Paper BB16-2014

Introduction to SAS® Hash Objects

Chris Schacherer, Clinical Data Management Systems, LLC

ABSTRACT

The SAS hash object is an incredibly powerful technique for integrating data from two or more datasets based on a common key. The current work describes the basic methodology for defining, populating, and utilizing a hash object to perform lookups within the DATA step and provides examples of situations in which the performance of SAS programs is improved by their use. Common problems encountered when using hash objects are explained, and tools and techniques for optimizing hash objects within your SAS program are demonstrated.

INTRODUCTION

The most time-consuming SAS programming steps are usually those involving sorting or joining datasets. Schacherer (2011) described a situation in which creation of a healthcare encounters data mart took 36 hours to complete—mainly due to the number of intermediate datasets that were being created to bring together data from a number of source tables. Utilizing SAS hash objects to accomplish many of the data joins and "lookups", the program was rewritten to build the same data mart in less than 6 hours. This performance improvement is the main reason SAS programmers become interested in SAS hash objects; they provide a very fast way to look up data from one dataset based on a common key linking records in another dataset. As Secosky and Bloom (2007, p.1) describe them, a SAS 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.

Because the hash object entries are held in memory, finding the data value that corresponds to a given key happens much faster than it would if the records were read from disk. For example, say you had a dataset "claims" containing 100 million healthcare claims records and another dataset "providers" that contained information on 10,000 healthcare providers associated with those healthcare claims. There are a variety of methods for integrating data elements from "providers" to the associated record in "claims". One common method would be to perform a left-join from "claims" to "providers" to produce a dataset with all of the records from the "claims" dataset and information from the "providers" dataset where a match is found on "provider_id".

Using PROC SQL, the new dataset "claims_providers" could be created as follows:

```
PROC SQL;
 CREATE TABLE claims_providers AS
       SELECT a.*,b.provider_last_name,
              b.provider_first_name,b.npi
         FROM eiw.claims a LEFT JOIN eiw.providers b
           ON a.provider_id = b.provider_id;
QUIT;
```

Keeping in mind that when SAS executes the preceding code, 100 million records are being read from the "claims" dataset and the 10,000 records in "providers" are being evaluated for a match to each "claims" record's value of "provider_id", on desktop installation of SAS, the query took **9 minutes and 25 seconds** to complete. Using a DATA step approach that utilizes a SAS hash object (also known as an associative array), the creation of the same dataset took **3 minutes and 8 seconds**—a 66% reduction in runtime.

The performance gains that can be achieved using hash objects become much more dramatic as the number of queries in your program (and their complexity) increase. In the following example, information associated with "providers" and "clinics" is added to healthcare "claims" records using two left joins—to retain all records in the claims file and add "provider" and "clinic" information where applicable.

```
PROC SQL;
 CREATE TABLE claims_providers AS
       SELECT a.*,b.clinic_name,b.clinic_city,b.clinic_state
         FROM (SELECT t1.*,t2.provider_last_name,t2.provider_first_name,t2.npi
                 FROM eiw.claims t1 LEFT JOIN eiw.providers t2
```

```
                     ON t1.provider_id = t2.provider_id) a
      LEFT JOIN eiw.clinics b
              ON a.clinic_id = b.clinic_id ;
   QUIT;
```

Working with the same "claims" and "providers" datasets as in the previous example (and adding the "clinics" dataset to the query), the PROC SQL step involving two left joins takes over **21 minutes and 8 seconds** to complete, whereas the equivalent DATA step using hash objects to perform the lookups takes just **4 minutes and 17 seconds**.

In both of these examples, the performance improvement using the DATA step approach with hash objects is due to the fact that the records being looked up (providers in the first example and both providers and clinics in the second example) are found via direct memory addressing instead of serial search of the physical file—what Dorfman (2000a,b) refers to as the difference between "comparing" and "searching". To provide an overly-simplistic analogy, the difference is comparable to search for "John Smith" by walking up and down every street in town, knocking on every door, and asking for John Smith (PROC SQL) versus looking up John Smith's address in the phone book and going directly to the listed address (DATA step with hash objects).

Of course there are a number of ways to optimize PROC SQL steps (see, Lafler, 2004a, 2004b, 2005), and there are other methods, such as user-defined formats, that can be used to rapidly lookup keyed values from other datasets (see for example, Schacherer, 2011 and Schacherer & Westra, 2010). Comparisons of the hash object to these different approaches have been documented elsewhere, but for situations in which the number of lookups being performed, the complexity of the keyed values, and the number and types of elements returned from the lookup are of any consequence, it is widely accepted that the performance of SAS hash objects is hard to beat[1]

A BRIEF HISTORY OF THE SAS HASH OBJECT

The concept of in-memory data elements is not new to programming in general, or SAS programming in particular. The underpinnings of the modern SAS hash object can be traced to the ingenious early work by Dorfman (2000b, 2001; see also Dorfman & Snell, 2002, 2003) who pointed out that arrays are "ideal for implementing just about any searching algorithm…" (Dorfman, 2001, p. 1). He demonstrated that in-memory table look-ups (or, direct address searches) could be programmed by hand in order to do the types of in-memory searches performed by the modern hash object. He described three such methods—key-indexed search, array as a bitmap, and hashing—that all have their basis in assigning key values to memory addresses (direct addressing) so that when a program later searches for the value (say, to match it to a corresponding value in another dataset) it can go directly to the specified address and find the value rather than compare the value of the key value in the first dataset to all of the values in the corresponding variable in another dataset.

As Dorfman explains, the purest form of direct addressing is key-indexing—in which a temporary array is created with one position allocated for each possible key value (say, all of the primary key values for a reference dataset). The subsetting of another dataset—to include only those records with a key match to the reference dataset—can be accomplished by evaluating whether the key value for a given record in the dataset being subset exists in the temporary array. Note, that in using this approach, records from the two datasets are not directly compared, but rather the "address/location" of the key value in the evaluated dataset is looked up in the temporary array, determined to be either null or not null, and in so doing, the answer to whether there is a matching record in the reference dataset is determined.

Adapting the example provided by Dorfman (2001), suppose you have a large dataset of healthcare claims "claims" and you want to subset it to include only those claims associated with providers from XYZ Health System "xyz_providers". Using a key-indexing approach, you might create that subset as follows:

```
    DATA xyz_doc_claims;
     ARRAY xyz_docs (1:10000) _TEMPORARY_;

     /* Load array "xyz_docs" with the provider_ids
        from dataset "xyz_providers" */
     DO UNTIL (eof1);
        SET xyz_providers end=eof1;
            xyz_docs(provider_id) = provider_id;
     END;
```

[1] But see Dorfman (1999) and Dorfman & Vyverman (2006) for descriptions of specific techniques and situations in which the hash approach can be outperformed by other memory-resident techniques.

```
      /* For each record in the "claims" dataset evaluate
         the "xyz_docs" array for a match to the current
         value of "provider_id" from "claims". */
      DO UNTIL (eof2);
         SET claims end=eof2;
             doc_search = xyz_docs(provider_id);
             IF doc_search > . THEN OUTPUT;
         END;
    RUN;
```

In the preceding code, the temporary array (xyz_docs) was created and filled with the primary key from the "xyz_providers" dataset. Next the "claims" dataset is evaluated and the temporary array is used to find whether the "provider_id" listed on the claim is in the array "xyz_docs". If it is, the record is output to the new dataset "xyz_doc_claims".

Note that we did not search "xyz_docs" for an applicable match; we went to the location in the array where the associated provider_id would be found and it either was or was not indicated to be a match.

As Dorfman (2001) put it more eloquently,

> *From the nature of the algorithm, it is clear that no lookup method is simpler and/or can run faster than key-indexing: It completes any search, hit or miss, without comparing any keys, via a single array reference. It also possesses the fundamental property: Its speed does not depend on the number of keys "inserted" into the table, i.e. any single act of key-indexed search takes precisely the same time." (p 1.)*

However, there are limitations to the key-indexing approach—namely, the amount of memory needed to hold all possible values in the key-indexed table[2]. Therefore, Dorfman devised another technique to expand the number of addressable keys that could be assigned in a given amount of memory—a technique called bitmapping, in which keyed values are replaced by smaller, single-bit (0/1) indicators in the temporary array. Though this technique successfully expands the number of key values that can be directly addressed, it is still limited in the range of values that it can accommodate because all possible integer values in the range must be represented. For example, even if only three values in a range of 1,000,000 possible values will be represented in the key-indexed table (say , "1", "789,917", and "999,999") all 1 million slots must be created and valued.

To overcome this issue, Dorfman (2001) introduced the concept of hashing using arrays. In a hashing approach, instead of a unique location being created for each possible key and loading a value only to the specific location created for that value (a pure direct-addressing approach), a smaller number of locations are defined in the array, an algorithm is developed to assign key values to a location in the array (with the possibility that two or more distinct keys will be mapped to the same location), and when two values are mapped to the same location, appropriate logic is executed to deal with the "collision"—building links to all key values that map to a given location. These "collision resolution policies" invariably result in some comparisons having to be made as part of the search—instead of automatically finding the key via its address. For example, if three key values (e.g., provider IDs "537", "8945", and "3215") all map to the same address/location, it is now not enough to know what that location is; once there, a comparison to each of these three values still needs to be made in order to determine if a match exists. However, this is still more efficient that comparing the current key value to the associated key variable in each record in another dataset. Therefore, the hash approach strikes a balance between the potentially higher performance of the key-indexing and bitmapping approaches and a more efficient use of available memory.

When Dorfman first shared these techniques with the SAS community, it seemed obvious that the magnitude of performance gains these methods could provide would initiate an immediate shift in how SAS programmers approached problems involving joining records from different datasets. However, as judged by the relatively small number of SUGI, SGF, and regional user group papers on the topic, these revolutionary techniques remained relatively obscure—except for a small number of hash enthusiasts. It could be that interest was lackluster because, as a percentage of all SAS programs written, those programs in which a substantive performance difference can be achieved using hashing approaches may be relatively small[3]. However, at least part of the problem was probably the seeming complexity of the hand-coded programming solutions; they challenged people to think about array

[2] Also, as is discussed later, this approach does not yield the same breadth of functionality users have come to expect from the SAS hash object.

[3] Although, the author suspects that this percentage may also be somewhat under-estimated because programmers are not learning the techniques to utilize hash approaches.

processing in novel ways that can be a bit intimidating.

SAS helped remove the latter barrier to adoption of hash techniques by introducing the hash object in version 9.0 (beta) and version 9.1 (production). The SAS hash object is meant to "enable you to quickly and efficiently store, search, and retrieve data based on lookup keys" (SAS, Inc., 2004, p. 36). Conceptually, the hash object provides programmers the means to easily define and utilize a hash table within the DATA step. Through the **DATA step Component Object Interface** (SAS, Inc., 2011a) certain data elements (of which the hash object is one class) can be utilized within the DATA step. According to SAS, Inc. (2011a) these data elements are made up of **attributes** (which represent properties of the element), **methods** (the predefined operations the object can perform), and **operators** (special, more complex operations that usually use more complex logic operating against the data contained in the element). Dorfman & Vyverman (2006) appropriately describe the hash object as something of a black box and liken it to a SAS PROCEDURE due to its ability to perform specific operations. The difference between object operations (or, methods) and PROCs, however, is that the object methods can be called and executed within the DATA step. These "methods" provide the main means of interacting with data inside the black box, attributes are examined to learn about the contents of the black box, and the **DATA step object dot notation** is the syntax one uses to access the attributes, methods, and operators.

BASIC HASH OBJECT SYNTAX

So, how does one create and use a hash object? Consider the previous example in which PROC SQL was used to join 100 million rows of claims data to 10,000 records containing provider details. The DATA step that creates the same dataset using a hash object might look something like the following:

```
  DATA work.claims_providers;
  LENGTH provider_lname provider_fname $25 npi $10;

❶  DECLARE HASH provider(dataset:'eiw.providers');
  ❷ provider.DEFINEKEY ('provider_id');
  ❸ provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
  ❹ provider.DEFINEDONE();

❺DO UNTIL (eof_claims);
   SET eiw.claims END = eof_claims;
       rc = provider.FIND();
  ❻    IF rc ne 0 THEN DO;
          provider_lname = 'Provider Not Found';
          provider_fname = 'Provider Not Found';
          npi            = 'xxxxxxxxxx';
       END;
❼OUTPUT;
  END;
  STOP;
  RUN;
```

❶ The hash object "provider" is identified to SAS as an object of type "hash" using the DECLARE statement[4]. In this example, the dataset attribute of the hash object "provider" is specified as "eiw.providers". When the hash object is instantiated at runtime, the hash object will load records from that dataset into the in-memory data element "provider".

❷ Using the component object interface dot notation, the DEFINEKEY method is called from the provider object. Again, methods can be thought of as the operations that an object is capable of executing; they are a way to directly communicate with the hash object. In this case what we want to do with the "provider" object is to define the key value to use in performing lookups from the in-memory table. Recall that Secosky and Bloom (2007) described the hash object as a table with a key part and a data part. In this case the data about our providers are going to be looked up from the in-memory table by referencing the "provider_id"—the key associated with each hash table entry.

[4] In addition to the keyword "hash", you can also specify the declaration of an "associativearray"—the original name for this component object type.

❸ The variables "provider_lname", "provider_fname", and "npi" represent the data that we want to return from the hash object entry when a match for the associated provider_id is found; the data portion of the object is specified with the DEFINEDATA method.

❹ Finally, when the hash object definition has been completed, a call to the DEFINEDONE method is made to complete initialization of the object.

❺ At runtime, after the hash object is defined and initialized, the program will loop through the source dataset "eiw.claims" until the end of the dataset is reached.

❻ During the processing of each "claims" record, a call is made to the FIND method of the "provider" hash. By default, the FIND method takes the current value of "provider_id", searches the "provider_id" entries in the "provider" hash for a match, and (if a match is found) assigns the hash entry's values for "provider_lname", "provider_fname", and "npi" to the current "claims" record being processed. In the case where a match is found, the value "0" is assigned to the local variable "rc", otherwise, "rc" is assigned a non-zero value. In the present example, the value assigned to this return code (rc) is used to determine whether additional manipulations to the provider information are necessary. In this case, if a match to the current value of "provider_id" is not found in the hash table, the value "Provider Not Found" is assigned to both "provider_lname" and "provider_fname" and an equivalent indicator is assigned to "npi".

❼ At the end of each loop through the "claims" dataset, the record processed in that loop is output to "claims_providers" dataset.

BEHIND THE SCENES IN THE PDV

In the preceding example, one seemingly innocuous line is the "LENGTH" statement that immediately follows the DATA statement.

```
DATA work.claims_providers;
LENGTH provider_lname provider_fname $25 npi $10;
```

The reason this is done is so that the values returned from the hash object will have an assigned location in the program data vector. As described by Whitlock (2006, p. 5):

> *The program data vector, abbreviated as PDV, is a logical concept to help you think about the manipulation of variables in the DATA step. Prior to version 6 it was also a physical concept referring to the specific area in memory where DATA step variables were stored. In version 6 the memory structure was reorganized to speed up the system time spent outside the ILDS* [implied loop of the data step]*, but the PDV concept continues to be a good way for the programmer to think about what is important to him in variable manipulation. The PDV can be thought of as one long list of storage areas for all the named variables of the DATA step.*

As illustrated below, if one were to simply copy one dataset to another and compute one new variable, the values represented in the PDV during execution might look something like the following during different stages in the implied loop of the data step.

```
DATA work.test;
  SET eiw.claims;
  amount_due = bill_amount - copay;
RUN;
```

PDV Contents – Record = 1 (<u>before</u> "amount_due" computed)	
_N_	1
ERROR	0
claim_id	125468785
provider_id	2583
clinic_id	1255
bill_amount	125.25
copay	25
amount_due	.

PDV Contents – Record = 1 (<u>after</u> "amount_due" computed)	
_N_	1
ERROR	0
claim_id	125468785
provider_id	2583
clinic_id	1255
bill_amount	125.25
copay	25
amount_due	100.25

PDV Contents – Record = 2 (<u>before</u> "amount_due" computed)	
_N_	2
ERROR	0
claim_id	62238547
provider_id	589
clinic_id	37
bill_amount	11327
copay	0
amount_due	.

Note that the internal variables (_N_ and _ERROR_) are also represented in the PDV—though they cannot be output to the dataset without being assigned as the values of a local variable.

The point of discussing the PDV is to point out that the variables defined as the KEY and DATA components of the hash object are not automatically recognized by the DATA step as are those in a dataset declared in a SET statement. The role played by the LENGTH statement in the previous hash object example, therefore, is to correctly define within the PDV the variables that will be returned by the hash object FIND call. Put another way, if the LENGTH statement was commented out of the previous example and there were no other references to these variables in the DATA step (other than their inclusion in the hash object declaration) no locations would be defined in the PDV for the values returned from the FIND method call. In this case, the following error would occur:

```
DATA work.claims_providers;
*LENGTH provider_lname provider_fname $25 npi $10;

  DECLARE HASH provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
 OUTPUT;
STOP;
RUN;
```

```
ERROR: Undeclared data symbol provider_lname for hash object at line 1341 column 5.
ERROR: DATA STEP Component Object failure.  Aborted during the EXECUTION phase.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 1 observations read from the data set EIW.CLAIMS.
WARNING: The data set WORK.CLAIMS_PROVIDERS may be incomplete.  When this step was
stopped there were 0 observations and 14 variables.
```

The reason this happens is that the hash object is a runtime component—being created only as the DATA step is executed—and the variables declared within it are not recognized by the compiler as the executable version of the program logic is prepared (Dorfman & Vyverman, 2009, Eberhardt, 2011). In other words, the compiler cannot "see" inside the hash object to gather information about the variables specified as the KEY and DATA components. Therefore, there is no "slot" in the PDV for the variable values returned from the hash object FIND call; they cannot make their way into the PDV nor, therefore, out onto the dataset record.

Another indicator of the compiler's failure to access the dataset description provided in the hash object is that the following example (with the LENGTH statement enabled) results in "variable is uninitialized" NOTES being written to the log. After declaring the variables in the LENGTH statement, they are not found anywhere in the executable code (Slaughter & Delwiche, 1997) and the following notes are generated to the log:

```
DATA work.claims_providers;
LENGTH provider_lname provider_fname $25 npi $10 ;

  DECLARE hash provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
     OUTPUT;
 STOP;
```

```
RUN;


NOTE: Variable provider_lname is uninitialized.
NOTE: Variable provider_fname is uninitialized.
NOTE: Variable npi is uninitialized.
```

Though the variables are still successfully created on the new dataset and assigned the values returned from the FIND method call, some programmers do not like the superfluous notes written to the log (especially cumbersome where large numbers of variables are assigned in the DEFINEDATA method call)[5]. For this reason, one may want to use CALL MISSING (which sets the values of listed variables to missing) to suppress these messages.

```
DATA work.claims_providers;
LENGTH provider_lname provider_fname $25 npi $10 ;

  DECLARE hash provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

CALL MISSING(provider_lname, provider_fname, npi);

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
 OUTPUT;
END;
STOP;
RUN;
```

However, using CALL MISSING to turn off notes that are not really causing any problems and manually declaring variables using a LENGTH statement (when these variables do, in fact, exist in a SAS dataset already) adds unnecessarily to your code. In most cases, you can get the information about the hash object variables in a manner that is much more efficient and reliable[6]. As described by Dorfman, Shajenko, & Vyverman (2008), Dorfman & Eberhardt (2011), and Eberhardt (2011), a more efficient way to identify variables to the PDV (in a way that is guaranteed to produce reliable parameter type matching) is to issue a conditional "SET" statement that specifies the dataset that sources the hash object. In the following example, the condition "IF _N_ = 0" is used to specify when the SET statement should be executed and the name of the dataset sourcing the hash object is specified. Because _N_ (the internal indicator of the number of the implied internal loop step currently executing) will never be equal to zero, the SET statement will never be executed. However, in compiling the DATA step, SAS recognizes the "possibility" that the dataset will be accessed and so the variable descriptors are read and allocated a spot within the PDV. In addition to the code executing more cleanly with respect to unwanted notes cluttering the log, this approach has the added benefit of your code being more robust to changes to the source data. For example, if the length of "provider_lname" was changed in the source data, you would need to manually recode a LENGTH statement, but the approach in the current example would accommodate that change automatically.

```
DATA work.claims_providers;
IF _N_ = 0 THEN SET eiw.providers;

  DECLARE hash provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();
```

[5] Also, as suggested by Schacherer (2013), controlling the output of these does demonstrate a certain level of mastery over understanding your data and the programming logic being applied to them.

[6] Although there are certainly situations in which a manual approach may still be warranted.

```
  DO UNTIL (eof_claims);
   SET eiw.claims END = eof_claims;
       rc = provider.FIND();
   OUTPUT;
  END;
  STOP;
  RUN;
```

DATA STEP MECHANICS

In addition to the unique issues that arise with respect to the PDV when including a hash object in the DATA step, the mechanics of the DATA step also warrant further discussion. Comparing the previous example (below, left) with a commonly-used alternative (below, right) you will notice the key difference is that the example on the right makes declaration of the hash object conditional on the implicit loop of the DATA step being on the first iteration (i.e., when _N_ = 1).

```
DATA work.claims_providers3b;
IF _N_ = 0 THEN SET eiw.providers;

  DECLARE hash
    provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname',
                        'provider_fname',
                        'npi');
    provider.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
 OUTPUT;
END;
STOP;
RUN;
```

```
DATA work.claims_providers3b;
IF _N_ = 0 THEN SET eiw.providers;

IF _N_ = 1 THEN DO;
  DECLARE hash
    provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname',
                        'provider_fname',
                        'npi');
    provider.DEFINEDONE();
END;

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc1 = provider.FIND();
 OUTPUT;
END;
RUN;
```

The reason this conditional logic is applied is that when the DATA step reaches the end of the programming logic, focus is returned again to the top of the DATA step. If this condition was not in place, the hash object would be created a second time—although the "claims" dataset would not be processed again because the condition that controls that loop has already been satisfied. In the following example (where the "_N_ = 1" condition is commented out) the LOG displays evidence of this behavior; "eiw.providers" is read once when _N_ = 1 and once when _N_ = 2. Including the "STOP" at the end of the program logic (as in the preceding example) stops the data step at the end of the _N_ = 1 loop so there is no need for the "_N_ = 1" condition.

```
DATA work.claims_providers;
IF _N_ = 0 THEN SET eiw.providers;

/*IF _N_ = 1 THEN DO;*/
  DECLARE hash provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();
/*END;*/

DO UNTIL (eof_claims);
```

```
     SET work.claims END = eof_claims;
         rc = provider.FIND();
     OUTPUT;
    END;
    RUN;
```

```
NOTE: There were 10000 observations read from the data set EIW.PROVIDERS.  (where _N_ = 1)
NOTE: There were 10000 observations read from the data set EIW.PROVIDERS.  (where _N_ = 2)
NOTE: There were 1000000 observations read from the data set WORK.CLAIMS.
```

It should be noted that all three examples correctly create the same "claims.providers" dataset, but strictly speaking, the example in which only one iteration through the DATA step logic is executed is more efficient—as the "STOP" at the end of the DATA step stops SAS from returning to the top of the DATA step again.

With a solid understanding of how the hash object is integrated with the PDV and DATA step, the following examples expand on the basic hash object syntax to demonstrate some alternative ways that the hash object can be built and used.

ADDITIONAL HASH MECHANICS

MANUALLY LOADING THE HASH OBJECT.

Another variant of the basic hash object syntax that you might run across is a three step approach that separates the declaration of the hash object from its population with data (a "build, fill, use" approach as opposed to a "build and use" approach). In the following example, after declaring the hash object without specifying the source dataset's name, a DO loop is executed to read the records from "eiw.providers" and add them to the hash table using the ADD method of the "provider" hash object. As each record is read, the ADD method inserts the values of the KEY and DATA variables into the new hash object entry.[7]

```
DATA work.claims_providers;

  DECLARE hash provider();
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_providers);
 SET eiw.providers END = eof_providers;
 rc0 = provider.add();
END;

DO UNTIL (eof_claims);
 SET work.claims END = eof_claims;
     rc1 = provider.FIND();
 OUTPUT;
END;
RUN;
```

Note that in this approach, even though there is neither an "IF _N_ = 1" nor a "STOP", the log reveals that the records from "eiw.providers" are read only once—because when the hash object is instantiated again during the _N_ = 2 loop, the dataset is not identified.

```
NOTE: There were 10000 observations read from the data set EIW.PROVIDERS.
NOTE: There were 1000000 observations read from the data set WORK.CLAIMS.
```

[7] It should be noted that records with duplicate values of "provider_id" will not be loaded into the hash object after the first, unique occurrence of that particular "provider_id". So, it is possible for a 10,000 record dataset to load only 9589 records, for example, if the dataset contains duplicate values of "provider_id". This is true regardless of the method used to fill the hash object and has to do with the default behavior of the hash object. This topic is covered in more depth later in the paper.

Because the loading of data into the hash object is outside of the hash object definition, you now have more control over the loading of records into the hash object. Suppose you only wanted to load certain records from "providers" into the hash object. As shown in the following example, you could control the records loaded into the hash object by using a subsetting WHERE clause—in this case to load only providers from Texas and Wisconsin.

```
DATA work.claims_providers;

  DECLARE hash provider();
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_providers);
 SET eiw.providers END = eof_providers;
 WHERE state IN ('TX','WI');
 rc0 = provider.add();
END;

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
     IF rc ne 0 THEN DO;
       provider_lname = 'Provider Not Found';
       provider_fname = 'Provider Not Found';
       npi            = 'xxxxxxxxxx';
     END;
 OUTPUT;
END;
RUN;
```

USING DATASET OPTIONS IN HASH OBJECT DECLARATION.

Beginning in SAS 9.2 the DATASET argument of the hash declaration can also include dataset options. In the following example, a WHERE clause is used to subset the "providers" records that are loaded to the "provider" hash object—having the same effect as using a WHERE clause to conditionally load the data in a separate DO LOOP and eliminating the need for additional lines of code.

```
DATA work.claims_providers;
IF _N_ 0 THEN SET eiw.providers;

  DECLARE hash provider(DATASET:'eiw.providers (WHERE=(state in ("TX","WI")))' );
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
     IF rc ne 0 THEN DO;
       provider_lname = 'Provider Not Found';
       provider_fname = 'Provider Not Found';
       npi            = 'xxxxxxxxxx';
     END;
 OUTPUT;
END;
```

```
STOP;
RUN;
```

EXPLICITLY-KEYED FIND METHOD CALLS

The arguments passed to the hash object method calls can expand the functionality of the hash object significantly. In the previous examples the resulting "claims_providers" dataset contained "provider_lname", "provider_fname", and "npi" values as a result of the those variable names being assigned in the PDV and the subsequent call to the FIND method returning values (or not) to those slots in the PDV based on the current "claims" record's value of "provider_id". But what if the variable containing the values of "provider_id" in the "claims" dataset had a different name (say, "primary_provider")? Moreover, what if you wanted to use the same hash object to lookup values for more than one variable?

In the following example, suppose that the "claims" dataset includes both a "provider_id" and a "referring_provider_id" and you want to use the same hash object to lookup the name and NPI of both the provider and the referring provider. As shown in the following example, this can be achieved by making two separate calls to the FIND method of the "provider" hash object (Warner-Freeman, 2007). The first call takes the same form as in the previous examples; FIND is called with no arguments—implicitly searching the hash object for a match based on the value of the current record's "provider_id" value. The second call includes an explicit KEY argument to indicate that the values of the "provider_id" key in the hash object should be searched for a match to the current record's value of "referring_provider_id". If a match is found, the values of "provider_lname", "provider_fname", and "npi" corresponding to the value of "referring_provider_id" are returned and subsequently assigned to the local variables "ref_provider_lname", "ref_provider_fname", and "ref_npi".

```
DATA work.claims_providers (DROP=provider_lname provider_fname npi rc1 rc2);
IF _N_ = 0 THEN SET eiw.providers;

  DECLARE hash provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_claims);
 SET work.claims END = eof_claims;

     rc1 = provider.FIND();
    IF rc1 = 0 THEN DO;
       primary_provider_lname = provider_lname;
       primary_provider_fname = provider_fname;
       primary_provider_npi = npi;
      END;
    ELSE DO;
       primary_provider_lname = 'Provider Not Found';
       primary_provider_fname = 'Provider Not Found';
       primary_provider_npi   = 'xxxxxxxxxx';
      END;
    rc2 = provider.FIND(KEY:referring_provider_id);
    IF rc2 = 0 THEN DO;
       ref_provider_lname = provider_lname;
       ref_provider_fname = provider_fname;
       ref_npi = npi;
      END;
     ELSE DO;
       ref_provider_lname = 'Provider Not Found';
       ref_provider_fname = 'Provider Not Found';
       ref_npi = 'xxxxxxxxxx';
```

```
         END;
    OUTPUT;
   END;
   STOP;
   RUN;
```

Especially when using the hash object for multiple, differently-keyed lookups, it is imperative that you pay close attention to how the results of the FIND call are handled. Note that in the preceding example, the return code (rc1) associated with the implicitly-keyed FIND call was used to conditionally assign the values of three new variables—"primary_provider_lname", "primary_provider_fname", and "primary_npi"—explicitly rather than keeping the results in the local variables "provider_lname", "provider_fname", and "npi" returned by the FIND call. The reason this approach was taken is that both searches ("provider.FIND()" and "provider.FIND(KEY: referring_provider_id") are going to return the results of their respective searches to the same locations in the PDV ("provider_lname", "provider_fname", and "npi").

If you were to use the logic provided in the prior examples to assign values to the local variables "provider_lname", "provider_fname", and "npi" based on the implicitly-keyed FIND call "FIND()", when the explicitly-keyed FIND call "FIND(KEY: referring_provider_id)" is subsequently made, the values in the "provider_lname", "provider_fname", and "npi" locations in the PDV would simply be over-written with the results of the explicitly-keyed FIND call every time the referring provider search is successful.

```
   …
        rc1 = provider.FIND();
        IF rc1 ne 0 THEN DO;
         provider_lname = 'Provider Not Found';
         provider_fname = 'Provider Not Found';
         npi = 'xxxxxxxxxx';
        END;

       rc2 = provider.FIND(KEY:referring_provider_id);
       IF rc2 = 0 THEN DO;
          ref_provider_lname = provider_lname;
          ref_provider_fname = provider_fname;
          ref_npi = npi;
             END;
           ELSE DO;
          ref_provider_lname = 'Provider Not Found';
          ref_provider_fname = 'Provider Not Found';
          ref_npi = 'xxxxxxxxxxx';
         END;
```

In other words, using the preceding code you could have (for a given record in "claims") a result of "John Smith, NPI: 1234567890" returned in response to the first FIND call "FIND()" and a result of "Mary Jones, NPI: 1807358473" returned as a result of the second (explicitly-keyed) FIND call "FIND(KEY:referring_provider_id)". However, the record output to the "claims_providers" dataset will look something like the following because the values in the provider variable slots would be overwritten by the results of the referring provider search.

provider_lname	provider_fname	npi	ref_provider_lname	ref_provider_fname	ref_provider_npi
Jones	Mary	1807358473	Jones	Mary	1807358473

In the following example, reversing the order of these two FIND calls (and their associated conditional logic) fixes the problem because now every time there is a successful search using FIND(), the values of "provider_lname", "provider_fname", and "npi" are overwritten in the PDV by the results of that search, and if the search is not successful the values are explicitly assigned the value "Unknown".

```
   …
       rc2 = provider.FIND(KEY:referring_provider_id);
       IF rc2 = 0 THEN DO;
          ref_provider_lname = provider_lname;
          ref_provider_fname = provider_fname;
          ref_npi = npi;
         END;
        ELSE DO;
          ref_provider_lname = 'Provider Not Found';
          ref_provider_fname = 'Provider Not Found';
          ref_npi = 'xxxxxxxxxx';
         END;
        rc1 = provider.FIND();
        IF rc1 ne 0 THEN DO;
         provider_lname = 'Provider Not Found';
         provider_fname = ''Provider Not Found';
         npi = 'xxxxxxxxxx';
        END;
```

provider_lname	provider_fname	npi	ref_provider_lname	ref_provider_fname	ref_provider_npi
Smith	John	1234567890	Jones	Mary	1807358473

This example makes clear, once again, the importance of understanding the inner workings of the DATA step and the PDV in effectively using hash objects.

COMPOSITE KEYS

All of the examples so far have involved hash objects with a single value "provider_id" as the key. However hash object entries can just as likely be identified by composite keys. In the following example, a large dataset of billing claims for professional clinical services is processed to model physician revenue. Each claim line-item contains a payer ID (payer_id), amount billed (bill_amt), and the year when services were provided (svc_year). Contracts with the different payers change from year to year and as a result of those changes the "modifier" associated with different payers (i.e., the discount that different payers receive) may also change. As a result, in 2007 a payer may have agreed to pay 82% of the standard billed rate, in 2008, 81.5% and so on. Therefore, in order to get the correct modifier on each claim record, the payer and the service year must be used to find the appropriate modifier.

The difference between this hash object and those in the previous examples is fairly minor. The hash object "modifiers" is defined based on the dataset "eiw.payment_modifiers", but in this example the DEFINEKEY call indicates that the unique combination of two values is required to form the key. The DEFINEDATA call specifies that only a single value will be returned as a result of a successful search of the hash entries. When the FIND call is executed, "modifiers" will be searched for an entry that matches the current "claims" record values of "payer_id" and "svc_year". If a match is found, the "modifier" value contained in the hash object entry will be returned to the modifier location in the PDV and will be written to the "pro_claims" record.

```
DATA work.pro_claims;
IF _N_ = 0 THEN SET eiw.payment_modifiers;

  DECLARE hash modifiers(dataset:'eiw.payment_modifiers');
    modifiers.DEFINEKEY ('payer_id','svc_year');
    modifiers.DEFINEDATA('modifier');
    modifiers.DEFINEDONE();

DO UNTIL (eof_claims);
 SET work.claims END = eof_claims;
     rc = modifiers.FIND();
     IF rc NE 0 THEN modifier = 1;
 OUTPUT;
END;
STOP;
RUN;
```

Note, here, that it is important that you have a clear understanding of what you should do in the case where a match is not found. Do you assign "1" as the modifier? Or "0"? Missing? It depends on what you plan to do with these data.

MULTIPLE HASH OBJECTS IN A DATA STEP

Another way in which you might expand your use of hash objects is to use multiple hash objects in a single DATA step. For example, Schacherer (2011) demonstrated the creation of a healthcare data mart in which a series of summary statistics where calculated on a large claims dataset and then each of those summary statistics was added to a fact table record by creating several hash objects and then processing the fact table with a FIND call to each of the hash objects containing the summary statistics:

```
DATA etl.billing_fact;

  DECLARE HASH payment(dataset:'etl.payments');
    payment.DEFINEKEY ('claimid');
    payment.DEFINEDATA('payment_amount');
    payment.DEFINEDONE();

  DECLARE HASH voided(dataset:'etl.voided_charges');
    voided.DEFINEKEY ('claimid');
    voided.DEFINEDATA('voided_charges');
    voided.DEFINEDONE();

  DECLARE HASH bad(dataset:'etl.bad_debt');
    bad.DEFINEKEY ('claimid');
    bad.DEFINEDATA('bad_debt');
    bad.DEFINEDONE();

  DECLARE HASH co_pay(dataset:'etl.copays');
    co_pay.DEFINEKEY ('claimid');
    co_pay.DEFINEDATA('copay');
    co_pay.DEFINEDONE();

  DO UNTIL (eof_fact1);
   SET work.claims END = eof_fact1;
       rc1 = payment.FIND();
      IF rc1 ne 0 THEN payment_amount = 0;
       rc2 = voided.FIND();
      IF rc2 ne 0 THEN voided_charges = 0;
      rc3 = bad.FIND();
      IF rc3 ne 0 THEN bad_debt = 0;
      rc4 = co_pay.FIND();
      IF rc4 ne 0 THEN copay = 0;
   OUTPUT;
   END;
   STOP;
RUN;
```

```
NOTE: There were 98983738 observations read from the data set ETL.PAYMENTS.
NOTE: There were 922158 observations read from the data set ETL.VOIDED_CHARGES.
NOTE: There were 1112738 observations read from the data set ETL.BAD_DEBT.
NOTE: There were 58763258 observations read from the data set ETL.COPAYS.
NOTE: There were 100000000 observations read from the data set WORK.CLAIMS.
NOTE: The data set ETL.BILLING_FACT has 100000000 observations and 16 variables.
```

HASH ISSUES

Hopefully, the previous examples have given you a feel for some of the ways in which you can use hash objects to lookup values from one dataset for integration into another dataset. Even with a good understanding of the basic hash object techniques, however, when you are first getting started with these techniques you are likely to encounter one or more of the following issues.

MEMORY-RESIDENT ALSO MEANS MEMORY-LIMITED

The most important limitation to keep in mind when utilizing the hash object is that because the hash object is memory-resident, its size is limited to the amount of memory available to SAS. In the following example a hash object containing the professional fees data associated with a set of claims is built with the intention of looking up the value of "professional_fees" corresponding to each value of "claimid" in the "claims" dataset.

```
DATA work.pro_claims;
IF _N_ = 0 THEN SET eiw.professional_fees;

  DECLARE hash profee(dataset:'eiw.professional_fees');
    profee.DEFINEKEY ('claimid');
    profee.DEFINEDATA('pro_fees');
    profee.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = profee.FIND();
     IF rc NE 0 THEN pro_fees = 0;
 OUTPUT;
END;
STOP;
RUN;
```

However, as the subsequent error describes, the data step could not be completed because of a memory failure. SAS attempted to load the hash object with data from "professional_fees" but the available memory was exhausted before all of the 20 million records in "professional_fees" could be loaded. When SAS ran out of available memory, it stopped processing the DATA step.

```
ERROR: Hash object added 15728624 items when memory failure occurred.
FATAL: Insufficient memory to execute DATA step program. Aborted during the EXECUTION phase.
ERROR: The SAS System stopped processing this step because of insufficient memory.
NOTE: There were 1 observations read from the data set EIW.CLAIMS.
WARNING: The data set WORK.PRO_CLAIMS may be incomplete.  When this step was stopped there
were 0 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           34.34 seconds
      cpu time            25.03 seconds
```

To understand how much memory is being used by the DATA step, you can add the FULLSTIMER option to your SAS session as in the following example.

```
OPTION FULLSTIMER;
DATA work.pro_claims;
IF _N_ = 0 THEN SET eiw.professional_fees;
  DECLARE hash profee(dataset:'eiw.professional_fees', hashexp: 20);
...
<REMAINING DATA STEP CODE>
...
RUN;
OPTION NOFULLSTIMER;
```

The additional performance metrics provided by the FULLSTIMER option show that 492 megabytes of memory were used by the DATA step at the time it stopped.

```
ERROR: Hash object added 15728624 items when memory failure occurred.
FATAL: Insufficient memory to execute DATA step program. Aborted during the EXECUTION phase.
ERROR: The SAS System stopped processing this step because of insufficient memory.
NOTE: There were 1 observations read from the data set WORK.CLAIMS.
WARNING: The data set WORK.PRO_CLAIMS may be incomplete.  When this step was stopped there
were 0
         observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           29.54 seconds
      user cpu time       24.25 seconds
      system cpu time     1.65 seconds
      memory              492210.91k
      OS Memory           498196.00k
      Timestamp           10/19/2013 04:10:07 PM
```

A check of the MEMSIZE option shows that the system memory available to SAS is set to 500 megabytes. The peak memory usage statistic (OS Memory) shows that the DATA step was bumping up against the memory made available to SAS[8]. Assuming that (1) there is sufficient physical memory on your system and (2) that a sufficient amount of this physical memory is free to be used by SAS (and is not otherwise being consumed by other processes running on your system) you could increase the value of MEMSIZE to keep the DATA step from running out of memory. In the following example, the MEMSIZE option in the configuration file ("C:\Program Files\SASHome\SASFoundation \9.3\nls\en\sasv9.cfg") is changed from "500M" to "1G" to increase the amount of memory available to SAS.

```
...
-WORK "!TEMP\SAS Temporary Files"
*-MEMSIZE 500M
-MEMSIZE 1G
...
```

With this change saved, SAS is restarted, and the DATA step is rerun. This time, the hash object was successfully loaded—reading 20 million records from "professional_fees" and writing the 100 million row claims file "pro_claims". The output made available by the FULLSTIMER option shows that 655 megabytes were needed to successfully load the hash object and complete the DATA step.

```
NOTE: There were 20000000 observations read from the data set EIW.PROFESSIONAL_FEES.
NOTE: There were 100000000 observations read from the data set EIW.CLAIMS.
NOTE: The data set WORK.PRO_CLAIMS has 100000000 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           6:40.59
      user cpu time       3:46.76
      system cpu time     28.73 seconds
      memory              655982.13k
      OS Memory           666536.00k
      Timestamp           10/19/2013 04:56:09 PM
```

Admittedly, though, if the dataset sourcing your hash object is "very large" and/or involves a large number of variables, you probably want to determine ahead of time whether you have enough memory to accommodate the intended hash object. To determine the expected memory needs of your hash object, you could take a small subset of the dataset that will source your hash object and utilize the "NUM_ITEMS" and "ITEM_SIZE" attributes of the hash object in order to estimate the amount of memory required by the hash object.

```
DATA work.profees_small;
 SET eiw.professional_fees (OBS=50);
RUN;
```

[8] See Williams, Easter and Bradshear (2009) for more information on the statistics available from FULLSTIMER and its use in performance tuning your SAS program.

```
DATA _NULL_;
IF _N_ = 0 THEN SET work.profees_small;

  DECLARE hash profee(dataset:'work.profees_small', hashexp:8);
    profee.DEFINEKEY ('claimid');
    profee.DEFINEDATA('pro_fees');
    profee.DEFINEDONE();

item_size = profee.ITEM_SIZE;
PUT 'hash entry size: ' item_size;
num_items = profee.NUM_ITEMS;
PUT 'num hash entries: ' num_items;
total_memory = (item_size*num_items)/1024;
PUT 'total hash memory estimate: ' total_memory;
STOP;
RUN;
```

In the preceding example, the "professional_claims" dataset is subset to 50 rows of data and this subset is used to source the hash object "profee". After the hash object has been defined, the "NUM_ITEMS" and "ITEM_SIZE" attributes of the hash object are used to estimate the number, size, and total memory used by the hash object. As illustrated by the following log entries, each entry in the hash object is estimated to use 32 bytes of memory. There are 50 elements in the table, so the total memory estimate for this hash object is 1.6 kilobytes.

```
NOTE: There were 50 observations read from the data set WORK.profees_small.
**hash entry size: 32
**num hash entries: 50
**total hash memory estimate: 1.5625
```

As describes by SAS, Inc. (2011b, p. 2110 – 2111), however:

The ITEM_SIZE attribute returns the size (in bytes) of an item, which includes the key and data variables and some additional internal information. You can set an estimate of how much memory the hash object is using with the ITEM_SIZE and NUM_ITEMS attributes. The ITEM_SIZE attribute does not reflect the initial overhead that the hash object requires, nor does it take into account any necessary internal alignments. Therefore, the use of ITEM_SIZE does not provide exact memory usage, but it does return a good approximation.

Applying this same estimation technique to our previous example (in which the hash object loaded 20 million records from "professional_fees"), the estimate (625 MB) is a fairly close approximation to the 656 MB actual memory usage.

```
NOTE: There were 20000000 observations read from the data set EIW.PROFESSIONAL_FEES.
**hash entry size: 32
**num hash entries: 20000000
**total hash memory estimate: 625000
NOTE: There were 100000000 observations read from the data set EIW.CLAIMS.
NOTE: The data set WORK.PRO_CLAIMS has 100000000 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           6:40.59
      user cpu time       3:46.76
      system cpu time     28.73 seconds
      memory              655982.13k
      OS Memory           666536.00k
      Timestamp           10/19/2013 04:56:09 PM
```

Using the previous technique, you can arrive at a fairly good estimate of the memory needs of your intended hash object. The next consideration is whether your system has enough available memory to handle the anticipated usage. If you do not know your MEMSIZE setting, you can determine it using the "GETOPTION" function as in the following example.

```
DATA _NULL_;
     memsize = (GETOPTION('MEMSIZE'));
     PUT '**mem=' memsize;
RUN;

**memsize=4294967295
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

The system on which this DATA step was run has MEMSIZE set to 4 gigabytes. However just because MEMSIZE is set at 4 gigabytes does not mean that there are 4 gigabytes of memory available to SAS. Other processes may be consuming resources on the system as well. Therefore, as Secosky and Bloom (2007, p. 6-7), suggest:

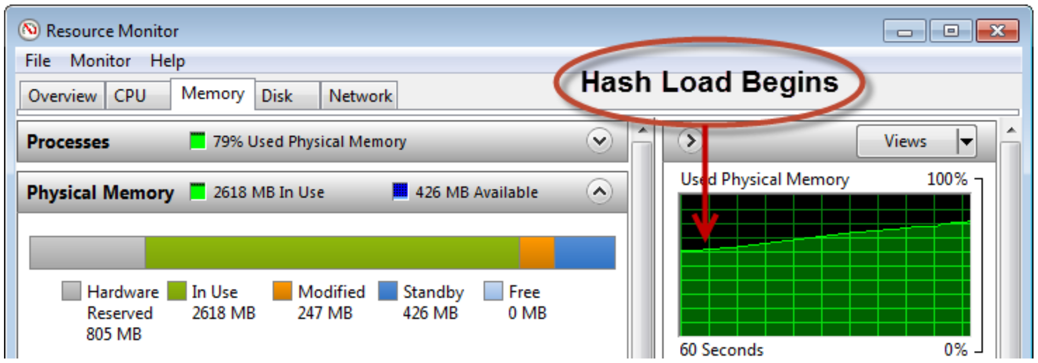
> *In order to determine the amount of memory available to SAS, the system option XMRLMEM can be retrieved. XMRLMEM is an undocumented diagnostic option that reports the amount of memory in bytes available to SAS without involving the operating system's virtual memory. While a rough number, it gives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

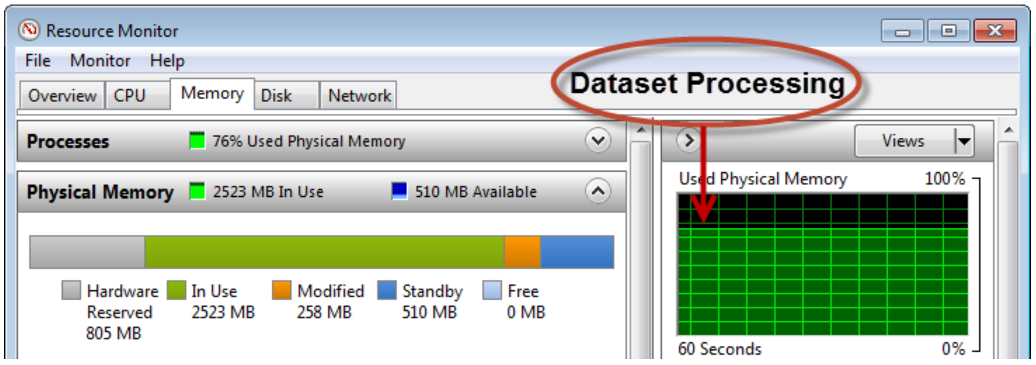
The following example demonstrates that the same system that has MEMSIZE set to 4 gigabytes has only 1.5 gigabytes available to the SAS session. Therfore, the upper limit on the size of a hash object on this system would be 1.5 gigabytes.

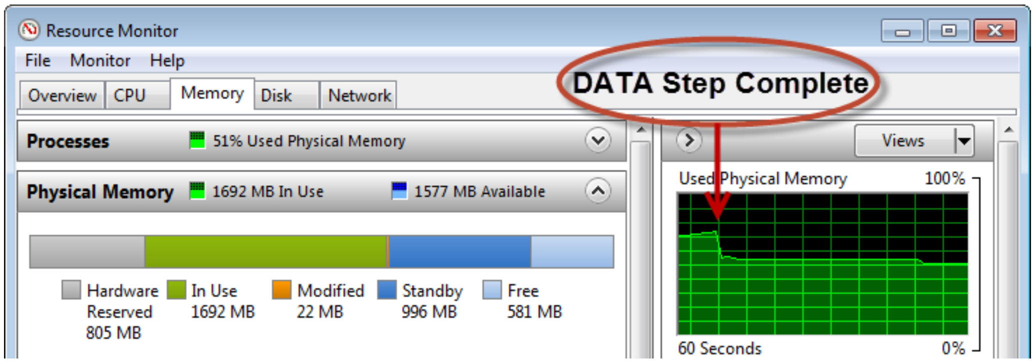
```
DATA _NULL_;
     mem = (GETOPTION('XMRLMEM'));
     PUT '**mem=' mem;
RUN;

**mem=1505112064
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

THE TRANSIENT HASH OBJECT

When DATA steps utilizing hash objects complete, the memory that they utilize is released. As demonstrated in the following measurements of memory usage during the previous, successful use of the hash object to lookup values from "professional_fees", memory utilization increases while the hash object is loaded, remains at the relatively high steady state while the DATA step is processed, and then drops back to the previous level when the memory occupied by the hash object is released at the end of the DATA step.







The release of the memory used by the hash object is indicative of one of the key characteristics of the hash object; it is a transient entity that "lives" only for the length of the DATA step in which it is created. Once the DATA step completes, the hash object ceases to exist. Because of this, attempts to call the hash object in a DATA step other than the one in which it was created will inevitably fail. If, for example, you attempted to access the "profee" hash object created in the previous DATA step, your code would generate an error informing you that "profee" is not an object; it no longer exists.

```
DATA _NULL_;
  item_size = profee.ITEM_SIZE;
  PUT 'hash entry size: ' item_size;
RUN;
```

```
358  DATA _NULL_;
359    item_size = profee.ITEM_SIZE;
                   ----------------
                   557
ERROR: DATA STEP Component Object failure.  Aborted during the COMPILATION phase.
ERROR 557-185: Variable profee is not an object.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

WHAT'S IN A NAME

Note also that in the preceding example, the SAS log refers to "profee" as a "variable". The hash object is identified by its name in the PDV, so it is no surprise that the previous SAS error refers to it in this way. However, it is important to recognize that because the hash object occupies a named space within the PDV, no dataset variables can occupy that same space. In the following example, the dataset "professional_fees" contains a variable named "profee".

```
DATA work.pro_claims;
IF _N_ = 0 THEN SET eiw.professional_fees;

  DECLARE hash profee(dataset:'eiw.professional_fees');
    profee.DEFINEKEY ('claimid');
    profee.DEFINEDATA('bill_amt');
    profee.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = profee.FIND();
     IF rc NE 0 THEN pro_fees = 0;
 OUTPUT;
END;
STOP;
RUN;
```

When the preceding code is executed, it generates the following log entries—indicating that the hash object "profee" cannot be created because "profee" is already defined. "Profee" was defined, of course, when the variables in the dataset "professional_fees" were identified by the "SET" statement "`IF _N_ = 0 THEN SET eiw.professional_fees;`".

```
368  DATA work.pro_claims;
369  IF _N_ = 0 THEN SET eiw.professional_fees;
370
371    DECLARE hash profee(dataset:'eiw.professional_fees');
                           -
                           567
372      profee.DEFINEKEY ('claimid');
         ----------------
         557
ERROR: DATA STEP Component Object failure.  Aborted during the COMPILATION phase.
ERROR 567-185: Variable profee already defined.
ERROR 557-185: Variable profee is not an object.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.00 seconds
```

This error can be resolved easily by giving the hash object another name (say, "profees"); then there will be no name conflict between the hash object and the variables in the dataset that sources it.

Conversely, if the hash object is assigned the same name as a variable in the dataset being processed, the name conflict generates a different error, but the cause is the same—attempting to create identically named variables in the PDV. In the following example, the dataset "claims" contains a variable called "profee". By the time SAS begins processing the DO-UNTIL LOOP, the PDV already contains an entry for "profee" (the hash object), so when the SET statement attempts to start processing records from "claims" there is already an entry in the PDV (of type, "object"). Because the scalar "profee" variable cannot be placed in the PDV, the DATA step stops with an error.

```
481
482  DATA work.pro_claims;
483  IF _N_ = 0 THEN SET work.small;
484
```

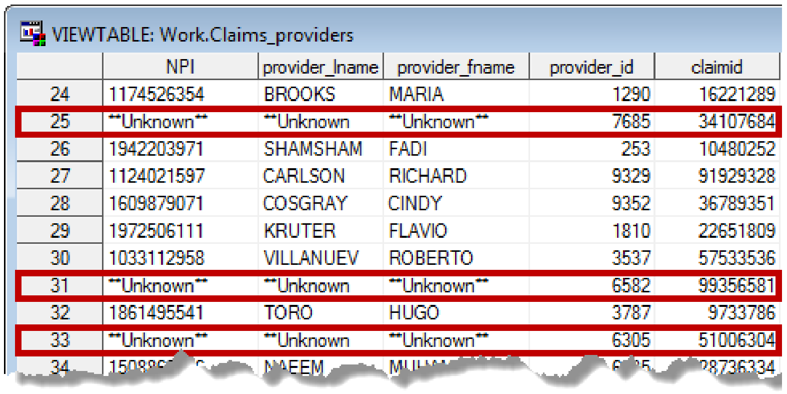
```
485    DECLARE hash profee(dataset:'small');
486      profee.DEFINEKEY ('claimid');
487      profee.DEFINEDATA('var_a');
488      profee.DEFINEDONE();
489
490  DO UNTIL (eof_claims);
491   SET work.claims END = eof_claims;
ERROR: Variable profee has been defined as both object and scalar.
```

KNOW YOUR DATA AND CONTROL LOOKUP RESULTS WRITTEN TO THE OUTPUT RECORD

In addition to the hash object issues that result in obvious errors generated to the LOG, the nuances of working with the hash object also involve explicitly controlling the values assigned to the variables you are looking up from the hash object. In the following example, the dataset "claims" does not contain variables named "provider_lname", "provider_fname" and "npi". The first and last names of the associated provider for each claim (and his or her NPI) are looked up from the hash object "provider" based on the value of "provider_id" on the "claims" record.

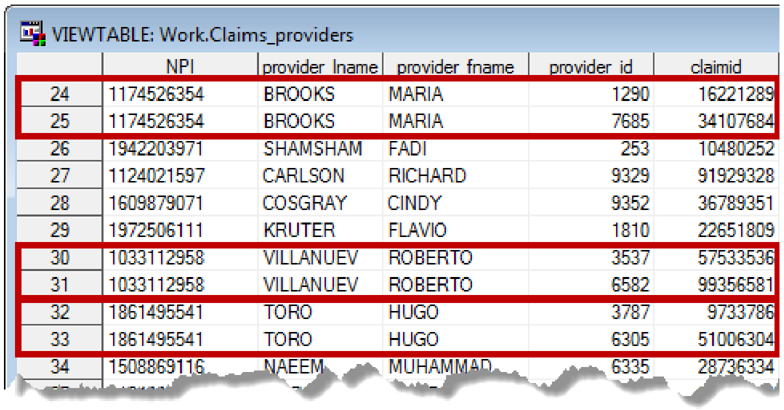
```
DATA work.claims_providers (keep= claimid provider_id npi provider_lname
                                  provider_fname );
IF _N_ = 0 THEN SET eiw.providers;

  DECLARE hash provider(dataset:'eiw.providers');
    provider.DEFINEKEY ('provider_id');
    provider.DEFINEDATA('provider_lname', 'provider_fname','npi');
    provider.DEFINEDONE();

DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = provider.FIND();
     IF rc ne 0 THEN DO;
        provider_lname = '**Unknown**';
        provider_fname = '**Unknown**';
        npi = '**Unknown**';
     END;
 OUTPUT;
END;
STOP;
RUN;
```

If a match to the provider id is not found in the hash object (rc not equal to 0) then the variables populated by the search are assigned the value "**Unknown**".


VIEWTABLE: Work.Claims_providers

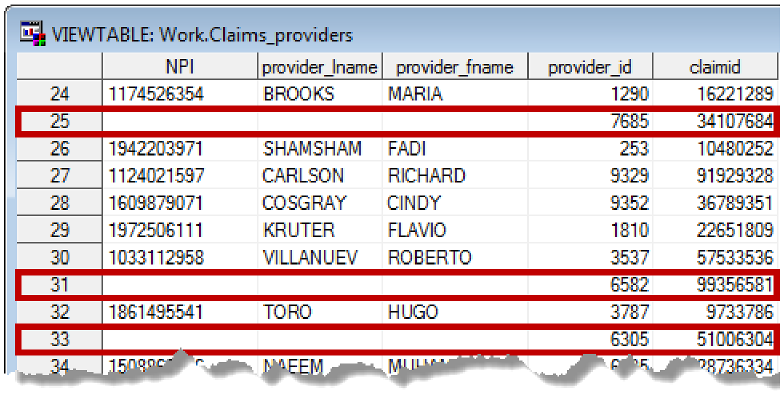
	NPI	provider_lname	provider_fname	provider_id	claimid
24	1174526354	BROOKS	MARIA	1290	16221289
25	**Unknown**	**Unknown	**Unknown**	7685	34107684
26	1942203971	SHAMSHAM	FADI	253	10480252
27	1124021597	CARLSON	RICHARD	9329	91929328
28	1609879071	COSGRAY	CINDY	9352	36789351
29	1972506111	KRUTER	FLAVIO	1810	22651809
30	1033112958	VILLANUEV	ROBERTO	3537	57533536
31	**Unknown**	**Unknown	**Unknown**	6582	99356581
32	1861495541	TORO	HUGO	3787	9733786
33	**Unknown**	**Unknown	**Unknown**	6305	51006304
34	[illegible]	NAEEM	[illegible]	[illegible]	28736334

Unlike PDV values that are automatically valued by loading a new record from "claims", the PDV entries that contain values returned from the hash object entries are not automatically cleared as each record in "claims" is processed. If a match is found in the hash object for the current value of "claimid", the values contained in the hash object entry are assigned to the PDV entries for "provider_fname", provider_lname", and "npi". However, if a match is not found, there is no implicit "clearing" of the PDV values associated with these variables; this is why the explicit assignment of "**Unknown**" is so important in this case. If that assignment is not made, the value currently assigned in the PDV (from the most recent successful search of the hash object) will be assigned to the current "claims_providers" output record. Instead of "**Unknown**", the "claims_providers" record for which there was no match in the hash object will be assigned the values returned from the most recent successful search.


VIEWTABLE: Work.Claims_providers

	NPI	provider_lname	provider_fname	provider_id	claimid
24	1174526354	BROOKS	MARIA	1290	16221289
25	1174526354	BROOKS	MARIA	7685	34107684
26	1942203971	SHAMSHAM	FADI	253	10480252
27	1124021597	CARLSON	RICHARD	9329	91929328
28	1609879071	COSGRAY	CINDY	9352	36789351
29	1972506111	KRUTER	FLAVIO	1810	22651809
30	1033112958	VILLANUEV	ROBERTO	3537	57533536
31	1033112958	VILLANUEV	ROBERTO	6582	99356581
32	1861495541	TORO	HUGO	3787	9733786
33	1861495541	TORO	HUGO	6305	51006304
34	1508869116	NAEEM	MUHAMMAD	6335	28736334

If, on the other hand, the "claims" dataset does contain variables named "npi", "provider_lname", and "provider_fname", then, each record fetched into the PDV from "claims" will replace the values of these variables in the PDV and the hash object lookup will only replace the values in those instances where the search of the hash object is successful. In this scenario, the removal of the controlling "IF" statement will not result in the errors encountered in the preceding scenario.


VIEWTABLE: Work.Claims_providers

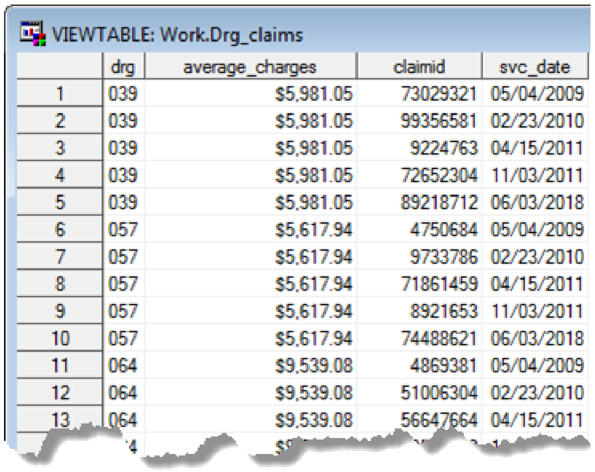
	NPI	provider_lname	provider_fname	provider_id	claimid
24	1174526354	BROOKS	MARIA	1290	16221289
25				7685	34107684
26	1942203971	SHAMSHAM	FADI	253	10480252
27	1124021597	CARLSON	RICHARD	9329	91929328
28	1609879071	COSGRAY	CINDY	9352	36789351
29	1972506111	KRUTER	FLAVIO	1810	22651809
30	1033112958	VILLANUEV	ROBERTO	3537	57533536
31				6582	99356581
32	1861495541	TORO	HUGO	3787	9733786
33				6305	51006304
34	15088[illegible]	NAEEM	[illegible]	[illegible]	[illegible]8736334

HANDLING DUPLICATE RECORDS

Another characteristic of your data that is vitally important to understand when using hash objects to perform lookups is the combination of columns (variables) that uniquely identify each record in the hash object's source dataset—the primary key. With one exception, all of the previous examples have used a single-column integer primary key to lookup data in the hash object (e.g., "provider_id", "claimid, etc.). In the following example, a dataset "eiw.drg_charges" is used to add the "average_charge" for a given diagnosis-related group (DRG) to each record in "claims".

```
DATA work.drg_claims (keep=claimid drg average_charges svc_date);
IF _N_ = 0 THEN SET work.drg_charges;

  DECLARE hash drgs(dataset:'work.drg_charges');
    drgs.DEFINEKEY ('drg');
    drgs.DEFINEDATA('average_charges');
    drgs.DEFINEDONE();
DO UNTIL (eof_claims);
 SET work.claims END = eof_claims;
     rc = drgs.FIND();
     if rc ne 0 then average_charges = .;
  OUTPUT;
END;
STOP;
RUN;
```

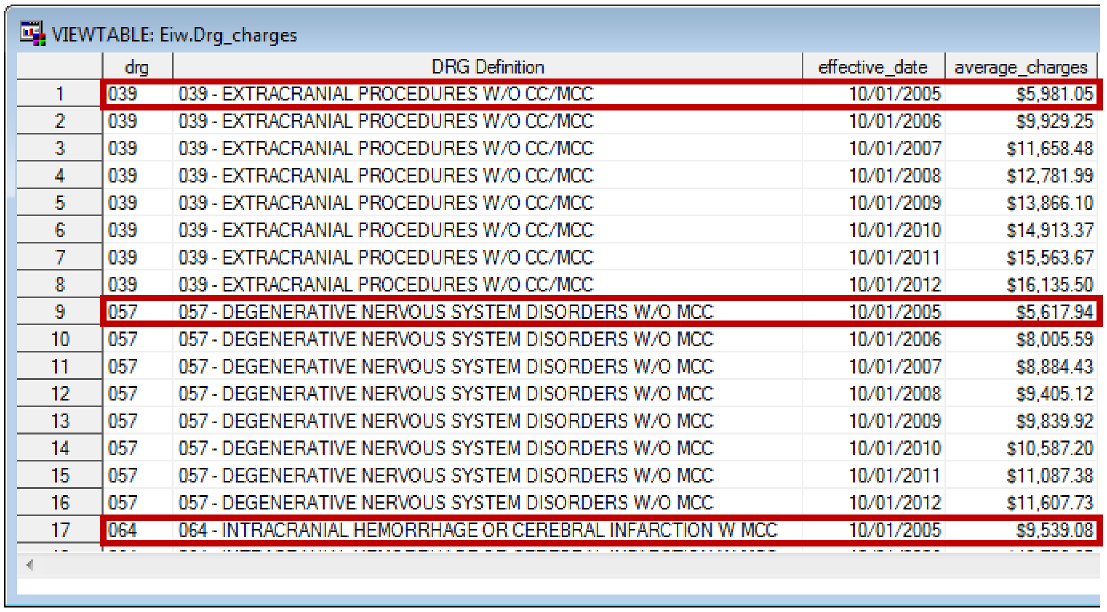
No errors are generated during execution of this code and the resulting dataset appears to show that the "average_charges" were added to each claim consistently according to the corresponding DRGs.



VIEWTABLE: Work.Drg_claims

	drg	average_charges	claimid	svc_date
1	039	$5,981.05	73029321	05/04/2009
2	039	$5,981.05	99356581	02/23/2010
3	039	$5,981.05	9224763	04/15/2011
4	039	$5,981.05	72652304	11/03/2011
5	039	$5,981.05	89218712	06/03/2018
6	057	$5,617.94	4750684	05/04/2009
7	057	$5,617.94	9733786	02/23/2010
8	057	$5,617.94	71861459	04/15/2011
9	057	$5,617.94	8921653	11/03/2011
10	057	$5,617.94	74488621	06/03/2018
11	064	$9,539.08	4869381	05/04/2009
12	064	$9,539.08	51006304	02/23/2010
13	064	$9,539.08	56647664	04/15/2011

However, looking at the dataset that sourced the hash object it becomes clear that the "average_charges" should have probably been assigned based on the "effective_date" of the "drg_charges" record and the "svc_date" of the claim. The "average_charges" corresponding to the period during which the first claim record was generated (svc_date = 05/04/2009) should have been $12,781.99. In fact, all of the claims indicating DRG value "039" have the same value of "average_charges" assigned—that associated with the first record in the "eiw.drg_charges" dataset with a DRG value of "039". By default, when hash objects are loaded only the first record with a given key value is loaded; subsequent records with the same key value are ignored. Therefore, the hash object in the preceding example contained only those entries highlighted below; records with duplicate keys were not loaded.



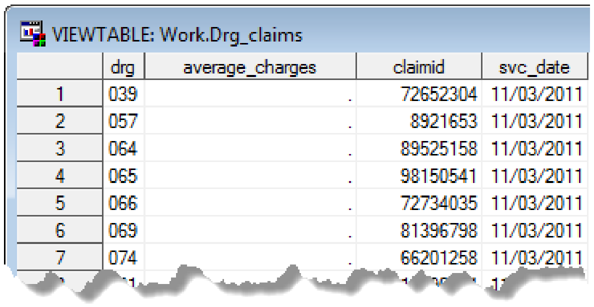
VIEWTABLE: Eiw.Drg_charges

	drg	DRG Definition	effective_date	average_charges
1	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2005	$5,981.05
2	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2006	$9,929.25
3	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2007	$11,658.48
4	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2008	$12,781.99
5	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2009	$13,866.10
6	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2010	$14,913.37
7	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2011	$15,563.67
8	039	039 - EXTRACRANIAL PROCEDURES W/O CC/MCC	10/01/2012	$16,135.50
9	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2005	$5,617.94
10	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2006	$8,005.59
11	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2007	$8,884.43
12	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2008	$9,405.12
13	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2009	$9,839.92
14	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2010	$10,587.20
15	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2011	$11,087.38
16	057	057 - DEGENERATIVE NERVOUS SYSTEM DISORDERS W/O MCC	10/01/2012	$11,607.73
17	064	064 - INTRACRANIAL HEMORRHAGE OR CEREBRAL INFARCTION W MCC	10/01/2005	$9,539.08

Of course, the data in "eiw.drg_charges" does have an appropriate primary key; it is a composite key created by the combination of the "drg" and "effective_date" columns. This combination makes each record in the dataset unique, and specifying this combination in the DEFINEKEY method would successfully load all of the records from "eiw.drg_charges" into the hash object.

```
DATA work.drg_claims (keep=claimid drg average_charges svc_date);
IF _N_ = 0 THEN SET eiw.drg_charges;

  DECLARE hash drgs(dataset:'eiw.drg_charges');
    drgs.DEFINEKEY ('drg','effective_date');
    drgs.DEFINEDATA('average_charges');
    drgs.DEFINEDONE();

DO UNTIL (eof_claims);
 SET work.claims END = eof_claims;
     rc = drgs.FIND(KEY: drg, KEY: svc_date);
     if rc ne 0 then average_charges = .;
  OUTPUT;
END;
STOP;
RUN;
```

However, performing the lookup based on this key will only result in a successful search in those rare instances in which the "svc_date" exactly matches the "effective_date" for the current "drg" value.


VIEWTABLE: Work.Drg_claims

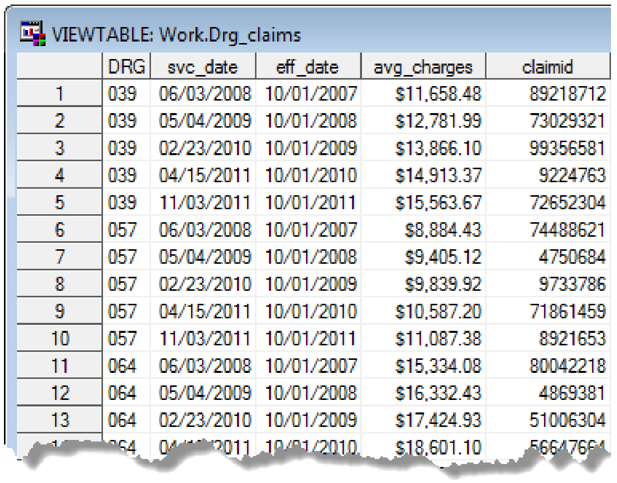
	drg	average_charges	claimid	svc_date
1	039	.	72652304	11/03/2011
2	057	.	8921653	11/03/2011
3	064	.	89525158	11/03/2011
4	065	.	98150541	11/03/2011
5	066	.	72734035	11/03/2011
6	069	.	81396798	11/03/2011
7	074	.	66201258	11/03/2011

In order to successfully perform the search in manner that will allow the correct comparison between the service date on the claim and the effective date on the "eiw.drg_charges" dataset, you need to take advantage of functionality available beginning in SAS 9.2 that allows duplicate key entries in the hash object (see Ray & Secosky, 2008 for an in-depth discussion of this functionality and the methods it enables). In the following example the MULTIDATA option is included in the hash object declaration so that multiple entries with the same "drg" value can be loaded. Therefore, there could potentially be multiple entries in the hash object that match the "drg" value on a given record. After the FIND method call, a DO WHILE loop is executed with the exit condition based on the value of the return code "rc"; the loop will process all hash object entries with a DRG that matches the "drg" value in the current "claims" record. The "svc_date" on the current "claims" record is evaluated against the "effective_date" value of each successful match from the hash object and that value is, in turn, evaluated against any values of effective date assigned from previously assessed matches returned from the hash object for that DRG. At the bottom of this loop, the FIND_NEXT method is used to advance the loop to the next matching entry returned from the hash object and it is evaluated in the same manner—eventually assigning the appropriate "averaged_charges" and "effective_date" to the local variables "avg_charges" and "eff_date", respectively, for output to the "drg_claims" record.

```
DATA work.drg_claims (keep=claimid drg svc_date avg_charges eff_date);
IF _N_ = 0 THEN SET work.drg_charges;

  DECLARE hash drgs(dataset:'work.drg_charges', multidata:'yes');
    drgs.DEFINEKEY ('drg');
    drgs.DEFINEDATA('average_charges','effective_date');
    drgs.DEFINEDONE();
```

```
DO UNTIL (eof_claims);
 SET eiw.claims END = eof_claims;
     rc = drgs.FIND();
    eff_date = .;
    DO WHILE (rc=0);
      IF svc_date >= effective_date > eff_date THEN DO;
         avg_charges = average_charges;
         eff_date = effective_date;
       END;
      rc=drgs.FIND_NEXT();
    END;
  OUTPUT;
END;
STOP;
RUN;
```

The resulting dataset now contains the "avg_charges" associated with the most recent effective date as of the service date of the claim.


VIEWTABLE: Work.Drg_claims

	DRG	svc_date	eff_date	avg_charges	claimid
1	039	06/03/2008	10/01/2007	$11,658.48	89218712
2	039	05/04/2009	10/01/2008	$12,781.99	73029321
3	039	02/23/2010	10/01/2009	$13,866.10	99356581
4	039	04/15/2011	10/01/2010	$14,913.37	9224763
5	039	11/03/2011	10/01/2011	$15,563.67	72652304
6	057	06/03/2008	10/01/2007	$8,884.43	74488621
7	057	05/04/2009	10/01/2008	$9,405.12	4750684
8	057	02/23/2010	10/01/2009	$9,839.92	9733786
9	057	04/15/2011	10/01/2010	$10,587.20	71861459
10	057	11/03/2011	10/01/2011	$11,087.38	8921653
11	064	06/03/2008	10/01/2007	$15,334.08	80042218
12	064	05/04/2009	10/01/2008	$16,332.43	4869381
13	064	02/23/2010	10/01/2009	$17,424.93	51006304
[illegible]	[illegible]	[illegible]	10/01/2010	$18,601.10	[illegible]

ADVANCED HASHING / NEXT STEPS

As demonstrated in the preceding example, the functionality of the SAS hash object continues to evolve, and methods for applying it to solve new problems evolve along with it. As you become comfortable with the hash object, you might consider expanding its use beyond the types of simple lookups demonstrated in the current work. As demonstrated elsewhere, the hash object can also be used to perform sorted file output (Eberhardt, 2011) and file-splitting (Dorfman, Vyverman, & Dorfman, 2010), and you can even create hashes of hashes (Dorfman & Vyverman, 2009; Hinson, 2013; Loren & Devenezia, 2011) to apply hashing techniques to hierarchical data processing problems.

REFERENCES

Dorfman, P. (1999). Array Lookup Techniques: From Sequential Search to Key-Indexing (Part 1). Proceedings of the Southeast SAS Users Group.

Dorfman, P. (2000a). Table Lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing. Proceedings of the Southeast SAS Users Group.

Dorfman, P. (2000b). Private Detectives in a Data Warehouse: Key-Indexing, Bitmapping, and Hashing. Proceedings of the 25th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Dorfman, P. (2001). Table Lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing. Proceedings of the 26th Annual SAS Users Group International Meeting.

Dorfman, P. & Eberhardt, P. (2011). Two Guys on Hash. Proceedings of the Southeast SAS Users Group.

Dorfman, P., Shajenko, L., & Vyverman, K. (2008). Hash Crash and Beyond. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.
Dorfman, P. & Snell, G. (2002). Hashing Rehashed. Proceedings of the 27th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Dorfman, P. & Snell, G. (2003). Hashing: Generations. Proceedings of the 28th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Dorfman, P. & Vyverman, K. (2006). Data Step Hash Objects as Programming Tools. Proceedings of the 26th Annual SAS Users Group International Meeting.
Dorfman, P. & Vyverman, K. (2009). The SAS Hash Object in Action. Proceedings of the SAS Global Forum 2009. Cary, NC: SAS Institute, Inc.
Dorfman, P., Vyverman, K., & Dorfman, V. (2010). Black Belt Hashigana. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.
Eberhardt, P. (2011). The SAS Hash Object: It's Time To .find() Your Way Around. Proceedings of the SAS Global Forum 2011. Cary, NC: SAS Institute, Inc.
Hinson, J. (2013). The Hash-of-Hashes as a "Russian Doll" Structure: An Example with XML Creation. Proceedings of the SAS Global Forum 2013. Cary, NC: SAS Institute, Inc.
Lafler, K.P. (2004a). Efficiency Techniques for Beginning PROC SQL Users. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Lafler, K.P. (2004b). PROC SQL: Beyond the Basics Using SAS. Cary, NC: SAS Institute, Inc.
Lafler, K.P. (2005). Manipulating Data with PROC SQL. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Loren, J. & DeVenezia, R. (2011). Building Provider Panels: An Application for the Hash of Hashes. Proceedings of the SAS Global Forum 2011. Cary, NC: SAS Institute, Inc.
Ray, R. & Secosky, J. (2008). Better Hashing in SAS® 9.2. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.
SAS Institute, Inc. (2004). What's new in SAS® 9.0, 9.1, 9.1.2, and 9.1.3. Cary, NC: SAS Institute Inc.
SAS Institute, Inc. (2011a). SAS® 9.3 Component Objects: Reference. Cary, NC: SAS Institute Inc.
SAS Institute, Inc. (2011b). SAS® 9.2 Language Reference: Dictionary, Fourth Edition. Cary, NC: SAS Institute, Inc.
Schacherer, C. (2011). Base SAS Methods for building Dimensional Data Models. Proceedings of SAS Global Forum 2012. Cary, NC: SAS Institute, Inc.
Schacherer, C. (2013). SAS® Data Management Techniques: Cleaning and Transforming Data for Delivery of Analytic Data Sets. Proceedings of SAS Global Forum 2013. Cary, NC: SAS Institute, Inc.
Schacherer, C. & Westra, B. (2010). A SAS Primer for Healthcare Data Analysts. Proceedings of the South Central SAS Users Group.
Secosky, J. & Bloom, J. (2007). Getting Started with the DATA Step Hash Object. Proceedings of SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.
Slaughter, S. & Delwiche, L. (1997). Errors, Warnings, and Notes (Oh My): A Practical Guide to Debugging SAS® Programs. Proceedings of the 26th Annual SAS Users Group International Meeting.
Warner-Freeman, J. (2007). I cut my processing time by 90% using hash tables – You can do it too! Proceedings of the Northeast SAS Users Group.
Whitlock, I. (2006). How to Think Through the SAS DATA Step. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
Williams, M., Easter, G., & Bradsher, S. (2009). Troubleshoot Your Performance Issues: SAS® Technical Support Shows You How. Proceedings of the SAS Global Forum 2009. Cary, NC: SAS Institute, Inc.

CONTACT INFORMATION

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

Christopher W. Schacherer, Ph.D.
Clinical Data Management Systems, LLC
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com