Paper S1-24

The SAS® Hash Object: It’s Time To .find() Your Way Around

Peter Eberhardt, Fernwood Consulting Group Inc.
Toronto ON, Canada

ABSTRACT

“This is the way I have always done it and it works fine for me.”

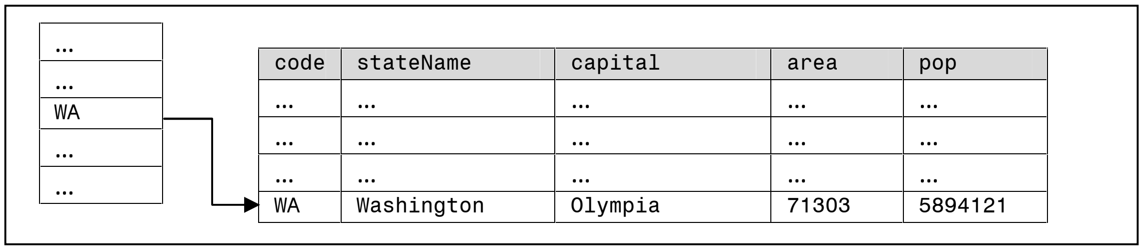
Have you heard yourself or others say this when someone suggests a new technique to help solve a problem? Most of us have a set of tricks and techniques from which we draw when starting a new project. Over time we might overlook newer techniques because our old toolkit works just fine. Sometimes we actively avoid new techniques because our initial foray leaves us daunted by the steep learning curve to mastery. For me, the PRX functions and the SAS® hash object fell into this category.

In this workshop, we address possible objections to learning to use the SAS hash object. We start with the fundamentals of the setting up the hash object and work through a variety of practical examples to help you master this powerful technique.

INTRODUCTION

Searching and table lookups are ubiquitous in SAS programmes. Many SAS programmers may not think of themselves as performing searches or table lookups in their programmes, however, if they have ever done a DATA step merge or a SQL join they have indeed performed searches and lookups. In general a table lookup is the process in which a ***key*** value is matched against a table of ***key/value*** pairs and the ***value*** is returned. For example, a common table lookup would take a state code (the ***key***) and match it against a state code/state name table (the ***key/value*** pair) to return the state name. The ***value*** part of the ***key/value*** can be, and often is, more complex than a single value; using the state table example, the ***value*** part of the table may be:

- State name
- State area
- State population
- State capital

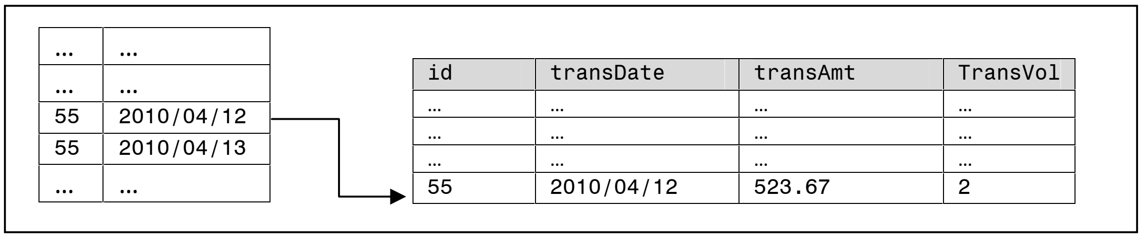


…
…
WA
…
…

code	stateName	capital	area	pop
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
WA	Washington	Olympia	71303	5894121

Figure 1. Simple Key and Complex Value

In addition, the ***key*** part may also be complex. For example a customer ID and transaction date may be the ***key***, and a transaction amount and transaction volume may the ***value*** (Figure 2).



…	…
…	…
55	2010/04/12
55	2010/04/13
…	…

id	transDate	transAmt	TransVol
…	…	…	…
…	…	…	…
…	…	…	…
55	2010/04/12	523.67	2

Figure 2. Complex Key and Value

The code to do these sorts of lookup will be familiar to most SAS programmers. Sample data step code could be:

```
DATA custValues;
     merge custDates custTrans;
     by ID transDate;
     keep ID transDate transAmt transVol;
run;
```

The equivalent SQL code could be:

```
PROC SQL;
      create table custValues as
      select cv.*
      from custDates as cd inner join custTrans as ct
      on  cd.ID        = ct.ID
      and ct.transDate = ct.transDate
      order by ct.ID, ct.transDAte;
QUIT;
```

There are other methods within SAS to perform table lookups, one of the most common being the use of SAS formats and the PUT() function; this has been the workhorse in-memory lookup method of choice by SAS programmers for years. In addition, some complex but efficient in memory lookups have been devised. Starting in SAS v9 a new, flexible and efficient method to do lookups was introduced: the SAS Hash Object..

The paper starts with a discussion about the general workings of the SAS® hash object, followed by an introduction to some object-oriented terms. From there, we learn by example, beginning with the fundamentals of setting up the hash object. Then we work through a variety of practical examples to help you master this powerful technique. Before we start into the hash object, I will introduce the data that will be used in the examples.

SAMPLE DATA

This paper will use four datasets. :

- ***FEECODES*** – a table of codes (5,436 rows) doctors can bill. The codes are grouped by a Section and Sub-Section of a billing schedule. There is also a dollar value for each code. This table is based upon data publicly available on the Ontario Ministry of Health and Long Term Care website (http://www.health.gov.on.ca/english/providers/program/ohip/sob/sob_mn.html)
- ***PATIENTS*** – a list of patients (10,281,246 rows) with some demographic information
- ***DOCTORS*** – a list of medical doctors (22,000 rows) with some demographic information
- ***TRANSACTIONS*** – a list of billing transactions (6,502 rows)

See Appendix A for the column names and the relationships.

Note: the contents of the PATIENTS, DOCTORS and TRANSACTIONS tables are all simulated data. The contents of the FEECODES table are based upon fee codes publically available on the Ontario Ministry of Health and Long Term Care website.

THE SAS HASH OBJECT

Unlike a DATA step merge and a SQL join which are disk based lookups, that is the look up table is a disk file which SAS will access repeatedly in the lookup process, the hash object is memory based; this means the table is read into memory once and it is accessed repeatedly in memory. Since memory based operations are faster than disk based operations we would expect lookups based upon a hash object to be faster than the disk based alternatives; a review of the literature shows this. Although there are mechanisms to tell SAS to load smaller tables into memory, or times when SAS will read smaller tables into memory as part of its own optimizations, DATA step merge and SQL joins are fundamentally disk based methods.

The first thing to know about the hash object is that it is an artifact of the DATA step; it cannot be used in PROC SQL or in any other SAS PROC. Moreover, it is a transient artifact of the DATA step; that is it is created in a DATA step, and when the DATA step ends the hash object disappears. The data that are loaded into the hash object can be saved in a SAS dataset, but the hash object itself is not saved. This is behavior similar to SAS arrays; the columns which make up an array may be saved in a SAS dataset, but the array disappears.

A DATA step is not limited to one hash object; you can create and use as many hash objects as you need. However, since hash objects are memory resident, you may end up with an out of memory problem if you load too much into hash objects. In general you will want to load the smaller lookup tables into hash objects and sequentially process the larger table that has the lookup keys.

A hash object can have an associated iterator object. While the hash object actually stores your data in memory, the associated iterator object is used to traverse the hash object – either forward (i.e. from first to last record) or backward (i.e. from last to first record).

The memory required for a hash object (and an associated iterator object) is allocated at run time; as you add or remove items (records) from the hash object SAS will automatically allocate or release memory. If you need memory for 10 items, then SAS will allocate memory for 10 items. If you need memory for 10 million items, SAS will allocate memory for 10 million items. You do not need to tell the hash object how much memory to allocate.

Data do not need to be sorted or indexed before they can be used in a hash table; moreover, the dataset with lookup keys can be processed in its existing order. Removing the need to sort before (or during as PROC SQL may do) lookup can be a tremendous saving. If you have a large table (say over 150 million rows) that is already in the order you want it, clearly we will have savings. In our example data the TRANSACTIONS dataset is in transaction date order with one entry for each date/doctor/patient/fee code combination. In a real world example this table would easily be 150 million rows. In order to lookup doctor, patient, and fee code information this table would have to be sorted four times

1. By doctor ID (DID)
2. By patient ID (PID)
3. By fee code (FEECODE)
4. By visit date (VISITDATE), doctor ID, patient ID, and fee code to return it to the correct order.

Clearly we should expect dramatic improvements in processing time if we load the lookup tables (DOCTORS, PATIENTS, and FEECODES) into memory, then do one sequential pass of the transactions dataset. Since the purpose of this paper (and the workshop based upon it) is to demonstrate the use of the hash object and not its efficiencies, the TRANSACTIONS dataset is relatively small; the other datasets are essentially real world sizes.

STEPS TO CREATING AND USING A HASH OBJECT

For simplicity I will break CREATING and USING into separate steps. In addition, there are alternative ways to create and use a hash object; as you gain more experience in using the hash object you should experiment with alternatives until you find steps that best suit your problems. Here is a sample DATA step that demonstrates creating and using a hash object:

```
DATA exercise01_loop;
* inform the datastep about the variables for the hash;
* it is YOUR responsibility to make sure the variables have the correct attributes;
     length feecode    $4.                      2
            section    $1.
            subSection 8.
            feeAmount  8.
            ;
     * there are multiple ways to declare the object;
     DECLARE hash  feecodes() ;                 1

     * inform the object which variable is the key;
     * NOTE: we DO NOT use the actual variable in the definition;
     *       we use character string.;
     rc=feecodes.defineKey('feecode');          2

     * you could also use character variables: ;
     * key='feecode';
     * rc=feecodes.defineKey(key);

     * inform the object which variable(s) is/are the data;
     rc=feecodes.defineData('section', 'subsection', 'feeAmount');   3

     * inform SAS we are done with the definition;
     rc=feecodes.defineDone();                  4

     do while (not done);
        set data.feecodes end=done;
        * good practice to capture the return code;
        rc = feecodes.add();                    5
        *even better practice to check for errors;
        if rc NE 0
        then
          do;
             put "Problem with .add()." feecode= section= subsection= feeAmount=;
          end;
     end;
     * we are doing nothing else, so stop;
     STOP;
RUN ;
```

To create a hash object there are four basics steps to follow; these are outlined below.

1. DECLARE the object:

```
DECLARE hash feecodes();
```

The DECLARE statement is used to name and instantiate (allocate memory for) the object. Although it is possible to DECLARE and instantiate using two commands all of the examples in the paper name and instantiate in the same statement.

In this example a hash object called ***feecodes*** is being created. If the parenthesis were left off the name, then the object would be created but not instantiated; the object would need to be instantiated with the _***NEW***_operator in a separate statement.

There several optional arguments that are available when declaring a hash object, Some of these will be introduced throughout the paper.

2. DEFINE the hash key.

```
rc=feecodes.defineKey('feecode');
```

feecodes is the name of the hash object, and ***defineKey()*** is one of its methods (a short discussion on object oriented terminology follows). The arguments to the ***defineKey()*** method are name(s) of the key variables, in this case feecode. Note that the argument is a string with the name of the variable, not the variable itself. Also note that the attributes of feecode (character variable, length 4) were defined earlier in the DATA step. Memory is allocated based upon the type and size of the data being loaded – it is your responsibility to ensure the key and data elements of the hash object are correctly defined. All the hash object methods return a code indicating success (0) or failure (~ 0).It is a good practice to capture and verify the return code.

3. DEFINE the data variables.

```
rc=feecodes.defineData('section', 'subsection', 'feeAmount');
```

feecodes is the name of the hash object, and ***defineData()*** is another of its methods. As with the key variables in the ***defineKey()*** method, the name(s) of the variables which constitute the values to be returned are passed in as strings. If you need the hash object to return the key value when the hash item is accessed, then the column(s) used as the key must be included in the list of data columns.

4. Complete the definitions

```
rc=feecodes.defineDone();
```

feecodes is the name of the hash object, and ***defineDone()*** is another of its methods. This method completes the definition of the hash object.

To use a hash object there are numerous methods available. In order to take advantage of the hash object we need to load the data:

5. Load data into a hash object

```
rc = feecodes.add();
```

feecodes is the name of the hash object, and ***add()*** is another of its methods. In this example we are reading each record from the FEECODES dataset in a loop, and adding each record to the hash object, also called ***feecodes***. I find it useful to give my hash object the same name as the underlying dataset it represents. Note that in addition to capturing the return code from the ***add()*** method, the programme is also testing for failure. As we will see later, there are reasons why the ***add()*** method will fail.

Although this sample programme demonstrates the important elements of creating and using a hash object, there is much more we can do. Before we delve deeper into the hash object, a brief review of some object oriented terminology is in order.

OBJECT ORIENTED TERMINOLOGY

To an experienced SAS programmer there is a lot that looks and 'sounds' odd about the SAS hash object. This is because the SAS hash object introduces a different programming paradigm into the SAS DATA step language – the Object Oriented programming paradigm.

In a DATA step programme we are used to dealing with variables (also called columns or fields); variables are simple data types – in SAS variables can be character or numeric. We can perform operations directly on variables (assign a value, add, multiply, etc,). We also have functions/call routines which can assign values to a variable or use variables for calculations. Objects on the other hand are complex data types. In general you cannot perform operations directly on objects; we can perform operations on an object only through the interface it makes available. Allowing access to the object only through its interface helps to ensure data integrity.

One of the first differences you will note is what the SAS documentation calls the "Dot Notation"; basically this is the method of using the interface of the object. With the SAS hash objects we have two interface types:

- Methods
- Attributes

Loosely speaking methods and attributes 'belong' to the object and to invoke them you have to specify both the object and the method or attribute. Speaking even more loosely methods are like functions that perform some action on the object, and attributes are values we can set/retrieve from the object. For example to invoke the ***defineKey()*** method on the ***feecodes*** object:

```
rc=feecodes.defineKey('feecode');
```

If we had a second object called ***doctors***, we would invoke its ***defineKey()*** method like:

```
rc=doctors.defineKey('DID');
```

How do we tell the difference between invoking a method and accessing an attribute on a hash object. First, since there are currently only two attributes available for the hash object we can probably remember them. The attributes available are:

- Item_size – the number of bytes each item in the hash object requires. Due to memory alignment issues and other internal storage issues, this may not be the exact amount memory used by each item

```
sizeInBytes = feecodes.item_size
```

- Num_items –the number of items in the hash object.

```
itemsInHash = feecodes.num_items
```

Both of these attributes are read-only; you can access the attribute but you cannot directly change the value of the attribute. Of course if you add or remove items in the hash you will indirectly change the attribute value of num_items. When you define the key and data items you indirectly change the item_size.

A quick view of accessing the attributes will show another way of detecting the difference between methods and attributes: methods always have a set of parentheses (e.g. ***feecodes.add()*** and attributes do not e.g. ***feecodes.item_size***).

USING THE HASH OBJECT

The remainder of this paper will demonstrate some uses of the hash object through the use of code snippet examples. All the code examples plus others are available to the workshop attendees; details will be provided at SAS Global Forum.

Earlier we reviewed the basic steps to creating and loading data into a hash object

1. DECLARE the object:
2. DEFINE the hash key.
3. DEFINE the data variables.
4. Complete the definitions
5. Load data into a hash object

In discussing the steps in defining the key and data elements it was pointed out that it is necessary to let the DATA step know the attributes of the key and data elements; we did this with a LENGTH statement. Although this works there is a potential problem. What happens if the attributes of one of the data values changes; for example, feecode changes from a four byte character to a six byte character? If you do not change your programme it will fail. Since the data elements will be read from a SAS dataset, we can use the SAS dataset to automatically provide the metadata we need:

```
     if _n_ = 0 then set data.feecodes;
```

This removes a potential maintenance issue in addition to cutting down on the amount of typing we need to do. Since _N_ will never equal zero (0), the statement is not executed and no records from the dataset will be read; however, the SAS compiler will read the metadata (in particular the column names and attributes) from the dataset and add the columns to the program data vector (PDV). Now if feecode changes from four bytes to six our programme automatically knows; one maintenance issue resolved before it happens.

We can further reduce the size of our programme by making a change to the DECLARE statement. In addition to naming and instantiating the hash object, we can also specify the data source:

```
     DECLARE hash feecodes(DATASET:'data.feecodes') ;
```

When the ***defineDone()*** method is invoked the hash object will be populated with records from the DATA.FEECODES table. The arguments shown here, and as we will see in other methods, are of the form ***TAG: value***. In this example ***DATASET:*** is the tag, and ***'data.feecodes'*** is the value. As with the arguments to ***defineKey()/defineData()*** we are providing a string with the name of the dataset. There are other argument tags available. The online documentation has a complete list of the tags, however we will be seeing most of them in the examples.

Put together our simplified programme snippet would look like:

```
     if _n_ = 0 then set data.feecodes;
     DECLARE hash  feecodes(DATASET:'data.feecodes') ;
       rc=feecodes.defineKey('feecode');
       rc=feecodes.defineData('feeAmount');
       rc=feecodes.defineDone();
```

DUPLICATE KEYS

In the original example of loading the hash object with the ***add()*** method we saw an example of checking for errors:

```
         rc = feecodes.add();
         if rc NE 0
```

One of the 'errors' we are like likely to encounter is a duplicate key value. By default the hash object keeps just one value of the key; moreover by default it keeps the first value it encounters. If you want to keep the last value of a key you can use the ***replace()*** method:

```
         rc = feecodes.replace();
         if rc NE 0
```

Prior to SAS 9.2, there was no way to have multiple items with the same key; we will see an example later on how to maintain multiple items with the same key. Logically one would expect a lookup table to have unique key values; however, since the hash object can be used for more than a simple lookup table it needs a mechanism to deal with duplicates. If you prefer to load your hash object by specifying a ***DATASET:*** tag, you can also specify if you want the first or last occurrence kept. By default we get the first occurrence. If we want the last occurrence we can use the ***DUPLICATE:*** tag:

```
      DECLARE hash  feecodes(DATASET:'data.feecodes', DUPLICATE:’Y’) ;
      * an alternate and perhaps a more ‘accurate’ specification;
      DECLARE hash  feecodes(DATASET:'data.feecodes', DUPLICATE:’replace’) ;
```

As you gain more experience with the hash object methods you can develop better strategies to deal with duplicates if the need arises.

FINDING HASH ENTRIES

Up to this point we have been focusing on creating and loading the hash object; however, we create hash objects in order to use them. One of the most common uses of the hash is as a lookup. To find an entry in the hash table we simply invoke its ***find()*** method

```
      do while (not done);
         set data.transactions end=done;  1
         rc = feecodes.find();  2
         if rc = 0     3
            then output;
            else
                do;
                   put "Feecode Not Found " feecode= rc=;
                end;
      end;
```

In this snippet:

1. We read a row from the TRANSACTIONS table. One of the columns is ***feecode***:
2. We invoke the ***find()*** method of the ***feecodes*** hash.

3. If the feecode from TRANSACTIONS was found in the hash, the return code was set to zero (0) and all the data values are returned to the DATA step for processing.

Notice that the ***find()*** method has no argument tags; it is using the current value of the feecode column since we told the hash object that the feecode column was the key. If the name of the variable you wish to lookup is not the same as the name of the key item in the hash (for example you may have two codes on input, one called feecode and one called altCode) you can use the ***KEY:*** argument tag

```
         rc = feecodes.find(key:altCode);
```

If you only want to know if the value exists in the hash object but do not want to retrieve the data values you can use the ***check()*** method:

```
         rc = feecodes.check();
```

If the entry is found the return code is set to zero but none of the data values are returned; you know the key exists in the table but you have not retrieved the data items. Looking back at the questions raised above regarding duplicate keys you can see that a strategy of using a mix of ***check()***, ***find()***, ***add()***, and ***replace()*** will allow us to properly handle duplicate key values.

LARGE HASH TABLES

A hash table size is limited by the amount of memory available to your SAS session. With the move to 64 bit operating systems and large memory capacity computers we can be potentially looking at very large hash tables. In order to improve performance of the hash table there is a argument tag we can use when declaring the hash object that can help with performance - the ***HASHEXP:*** tag.

```
      DECLARE hash  patients(DATASET:'data.patients', HASHEXP: 20);
```

Essentially ***hashexp*** is an indicator of how to allocate memory and distribute the hash items in memory. For a complete description of ***hashexp*** and the hash memory model see Dorfman et al 2008. As a simple rule, the larger your hash table the larger the value ***hashexp*** should take. The maximum size for ***hashexp*** is 20. Since both the size of each hash item as well as the number of records comes into play in terms of memory management you should try various values of ***hashexp*** to find the best performance for your application.

THE HASH ITERATOR

We have looked at how to create and load a hash object. Following that we looked at how to find and retrieve items in the hash object. If you need to do nothing more than these activities then you do not need a hash iterator. However, you will soon find many uses for memory resident tables beyond searching using ***find()*** or ***check().*** One use will be the ability to traverse your memory table from start to end, from end to start or both.

A hash iterator is associated with a specific hash object and operates only on that hash object. Before you declare your interator you must declare your hash object:

```
DECLARE hash  feecodes(DATASET:"data.feecodes, ORDERED:’A’);
       rc=feecodes.defineKey('feecode');
       rc=feecodes.defineData( 'feecode', 'feeAmount');
* declare an ITERATOR (hiter) on feecodes;
DECLARE hiter hi_feecodes('feecodes');
       rc=feecodes.defineDone();
```

In this snippet we are declaring a hash iterator (***hiter***) called ***hi_feecodes***. The argument in the declare is the name of the hash object. Once again the name is passed in as a string. Note the ***ORDERED:*** tag in the declare statement for the hash object; this is telling SAS to sort the items in the hash in ascending order. Although it is not necessary to order the items in a hash object in order to use the iterator, I find if I plan to traverse the hash object using an iterator I want the traverse in key order. The options available for the ***ordered:*** tag are

- 'A' (or 'ASCENDING') to return items in ascending order of the key
- 'Y' (or 'YES') same as 'A' ('ASCENDING')
- 'D' (or 'DESCENDING') to return the items in descending key order
- 'N' (or 'NO') items are returned in an undefined order. This is the default.

Note that the options are case insensitive.

TRAVERSING THE HASH OBJECT USING THE ITERATOR

The following code demonstrates creating and using the iterator object.

```
DATA _null_;
     if _n_ = 0 then set data.feecodes (keep=feecode feeAmount);
     DECLARE hash feecodes(DATASET:"data.feecodes (keep= feecode feeAmount",
ORDERED:'A' );   1
       rc=feecodes.defineKey('feecode');
      rc=feecodes.defineData( 'feecode', 'feeAmount');
     DECLARE hiter hi_feecodes('feecodes');  2
      rc=feecodes.defineDone();
     * start the beginning;
     put / "----- Traversing using the iterator first time";
     rc = hi_feecodes.first();  3
     * read over all items - keep track of the number of items in the hash;
     row = 1;
     do while (rc = 0) ;
        put row=  feecode= feeAmount= ;
        rc = hi_feecodes.next() ;  3
        row = row + 1;
     end ;
     * start the the end;
     put / "----- Traversing using the iterator second time";
     rc = hi_feecodes.last();  4
     row = 1;
     do while (rc = 0) ;
        put row=  feecode= feeAmount= ;
        rc = hi_feecodes.prev() ;  4
        row = row + 1;
     end ;
     STOP;
RUN ;
```

From the above code we see:

1. The use of the ***ORDERED:*** tag when declaring the has object. Also note the use of dataset options – in this case ***keep=***. Another useful dataset option is ***where=*** to only load a subset of the data into the hash object.
2. The Iterator object (***hi_feecodes***) is declared for the hash object feecodes.
3. The use of ***hi_feecodes.first() and hi_feecodes.next()*** to traverse the hash object in ascending order.
4. The use of ***hi_feecodes.last() and hi_feecodes,prev()*** to traverse the hash object in descending order.

How many times have you processed a file in one direction, then sorted and processed in another order? For example I often have to do this when doing cross record consistency checks where each row has a from/to date and I need to ensure that there is no overlap between the from date in the current row and the to date in the previous row. After processing the table once I have to process again to look for other inconsistencies, often in the opposite order; employment records or address records are examples of this type of processing. By using an iterator object it is possible to process the table 'up and down' as often as necessary yet only read the dataset one time.

SAVING THE HASH OBJECT

Earlier it was stated that the hash object disappears when the DATA step ends. Although the hash object disappears it is possible to save the contents of the hash object to a SAS dataset. There can be some interesting applications of this ability, two of which will be demonstrated:

1. Read in an unsorted dataset and write a sorted dataset
2. Create an undefined number of SAS datasets from an input data set. Each output dataset will contain a subset of the data and be named according to the key value.

To save the items in a hash object we simply use the ***output()*** method of the object; this will write all of the columns in the ***defineData()*** to a dataset. When you plan to write the contents of a hash object to a dataset make sure you include the key columns as part of the ***defineData()*** list. The following example demonstrates the ability to read in a dataset and write it out in a different order. The DOCTORS dataset is ordered by the postalcode column, however we will create a hash object keyed on the doctor ID (DID), then create a new dataset ordered by DID:

```
DATA transDoc;
     if _n_ = 0
     then
       do;
         set data.feecodes (keep=feecode feeamount);
         set data.doctors  ;
       end;
     DECLARE hash  feecodes(DATASET:'data.feecodes') ;
       rc=feecodes.defineKey('feecode');
       rc=feecodes.defineData('feeAmount');
       rc=feecodes.defineDone();

     DECLARE hash  doctors(DATASET:'data.doctors', ORDERED: ‘a’) ;  1
       rc=doctors.defineKey('did');
       rc=doctors.defineData(ALL:'YES');  2
       rc=doctors.defineDone();
     * now we need to read each record in the transactions table;
     do while (not done);
        set data.transactions end=done;
        rcFC = feecodes.find();
        rcDOC = doctors.find();    3
        doctorDOB = dob;
        output transDoc;    4
     end;
     * write out the sorted doctors table;
     rc = doctors.output(DATASET:'sortedDocs');    5
     STOP;
RUN ;
```

From the code we can see:

1. The use of the ***ordered:*** tag when declaring the hash object
2. The use of the ***ALL:*** tag in the ***defineData()*** method. This is a shortcut to include all the columns in the dataset as data items in the hash object.
3. Using the hash object as part of the processing
4. Saving the transaction data
5. Using the ***output()*** method to save the contents of the hash object. The ***DATASET:*** tag has the name of the output dataset. Notice that this dataset is not named on the DATA statement.

We can extend the use of the ***output()*** method to create multiple datasets. In this example we will create one dataset for each section in the FEECODES table:

```
data _null_ ;    1
     if 0 then set data.feecodes;
     length rec 8.;
     DECLARE hash feecodes (ordered: 'a');
       rc = feecodes.definekey ('rec');    2
       rc = feecodes.definedata('section', 'subsection', 'feecode', 'feeAmount');    3
       rc = feecodes.definedone() ;
     do until(done);
        do rec = 1 by 1 until ( last.section );    4
           set data.feecodes end=done;
           by section;
           feecodes.add() ;    5
        end ;
        feecodes.output (dataset: 'section' || section) ;    6
        feecodes.clear();    7
     end;
run
```

Some of the points to note from this code:

1. We are using a DATA _NULL_. Normally a DATA _NULL_ does not create output datasets.
2. We are using a 'dummy' value for the key.
3. We are specifying all of the columns as data items. We could have used the ***ALL:*** tag
4. We are reading the dataset section by section, the (last.section) will end each loop. Note also the use of the 'dummy' key as the loop counter. This ensures each key value is unique.
5. The data items are added to the hash object.
6. After all the records for a section are read into the hash object, an output dataset is created with names like sectionA (for section A), sectionB etc..
7. The ***clear()*** method removes all the entries from the hash object. The next iteration of the DO loop will add a new set of items into the hash object.

DUPLICATE KEYS REVISITED

In SAS 9.1 the hash object could not support duplicate key values. In SAS 9.2 this restriction was lifted. To allow for duplicate key values a new argument tag (***MULTIDATA:***) was added to the hash object declaration; in addition some new methods were added to allow find the duplicates. The following code demonstrates these:

```
* add a DUPLICATE record to feecodes table;
data feecodes;
     set data.feecodes end=done;
     output;
     if done
     then
       do;
          feecode = 'A007'; section = 'Z'; subsection = 99; feeamount = -1;
          output;                1
       end;
run;

* lookup feecode A007 -- check from more than one value;
DATA _null_;
     if _n_ = 0 then set feecodes (keep=feecode feeAmount);
     DECLARE hash  feecodes(DATASET:'feecodes', ORDERED:'A', MULTIDATA:"Y");   2
       rc=feecodes.defineKey('feecode');
       rc=feecodes.defineData( 'feecode', 'feeAmount');
       rc=feecodes.defineDone();

     * just look for the duplicate record;
     feecode = 'A007';
     rc = feecodes.find();                3
     put rc= feecode= feeAmount=;
     anotherCode = .;
     rc = feecodes.has_next(RESULT: anotherCode);   4
     do while (anotherCode NE 0);
        rc = feecodes.find_next();  5
        put rc= feecode= feeAmount= anotherCode=;
        rc = feecodes.has_next(RESULT: anotherCode);
     end;
     * we are doing nothing else, so stop;
     STOP;
RUN ;
```

Some of the points to note from this code:

1. We are forcing a duplicate value in the FEECODES dataset.
2. We are using the ***MULTIDATA**:* tag to tell the hash object to expect duplicate key values.
3. We are searching for the duplicate key value.
4. The ***has_next()*** method is used to check if there is another item with the same key. The ***has_next()*** method has one argument, ***RESULT:*** which will be used to tell us if there is another item with the same key. A non-zero value returned in the argument tells us there is another item with the same key.
5. The ***find_nextl()*** method will retrieve the next item for the duplicate key

CONCLUSION

This paper was intended to show how to take advantage of the SAS hash object; it was not intended to be an in depth study of the hash object. For an in-depth study of the hash object refer to any of the papers by Dorfman.

The hash object offers a number of useful and powerful constructs to help us better manage and process our data. However many of us have been slow to incorporate this object into our daily work – sometimes because we did not see the value in it, sometimes because the new syntax was daunting. The syntax is different. The terminology is different. Once you get over these minor differences the hash object is like the rest of the SAS programming language –easy to use and powerful.

This paper is meant to be used as background for a Hands-On Workshop. The workshop exercises go into more detail than is covered here.

REFERENCES

Dorfman, Paul. 2001. "Table Look-Up by Direct Addressing: Key-Indexing -- Bitmapping -- Hashing" *Proceedings of the Twenty-sixth SAS Users Group International Meeting*

Dorfman, Paul, Shajenko, Lessia and :Vyverman, Koen. 2008. "Hash Crash and Beyond", *Proceedings of the SAS Global Forum 2008 Conference*

Liu, Ying. 2007. "SAS Formats: More Than Just Another Pretty Face", *Proceedings of Fifteenth Annual Southeast SAS Users Group Conference*,

Loren, Judy. 2008. "*How* Do I Love Hash Tables? Let Me Count The Ways*!*", *Proceedings of the SAS Global Forum 2008 Conference*

Secosky, Jason and Bloom, Janet. "Getting Started with the DATA Step Hash Object", http://support.sas.com/rnd/base/datastep/dot/iterator-getting-started.pdf

ACKNOWLEDGMENTS

I would like to thank:

- Paul Dorfman for all the detailed explanation of hash objects in general and the SAS Hash Object in particular
- All of my clients, but especially the Department of Economics at the Ontario Medical Association, for giving me the opportunity to learn and develop.

CONTACT INFORMATION

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

Name:	Peter Eberhardt
Enterprise:	Fernwood Consulting Group Inc.
Address:	288 Laird Dr
City, State ZIP:	Toronto, ON M4G 3X5 Canada
E-mail:	peter@fernwood.ca
Web:	www.fernwood.ca

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX A: THE DATA TABLES

FEECODES TABLE:

Variables in Creation Order						
#	**Variable**	**Type**	**Len**	**Format**	**Informat**	**Label**
1	section	Char	1			
2	subSection	Num	8	11.	11.	Sub Section
3	feecode	Char	4			
4	feeAmount	Num	8	COMMA8.2		Fee Amount

PATIENTS TABLE:

Variables in Creation Order						
#	**Variable**	**Type**	**Len**	**Format**	**Informat**	**Label**
1	PID	Num	8			
2	studyID	Char	10			
3	postcode	Char	10	$10.	$10.	postcode
4	dob	Num	8	YYMMDD10.	YYMMDD10.	dob
5	sex	Char	1	$1.	$1.	sex

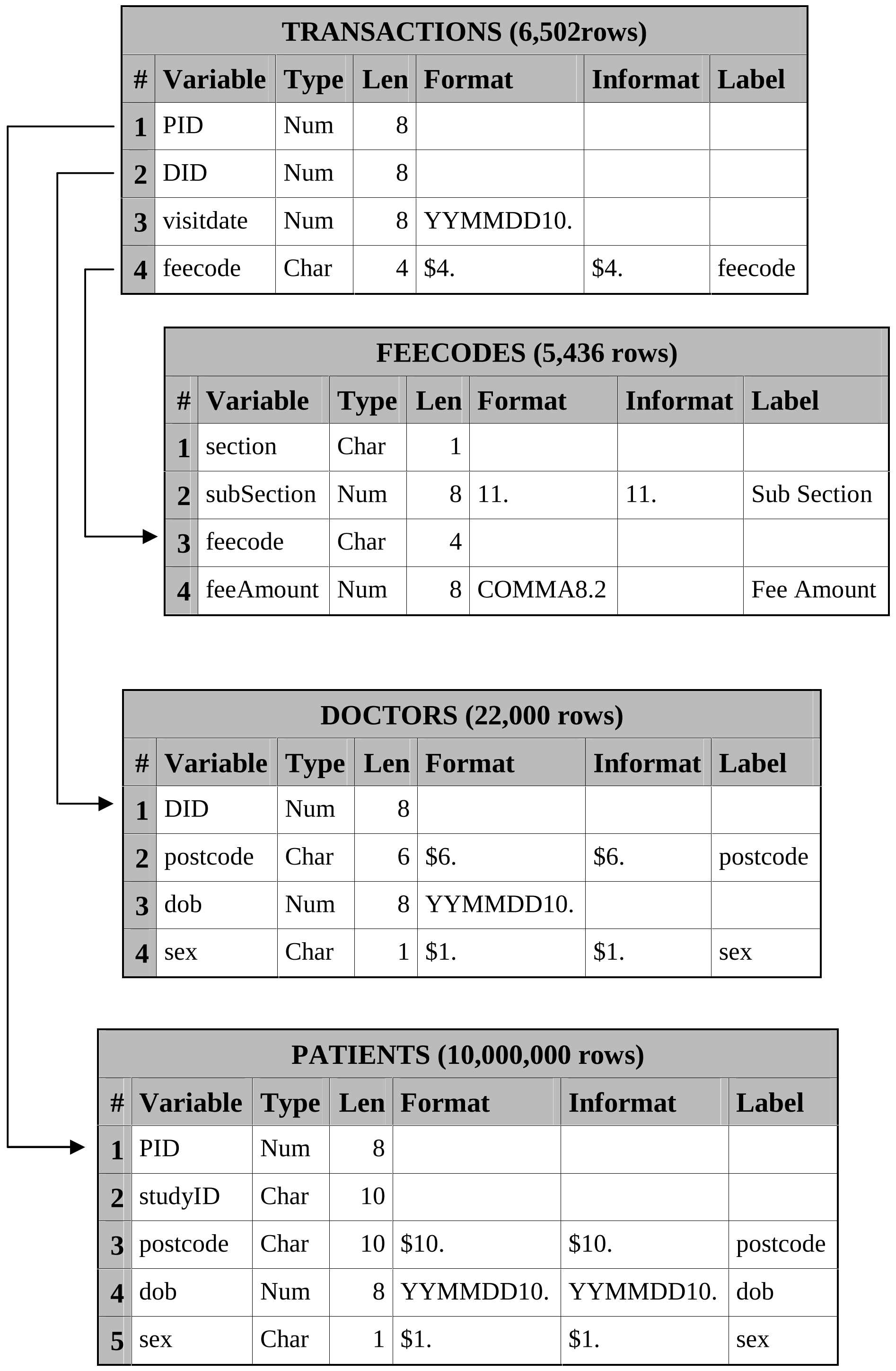
DOCTORS TABLE:

Variables in Creation Order						
#	**Variable**	**Type**	**Len**	**Format**	**Informat**	**Label**
1	DID	Num	8			
2	postcode	Char	6	$6.	$6.	postcode
3	dob	Num	8	YYMMDD10.		
4	sex	Char	1	$1.	$1.	sex

TRANSACTIONS TABLE:

Variables in Creation Order						
#	**Variable**	**Type**	**Len**	**Format**	**Informat**	**Label**
1	PID	Num	8			
2	DID	Num	8			
3	visitdate	Num	8	YYMMDD10.		
4	feecode	Char	4	$4.	$4.	feecode

RELATIONSHIPS:



TRANSACTIONS (6,502rows)

#	Variable	Type	Len	Format	Informat	Label
1	PID	Num	8			
2	DID	Num	8			
3	visitdate	Num	8	YYMMDD10.		
4	feecode	Char	4	$4.	$4.	feecode

FEECODES (5,436 rows)

#	Variable	Type	Len	Format	Informat	Label
1	section	Char	1			
2	subSection	Num	8	11.	11.	Sub Section
3	feecode	Char	4			
4	feeAmount	Num	8	COMMA8.2		Fee Amount

DOCTORS (22,000 rows)

#	Variable	Type	Len	Format	Informat	Label
1	DID	Num	8			
2	postcode	Char	6	$6.	$6.	postcode
3	dob	Num	8	YYMMDD10.		
4	sex	Char	1	$1.	$1.	sex

PATIENTS (10,000,000 rows)

#	Variable	Type	Len	Format	Informat	Label
1	PID	Num	8			
2	studyID	Char	10			
3	postcode	Char	10	$10.	$10.	postcode
4	dob	Num	8	YYMMDD10.	YYMMDD10.	dob
5	sex	Char	1	$1.	$1.	sex