are copied into a new hash record and the record is added to the hash object. If a record exists with the key being added, ADD returns a non-zero value. When the DATASET: argument tag cannot be used for one reason or another, using the SET statement and calling ADD in a loop may get the job done.

The REPLACE method replaces existing data in a hash record with new data. First, REPLACE looks for a record with a key equal to the values of the key variables. If a record is found, the data in the hash record is replaced with the current data in the data variables. If a record is not found, the value of the key and data variables are copied into a new hash record and the record is added to the hash object. The next example calls ADD and REPLACE to manipulate data in a hash object.

Given a list of soccer players and the time when they scored a goal, we would like to concatenate each player's goal times into a comma separated list and output the results to a data set. PROC MEANS is traditionally used to perform aggregations, yet PROC MEANS cannot compute aggregations of character values. Instead, we use a hash object.

In this program, a hash object is created with the key being the player's name and the data being the times they have scored. FIND is called to determine if a player is in the hash object. If the player is not found, a new record is added that contains the player's name and the first time they scored. If the player is found, the time they scored is concatenated to the other times they scored. The concatenated list is copied into the hash object by calling REPLACE.

```
data goals;
  input player $ when & $9.;
datalines;
Hill   1st 01:24
Jones  1st 09:43
Santos 1st 12:45
Santos 2nd 00:42
Santos 2nd 03:46
Jones  2nd 11:15
;

data _null_;
  length goals_list $ 64;
  if _N_ = 1 then do;
    declare hash h();
    h.defineKey('player');
    h.defineData('player', 'goals_list');
    h.defineDone();
  end;

  set goals end=done;
  if h.find() ^= 0 then do;
    goals_list = when;
    h.add();
  end;
  else do;
    goals_list = trim(goals_list) || ', ' || when;
    h.replace();
  end;

  if done then
    h.output(dataset:'goal_summary');
run;

proc print data=goal_summary;
run;
```

Output:

```
Obs    player    goals_list

 1     Hill      1st 01:24
 2     Santos    1st 12:45, 2nd 00:42, 2nd 03:46
 3     Jones     1st 09:43, 2nd 11:15
```

When the hash object is created, the key variable is PLAYER and the data variables are PLAYER and GOALS_LIST. The reason PLAYER is part of the data is discussed later. GOALS_LIST stores the list of the times each goal was scored and is the data to be updated when REPLACE is called.

For each observation read from GOALS, the FIND method determines if a record exists in the hash object for the player. If a record does not exist, we need to initialize the value of GOALS_LIST to the time of their first goal and call ADD to add the player to the hash object. If a record exists, FIND would have filled GOALS_LIST with a list of all the goals scored. We need to concatenate the latest goal scored to GOALS_LIST. The concatenation is performed and REPLACE is called to replace the player's old list with the new list we created in GOALS_LIST.

Once the input data set has been exhausted, the OUTPUT method writes the data portion of every record in the hash object to the data set listed after the DATASET: argument tag. In this case, the data set GOAL_SUMMARY contains observations that contain the player's name and their summary of goals. OUTPUT only outputs data variables. This is the reason PLAYER was defined as a data variable, so that it would appear in the final output.

OUTPUT can create a data set sorted by the key variables. Sorted output is produced when the ORDERED: argument tag is specified when the hash object is created. The value of ORDERED: can be 'yes' or 'a' for an ascending sort order, 'd' for a descending sort order, or 'no' for no sorting. If the ORDERED: argument tag is not specified, no sorting will occur. In the context of the prior example, to sort the output data set by player name, the code would be:

```
data _null_;
  length goals_list $ 64;
  if _N_ = 1 then do;
    dcl hash h(ordered:'a');
    ...
```

String aggregation could have been written using DATA step arrays or with the MODIFY statement with the KEY= option. The hash object solution is straight forward and succinct. The solution also demonstrates how the hash object is not sized. The hash object can grow to store as many records as fit into memory.

This section described the ADD, REPLACE, and OUTPUT methods. There are several other hash object methods and there is another object type, the hash iterator. A hash iterator is used to loop over the records in a hash object. Documentation for the hash iterator and methods not described in this paper can be found in the SAS 9 Online Doc [2]. Hash object and iterator examples can be found in the samples at support.sas.com [3].

MESSAGES AND MISSTEPS

The samples presented so far represent a "perfect world" scenario. They are purposefully simple and meant to illustrate the basics. But what may happen when creating new code in the "real world"? When learning a new coding technique, unexpected NOTES, WARNINGS and ERRORS may appear in the SAS log. Without a strong footing in the terminology, deciphering what messages mean and what should be done about them may be difficult. This section is meant to present a few of the more common messages you may see in the SAS log as you begin to practice writing your own hash code.

1. NOTE: VARIABLE XXXX IS UNINITIALIZED.

When running the two samples presented in this paper as written, this NOTE is written to the log. The SAS compiler does not see variable assignments in the hash object or hash iterator. If there is no explicit assignment statement for the variables specified in the DEFINEKEY and DEFINEDATA methods in the program itself, a NOTE is written to the log that the variables are not initialized.

To avoid these NOTES, you can use the CALL MISSING routine to initialize variables to missing. See the statement in bold in the sample below.

```
data results;
  length name treatment $ 8 gender $ 1;

  if _N_ = 1 then do;
    declare hash h(dataset:'participants');
    h.defineKey('name');
    h.defineData('gender', 'treatment');
    h.defineDone();
    call missing(gender, treatment);
  end;

  set weight;
  if h.find() = 0 then
    output;
run;
```

As an alternative to CALL MISSING, you can avoid the note about variables not being initialized by assigning an initial value to the affected variables with assignment statements or specify the SAS system option NONOTES to request that no NOTES are written to the SAS log.

2. ERROR: UNDECLARED DATA SYMBOL FOR XXXX FOR HASH OBJECT AT LINE N COLUMN M.

When the DEFINEDONE method executes, all of the variable names passed to DEFINEKEY and DEFINEDATA must exist in the PDV. If one does not, this message is output. Passing a variable name to DEFINEKEY or DEFINEDATA

does not add the variable to the PDV. Other programming logic, such as a LENGTH statement, must create the variable. Without the variable in question on the PDV, the hash object has no variable to retrieve the key or data value from.

If you encounter this message when creating your own hash code, make sure you have a LENGTH statement for the variable names passed to DEFINEKEY and DEFINEDATA and have given those variables an initial value as well. To replicate these messages with the code sample above, comment out the LENGTH statements and the CALL MISSING routine and resubmit.

3. ERROR 559-185: INVALID OBJECT ATTRIBUTE REFERENCE XX.YY.

In component object syntax, a pattern of XX.YY indicates an object attribute, where XX represents the hash object's name and YY represents an attribute. If a method call is missing its parentheses, the compiler interprets the code as an attribute and check the attribute table for an attribute of that name. If one is not found, the error "Invalid object attribute reference" is written to the log. An attribute that is not found in the attribute table also generates this error. To replicate this message using the sample above, remove the parentheses after the FIND method and resubmit.

4. ERROR 557-185: VARIABLE XXXX IS NOT AN OBJECT.

This error is generated when you misspell the hash object's name in method calls. To replicate this error using the sample above, misspell the name of the hash object H in the one of the methods.

5. DUPLICATE KEYS

When records are added to the hash object, the keys must be unique. When using DATASET: to load a hash object from a data set, if the data set contains duplicate keys the first record with the specific key value is added and no message is output. In SAS 9.2, the DUPLICATE: initialization parameter gives control over whether an error is produced when a duplicate key is encountered or whether the first or last duplicate is added.

Adding a duplicate key value to a hash object with the ADD method outputs a duplicate key error to the SAS log, if the return code of ADD is not assigned to a variable. If the return code is assigned to a variable, the variable holds a non-zero value and the error is not output to the log. In general, a return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log. While a component object program can be written without assigning a return code to a variable, it is suggested as "best practice" to assign the return code to a variable. Assigning and checking the return code of a method avoids any risk of an unexpected condition triggering an error message in the log.

MEMORY CONSIDERATIONS

When loading a hash object, if the hash object fills the memory available to SAS, the load fails. What should be done at this point? This section suggests way to reduce hash object memory usage. This section also covers how to estimate the number of records that will fit into a hash object, in order to determine if a hash object solution is feasible given the available memory.

DESIGN WITH MEMORY IN MIND

Designing a program to use less memory with a hash object ensures that the program both executes and does so with an efficient use of resources. This section presents several ways to reduce the amount of memory allocated by a hash object.

The first suggestion is to place as little data as necessary into the hash object. Designing a program to load smaller data sets into the hash object, then performing operations as observations are read from the larger data set, saves memory. Some programs can be written to load the larger data set into the hash object, yet this takes longer to execute and consumes more memory.

Often a smaller data set is a transaction data set and is roughly the same size each time the program executes and the larger master data set slowly grows over time. If the master data set were to be loaded into the hash object, the hash object size also grows over time and may not fit into available memory at some point in the future. A memory efficient design that plans for future growth loads the smaller, transaction data set into the hash object.

If the master data set must be loaded into the hash object, consider writing the code such that only a subset of observations is required to be loaded. This ensures all the data fits into the hash object. An observation filter can be created using a PROC SQL or DATA step view. The view can be loaded into the hash object with the DATASET: argument tag during hash object creation. Views have also shown to be helpful in renaming variables that are used by the hash object, so they do not conflict with other variables in a DATA step.

A final option is to make the hash object record as small as possible. For instance, if a value to be stored could have the value "yes" or "no", then use a character variable of length 1 to store "y" or "n" in the hash object instead of using a numeric variable to store 1 or 0. A numeric variable is always 8 bytes in a record.

ESTIMATING MEMORY USAGE

The size of a hash record is roughly the sum of the sizes of values being placed into the record. For instance, if a key is composed of two numeric values and the data is composed of a numeric value and a character value of length 40, the record size is roughly 8+8+8+40, or 64 bytes. In SAS 9.2, the hash object ITEM_SIZE attribute returns the size of a hash record.

If two million 64 byte records are loaded into the hash object, this takes roughly 128MB. If the SAS system option MEMSIZE is set larger than 128M and the machine can support executing SAS with at least 128MB of memory free when the hash object is loaded, the hash object load is successful.

In order to determine the amount of memory available to SAS, the system option XMRLMEM can be retrieved. XMRLMEM is an undocumented diagnostic option that reports the amount of memory available to SAS without involving the operating system's virtual memory. While a rough number, it gives a sense to the number of records that fit into the hash object. The DATA step below outputs the value of XMRLMEM.

```
data _null_;
  amt = getoption('xmrlmem');
  put amt=;
run;
```

Another way to estimate a hash object's memory requirements is to see how much memory the hash object takes with fewer records. This can be done by executing OPTION FULLSTIMER; before executing a DATA step. Turning on FULLSTIMER causes memory statistics about a step to be output to the SAS log after the step executes. If half of the records are loaded into the hash object, multiplying the amount of memory reported by FULLSTIMER by two roughly yields the memory required for the full job. This is a rough estimate since the memory usage reported includes the memory needed to execute the non-hash object portions of the DATA step.

COMPARISON WITH EXISTING TECHNIQUES

There are several key based lookup methods in SAS. Each has their pros and cons. This section compares the hash object with several of these techniques, in order to suggest guidelines when one lookup mechanism may be better than another. When considering each method, the performance is affected by the amount and type of disk I/O they perform. If the disk is not fragmented, random I/O is slower than sequential I/O and some methods perform more disk accesses than others. Also, the feasibility of using a method may be determined by the memory requirements of the mechanism.

SET WITH KEY=

When the DATA step's SET statement is used with the KEY**=** option, the DATA step uses the values in one or more DATA step variables to lookup the location of an observation with a data set index. The lookup searches a disk based data structure to determine which observation should be read. Programs that merge data from a transaction data set with values from a master data set usually maintain an index on the master data set. An index on the master data set enables sharing the index construction and maintenance cost with other applications that utilize the index. Programs that use SET with KEY**=** use little memory because only one observation from the master data set is stored in memory.

A hash object is similar to KEY**=** lookups, except the lookup occurs in memory instead of on disk. When combining master and transaction data with a hash object, a sequential pass is made through both the master and transaction data sets. This is different than using KEY**=**. Often indexed data sets are not sorted, resulting in random access of the master data set when using KEY**=**. Random I/O and the I/O required to access the index slow KEY**=** and other index based solutions. Index creation and keeping the index up-to-date can also take a significant amount of time. If the transaction data set fits into a hash object, a hash object solution is faster.

FORMATS

Formats and informats are traditionally invoked to format values for printing or to convert input text to numeric values. Formats and informats, can also be utilized as lookup tables. Format lookup tables can be constructed from a data set or view using PROC FORMAT with the CNTLIN**=** option. Formats support mapping one value to another value and mapping a range of values to one value. Once the format or informat is built, a lookup is performed by using the

PUT or INPUT function. The PUT and INPUT functions can be used anywhere in SAS that a function can be called, for instance, from a WHERE clause.

A hash object lookup is faster than a format lookup because the hash object uses a function that subsets the keys to be searched when looking for a key. A hash object uses less memory than a format and multiple values can be used for the key and data parts of a hash record. With formats, one value is used as the key and one value is used as the data. One can manufacture a format key or data value by concatenating multiple values together, however the computational cost of the concatenation increases execution time.

The format must be kept in sync with the data set the format is constructed from, while a hash object is typically loaded directly from the data set with the most current information. Also, the amount of memory and time used to create the format can be significant.

MERGE WITH BY

The MERGE statement with BY, also known as match-merging, combines observations from two or more data sets when the values of the BY variables are the same. The comparison of BY variables is fast and, when they are sorted, a sequential pass is made through all of the input data sets. Match-merging requires the input data sets to be sorted or have an appropriate index. With SAS 9, sorting is faster when the machine contains several processors. Match-merging uses little memory because only the current observation from each input data set is loaded into memory at a time.

A hash object does not require an input data set to be sorted or indexed, however the hash object must fit into memory. The input to a hash object need not be sorted and the lookups against a hash object need not occur in sorted order either. Relieving the requirement of sorting may improve the performance of programs that sort the data one way for one step, then sort the data another way for a different step.

TEMPORARY ARRAYS

When using a DATA step _TEMPORARY_ array, the key is a numeric value and the lookup is very fast because the key directly accesses the data. An array can only hold one item of data per element. If multiple items of data need to be stored, multiple arrays are usually declared. When an array is declared, all of the memory that the array needs is allocated. If the key value is not a number or is more than one value, it may be possible to map the key to a unique number that is used to index into the array. Paul Dorfman and Gregg Snell present several ways to map values to array elements, essentially creating a hash table from a DATA step array [1].

Hash objects allocate memory as records are added, however looking up a key in a hash object involves more time than indexing a value in an array. A hash object uses more memory than a DATA step _TEMPORARY_ array. If the key value doesn't easily map to a number used to index into the array, using a hash object simplifies the programming.

PROS AND CONS

The table below summarizes the pros and cons of various lookup methods. The pros and cons should be used as guidelines as they may or may not apply to a particular application.

Method	Pros	Cons
SET with KEY=	Small memory footprint. Index can be reused by other mechanisms like WHERE clauses and SQL joins.	Can involve slow random I/O. Index creation and updates take time.
Formats and Informats	Lookups can occur from several SAS language statements with PUT and INPUT functions. Support mapping a range of values to one value. Fast, binary tree lookup.	Key and data are only one value. Requires more memory than a hash object. Longer lookup time than a hash lookup. Building and maintaining the format takes execution time. Format must fit into memory.

Method	Pros	Cons
MERGE with BY	Fast, sequential access of input data sets. Small memory footprint. Sorting input data sets is faster in SAS 9 if the machine has multiple processors. Hashing uses one processor.	Input data sets must be sorted or have an appropriate index.
TEMPORARY Arrays	Fastest lookup by indexing directly to location of data. Memory footprint smaller than hash object or format memory footprint.	Array must fit in memory. One block of memory is allocated when the array is declared. Array index or "key" must be a numeric value or be mapped to a numeric value.
Hash Objects	Fast lookup of records that contain a particular key. Key and data can be composed of multiple values of both character and numeric type. Dynamically grow to fit as many records that fit into memory. Memory footprint smaller than when using an equivalent format. Support find, add, remove, clear, replace, sum and equals operations. Ability to output sorted and unsorted data from the hash object.	Hash object records must fit into memory. _TEMPORARY_ array lookup faster than using a hash object. Hash object has larger memory footprint than an equivalent array.

CONCLUSIONS

This paper introduced the functionality of the hash object and described examples that use the hash object to perform match-merge operations and string summarization. Instead of focusing on the "under the covers" details of why hashing is fast and how it is implemented in the DATA step, this paper presented the functionality of the hash object, described several examples that demonstrate common applications of the hash object, and compared the hash object to existing key lookup methods in SAS to help provide guidelines as to when one lookup method should be used over another.

When a value from one data set must be looked up in another data set, often times a hash object can be used to improve the performance of the lookup. In particular, if one of the input data sets fits in memory, a hash object is the fastest choice available. The hash object is also a grow-able data structure where records can be found, added, replaced, and removed. A data structure of this type within the DATA step allows users to write programs that were either difficult or impossible to code within SAS.

ACKNOWLEDGMENTS

Bill Heffner and Al Kulik extended the DATA step to include the DECLARE statement and dot syntax.. Al Kulik implemented the hash object and hash iterator. The authors appreciate the help of Robert Ray, Kevin Hobbs, Jane Stroupe, and Al Kulik in reviewing this paper.

REFERENCES

[1] Dorfman, P. M., Snell, G. 2003. Hashing: Generations. *Proceedings of the Twenty-eighth SAS Users Group International Meeting*, paper 4-28. http://www2.sas.com/proceedings/sugi28/004-28.pdf (accessed May 30, 2006).

[2] SAS Institute Inc. 2006. *SAS 9.1.3 Base SAS Language Reference*. http://support.sas.com/onlinedoc/913/docMainpage.jsp (accessed May 30, 2006).

[3] SAS Institute Inc. 2006. *Hash Object Samples*.
http://support.sas.com/ctx/samples/index.jsp?sxf=s&mdType=datastep&c1=componentobjects&datastep=componentobjects (accessed May 30, 2006).

CONTACT INFORMATION

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

Jason Secosky
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-677-8000
Jason.Secosky@sas.com

Janice Bloom
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-677-8000
Janice.Bloom@sas.com