Paper HOW-05

Two Guys on Hash

Paul M. Dorfman, *Independent Consultant, Jacksonville, FL*
Peter Eberhardt, Fernwood Consulting Group Inc. Toronto, ON, Canada

ABSTRACT

The SAS® hash object is no longer new, yet its use is still not widespread. Where it is used, often the strengths and capabilities of the hash object are underutilized. In this workshop we will quickly step through some introductory steps to ensure everyone is 'on the same page'; however, it does assume workshop attendees have some knowledge of the hash object - if not from practical experience, at least from attendance at an introductory workshop.

Once we lay some introductory groundwork we will work through some more interesting and challenging examples of the hash object in action.

Take a deep breath (but don't inhale) as we start our journey into the world of hash.

INTRODUCTION

What is the SAS hash object? It is a high-performance look-up table residing completely in the DATA step memory. The hash object is implemented via Data Step Component Interface (DSCI), meaning that it is not a part of the DATA step proper. Rather, you can picture it as a black-box device you can manipulate from inside the DATA step to ask it for lightning-quick data storage and retrieval services. For example, you can:

- give the hash object a key and tell it to ***add*** it along with associated data in the table or ***replace*** the data if the key is already there
- ***check*** whether the key is in the table, and if yes, find (retrieve) its data or remove the key with its data from the table
- ask the object to write its content to an ***output*** SAS data file independently of other DATA step actions
- make the hash object accumulate simple summary statistics for each key in the table
- instruct the hash object to have the table entries ***ordered*** by the key
- use a hash iterator to access the ***next*** key in the table starting from the ***first*** or ***last*** key or from any key at all, i.e. to enumerate the entries
- do all of the above with a *hash object containing other hash objects* rather than mere data.

"*High-performance*" means not only that all hash object operations are performed very rapidly, but also that the speed of Add, Check, Find, and Remove operations (called methods) does not depend on the number of keys in the table. In other words, the time necessary to find if a key is in the table is virtually the same for the table containing 100 or 1,000,000 keys.

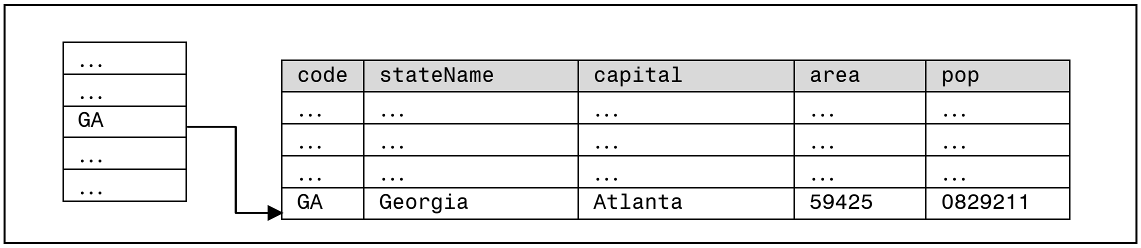
However, before being able to do all these wonderful things, you have to learn a few things:

- The hash object does not understand the DATA step language, and must be instead addressed using the so-called object-dot syntax. Luckily, to the extent of the hash object scope, it is very simple.
- You have to understand that all hash object operations – including the creation and deletion of the object itself –are performed at the DATA step run time, rather than compile time.
- You have to learn how to prepare the DATA step environment for working with the hash object, for without such preparation the object cannot even be created, let alone serve a useful purpose.

TABLE LOOK-UPS

Searching and table look-ups are ubiquitous in SAS programmes. Many SAS programmers may not think of themselves as performing searches or table look-ups in their programmes, however, if they have ever done a DATA step merge or a SQL join they have indeed performed searches and look-ups. In general a table look-up 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 look-up 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

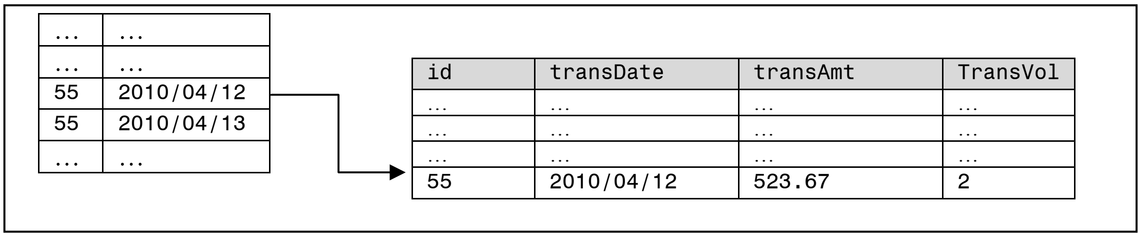


…
…
GA
…
…

code	stateName	capital	area	pop
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
GA	Georgia	Atlanta	59425	0829211

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 look-up 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 look-ups, one of the most common being the use of SAS formats and the PUT() function; this has been the workhorse in-memory look-up method of choice by SAS programmers for years. In addition, some complex but efficient in memory look-ups have been devised. Starting in SAS v9 a new, flexible and efficient method to do look-ups 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, we will introduce the data that will be used in some of the examples.

SAMPLE DATA

The introductory section of 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 million 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.

As we progress to more advanced concepts we will be creating data as part of the exercise; since the purpose of the more advanced exercises is to demonstrate the concepts the data are less critical. On the other hand, using consistent datasets makes learning the introductory concepts easier.

THE SAS HASH OBJECT

Unlike a DATA step merge and a SQL join which are disk based look-ups, that is the look up table is a disk file which SAS will access repeatedly in the look-up 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 look-ups 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 look-up tables into hash objects and sequentially process the larger table that has the look-up 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 look-up keys can be processed in its existing order. Removing the need to sort before (or during as PROC SQL may do) look-up 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 look-up 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 look-up 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 we will break CREATING and USING into five 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***).

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 look-up table to have unique key values; however, since the hash object can be used for more than a simple look-up 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 look-up. To find an entry in the hash table we simply invoke its ***find()*** method

```
     do while (not done);  1
```

```
         set data.transactions end=done;
         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 look-up 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.

HITER: HASH ITERATOR OBJECT

During both hash table load and look-up, the sole question we need to answer is whether the particular search key is in the table or not. The FIND and CHECK hash methods give the answer without any need for us to know what other keys may or may not be stored in the table. However, in a variety of situations we do need to know the keys and data currently resident in the table. How do we do that?

ENUMERATING HASH ENTRIES

In hand-coded schemes, it is simple since we had full access to the guts of the table. However, hash object entries are not accessible as directly as array entries. To make them accessible, SAS provides the *hash iterator* object, *hiter*, which makes the hash table entries available in the form of a serial list. Let us consider a simple program that should make it all clear.

```
data sample ;
    input k sat ;
    cards ;
185 01
971 02
400 03
260 04
922 05
970 06
543 07
532 08
050 09
067 10
;
run ;
data _null_ ;
    if 0 then set sample ;
    dcl hash hh ( dataset: ‘sample’, hashexp: 8, ordered: ‘A’) ;
    dcl hiter hi ( ‘hh’ ) ;
    hh.DefineKey ( ‘k’ ) ;
    hh.DefineData ( ‘sat’ , ‘k’ ) ;
    hh.DefineDone () ;
    do rc = hi.first () by 0 while ( rc = 0 ) ;
       put k = z3. +1 sat = z2. ;
       rc = hi.next () ;
    end;
    do rc = hi.last () by 0 while ( rc = 0 ) ;
       put k = z3. +1 sat = z2. ;
       rc = hi.prev () ;
    end ;
    stop ;
run
```

We see that now the hash table is instantiated with the ORDERED parameter set to ‘a’, which stands for ‘ascending’. When ‘a’ is specified, the table is automatically loaded in the ascending key order. It would be better to summarize the rest of the meaningful values for the ORDERED parameter in a set of rules:

- ‘a’ , ‘ascending’ = ascending
- ‘y’ = ascending
- ‘d’ , ‘descending’ = descending
- ‘n’ = internal hash order (i.e. no order at all, and the original key order is NOT followed)
- any other character literal different from above = same as ‘n’
- parameter not coded at all = the same as ‘n’ by default
- character expression resolving to the same as the above literals = same effect as the literals
- numeric literal or expression = DSCI execution time object failure because of type mismatch
-

Note that the hash object symbol name must be passed to the iterator object as a character string, either hard-coded as above or as a character expression resolving to the symbol name of a declared hash object, in this case, “HH”. After the iterator HI has been successfully *instantiated*, it can be used to fetch entries from the hash table in the key order defined by the rules given above.

To retrieve hash table entries in an ascending order, we must first point to the entry with the smallest key. This is done by the method FIRST():

```
rc = hi.first () ;
```

where HI is the name we have assigned to the iterator. A successful call to FIRST fetches the smallest key into the host variable K and the corresponding satellite - into the host variable SAT. Once this is done, each call to the NEXT method will fetch the hash entry with the next key in ascending order. When no keys are left, the NEXT method returns RC > 0, and the loop terminates. Thus, the first loop will print in the log:

```
k=050 sat=09
k=067 sat=10
k=185 sat=01
k=260 sat=04
k=400 sat=03
k=532 sat=08
k=543 sat=07
k=922 sat=05
k=970 sat=06
k=971 sat=02
```

Inversely, the second loop retrieves table entries in descending order by starting off with the call to the LAST() method fetching the entry with the largest key. Each subsequent call to the method PREV extracts an entry with the next smaller key until there are no more keys to fetch, at which point PREV returns RC > 0, and the loop terminates.

Therefore, the loop prints:

```
k=971 sat=02
k=970 sat=06
k=922 sat=05
k=543 sat=07
k=532 sat=08
k=400 sat=03
k=260 sat=04
k=185 sat=01
```

```
k=067 sat=10
k=050 sat=09
```

An alert reader might be curious *why the key variable had to be also supplied to the DefineData() method*? After all, each time the DO-loop iterates the iterator points to a new key and fetches a new key entry. The problem is that the host key variable K is updated only once, as a result of the HI.FIRST() or HI.LAST() method call. Calls to PREV() and NEXT() methods do not update the host key variable. However, a satellite hash variable does! So, if in the step above, it had not been passed to the DefineData() method as an additional argument, only the key values 050 and 971 would have been printed.

The concept behind such behavior is that only data entries in the table have the legitimate right to be "projected" onto its Data step host variables, whilst the keys do not. It means that if you need the ability to retrieve a key from a hash table, you need to define it in the data portion of the table as well.

ARRAY SORTING VIA A HASH ITERATOR

The ability of a hash iterator to rapidly retrieve hash table entries in order is an extremely powerful feature, which will surely find a lot of use in Data step programming. The first iterator programming application that springs to mind immediately is using its key ordering capabilities to sort another object. The easiest and most apparent prey is a SAS array. The idea of sorting an array using the hash iterator is very simple:

1. Declare an ordered hash table keyed by a variable of the data type and length same as those of the array.
2. Declare a hash iterator.
3. Assign array items one at a time to the key and insert the key in the table.
4. Use the iterator to retrieve the keys one by one from the table and repopulate the array, now in order.

A minor problem with this most brilliant plan, though, is that since the hash object table cannot hold duplicate keys, a certain provision ought to be made in the case the when the array contains duplicate elements. The simplest way to account for duplicate array items is to enumerate the array as it is used to load the table and use the unique enumerating variable as an additional key into the table. In the case the duplicate elements need to be eliminated from the array, the enumerating variable can be merely set to a constant, which below is chosen to be a zero:

```
data _null_ ;
    array a [-100000 : 100000] _temporary_ ;
** allocate sample array with random numbers, about 10 percent duplicate ;
    do _q = lbound (a) to hbound (a) ;
       a [_q] = ceil (ranuni (1) * 2000000) ;
    end ;
** set sort parameters ;
    seq = ‘A’ ; * A = ascending, D = descending ;
    nodupkey = 0 ; * 0 = duplicates allowed, 1 = duplicates not allowed ;
    dcl hash _hh (hashexp: 0, ordered: seq) ;
    dcl hiter _hi (‘_hh’ ) ;
    _hh.definekey (‘_k’, ‘_n’ ) ; * _n - extra enumerating key ;
    _hh.definedata (‘_k’ ) ; * _k automatically assumes array data type ;
    _hh.definedone ( ) ;
** load composite (_k _n) key on the table ;
** if duplicates to be retained, set 0 <- _n ;
    do _j = lbound (a) to hbound (a) ;
```

```
       _n = _j * ^ nodupkey ;
       _k = a [_j] ;
       _hh.replace() ;
    end ;
** use iterator HI to reload array from HH table, now in order ;
    _n = lbound (a) - 1 ;
    do _rc = _hi.first() by 0 while ( _rc = 0 ) ;
       _n = _n + 1 ;
       a [_n] = _k ;
       _rc = _hi.next() ;
    end ;
    _q = _n ;
** fill array tail with missing values if duplicates are delete ;
    do _n = _q + 1 to hbound (a) ;
       a [_n] = . ;
    end ;
    drop _: ; * drop auxiliary variables ;
    ** check if array is now sorted ;
    sorted = 1 ;
    do _n = lbound (a) + 1 to _q while ( sorted ) ;
       if a [_n - 1] > a [_n] then sorted = 0 ;
    end ;
    put sorted = ;
run
```

Note that choosing HASHEXP:16 above would be more beneficial performance-wise. However, HASHEXP=0 was chosen to make an important point a propos:

USING THE HASH ITERATOR TO SORT AN ARRAY

Since it means 2**0=1, i.e. a single bucket, *we have created a stand-alone AVL(Adelson-Volsky & Landis) binarytree in a Data step*, let it grow dynamically as it was being populated with keys and satellites, and then traversed it toeject the data in a predetermined key order. So, do not let anyone tell you that a binary search tree cannot becreated, dynamically grown and shrunk, and deleted, as necessary at the Data step run time. It can, and with very little programmatic effort, too!

Just to give an idea about this hash table performance in some absolute figures, this entire step runs in about 1.15 seconds on a desktop 933 MHz computer under XP Pro. The time is pretty deceiving, since 85 percent of it is spent inserting the data in the tree. The process of sorting 200,001 entries itself takes only scant 0.078 seconds either direction. Increasing HASHEXP to 16 reduces the table insertion time by about 0.3 seconds, while the time of dumping the table in order remains the same.

One may ask why bother to program array sorting, even if the program is as simple and transparent as above, if in SAS9, CALL SORTC() and CALL SORTN() routines seemingly offer the same functionality. In actuality, though, they are designed to sort variable lists, which may or may not be organized into arrays. As such, it does not allow the provision of sorting a temporary array by a single array reference; all of its elements must be explicitly listed. It is not impossible to assemble the references one by one using a macro or another technique, but with 200001 elements to sort, it takes over 30 seconds just to compile the step. Besides, such method does not allow accounting for duplicates, and for those who care about such things, the kludge looks aesthetically maladroit.

DATA STEP COMPONENT INTERFACE (DSCI)

Now that we have a taste of the new Data step hash objects and some cool programming tricks they can be used to pull, let us consider it from a little bit more general viewpoint. In Version 9, the hash table (associative array) introduces the first *component object* accessible via a rather novel thing called DATA Step Component Interface (DSCI). A component object is an abstract data entity consisting of two distinct characteristics: *Attributes and methods*. *Attributes* are data that the object can contain, and *methods* are operations the object can perform on its data.

From the programming standpoint, an object is a black box with known properties, much like a SAS procedure. However, a SAS procedure, such as SORT or FORMAT, cannot be called from a Data step at run-time, while an object accessible through DSCI - can. A Data step programmer who wants an object to perform some operation on its data, does not have to program it procedurally, but only to call an appropriate method.

THE OBJECT

In our case, the object is a hash table. Generally speaking, as an abstract data entity, *a hash table is an object providing for the insertion and retrieval of its keyed data entries in O(1), i.e. constant, time*. Properly built directaddressed tables satisfy this definition *in the strict sense*. We will see that the hash object table satisfies it *in the practical sense*. The attributes of the hash table object are keyed entries comprising its key(s) and maybe also satellites.

Before any hash table object methods can be called (operations on the hash entries performed), the object must be declared. In other words, the hash table must *be instantiated* with the DECLARE (DCL) statement, as we have seen above.

METHODS

The hash table methods are used to tell a hash object which functions to perform and how. New methods are being added in almost each new SAS release and version. As they currently stand, the methods are as follows:

- DefineKey() . Define a set of hash keys.
- DefineData() . Define a set of hash table satellites. This method call can be omitted without harmful consequences if there is no need for non-key data in the table. Although a dummy call can still be issued, it is not required.
- DefineDone() . Tell SAS the definitions are done. If the DATASET argument is passed to the table’s definition, load the table from the data set.
- ADD() . Insert the key and satellites if the key is not yet in the table (ignore duplicate keys).
- REPLACE() . If the key is not in the table, insert the key and its satellites, otherwise overwrite the satellites in the table for this key with new ones.
- REMOVE() . Delete the entire entry from the table, including the key and the data.
- FIND() . Search for the key. If it is found, extract the satellite(s) from the table and update the host Data step variables.
- REF() (new in 9.2). Consolidate FIND and ADD methods into a single method call. Particularly useful in summarizations where it provides for a simpler programming logic compared to calling FIND and ADD separately.
- CHECK() . Search for the key. If it is found, just return RC=0, and do nothing more. Note that calling this method does not overwrite the host variables.
- OUTPUT() . Dump the entire current contents of the table into a one or more SAS data set. Note that for the key(s) to be dumped, they must be defined using the DefineData method. If the table has been loaded in order, it will be dumped also in order. More information about the method will be provided later on.
- CLEAR() . Remove all items from a hash table without deleting the hash object instance. In many cases it is desirable to empty a hash table without deleting the table itself - the latter takes time. For instance, in Examples 6 and 9 below, program control redefined the entire hash instance from scratch when it passes through its declaration before each BY-group. For

better performance, declaration could be executed only once, and the table - cleaned up by calling the CLEAR method instead.

- EQUAL() (new in 9.2). Determine if two hash objects are equal.
- SETCUR() (new in 9.2). Specify a key from which iterator can start scrolling the table. It is a substantial improvement over 9.1 where the iterator could start only either from the beginning or end of the table.
- FIRST() . Using an iterator, fetch the item stored in a hash table *first*. If the table is ordered, the lowest-value item will be fetched. If none of the items have been fetched yet, the call is similar to NEXT().
- LAST() . Using an iterator, fetch the item stored in a hash table *last*. If the table is ordered, the highest-value item will be fetched. If none of the items have been fetched yet, the call is similar to PREV().
- NEXT() . Using an iterator, fetch the item stored *right after* the item fetched in the previous call to FIRST() or NEXT() from the same table.
- PREV() . Using an iterator, fetch the item stored *right before* the item fetched in the previous call to LAST() or PREV() from the same table.
- SUM() (new in 9.2). If SUMINC argument tag has been used in a hash table declaration, use SUM() method to retrieve summary counts for each (distinct) key stored in the table.

ATTRIBUTES

Just as a SAS data file has a descriptor containing items which can be retrieved without reading any data from the file (most notably, the number of observations), a hash object has attributes which can be retrieved without calling a single method. Currently, there are 2 attributes:

1. ITEM_SIZE. Returns a size of a hash object item in bytes.
2. NUM_ITEMS. Returns the total number of items stored in a hash object.

Note that the attributes are returned directly into a numeric SAS variable, and their syntax differs from that ofmethods in that parentheses are not used. For example for a hash object HH:

```
item_size = HH.item_size ;
num_items = HH.num_items ;
```

OPERATORS

Currently, there is only one operator:

1. _NEW_. Use it to create an instance of already declared component object.

The _NEW_ operator is extremely useful when there is a need to create more than one instance of a hash object and have the ability to store and manage each of them separately. Example 10 below and the ensuing discussion provide a good deal of detail about the operator using a real-world task.

A PEEK UNDER THE HOOD

We have just seen the tip of the hash iceberg from the outside. An inquiring mind would like to know: What is inside? Not that we really need the gory details of the underlying code, but it is instructive to know on which principles the design of the internal SAS table is based in general. A good driver is always curious what is under the hood. Well, *in general*, hashing is hashing is hashing - which means that it is always a two-staged process:

1. Hashing a key to its bucket
2. resolving collisions within each bucket. Hand-coded hashing cannot rely on the simple straight separate chaining because of the inability to dynamically allocate memory one entry at a time, while reserving it in advance could result in unreasonable waste of memory.

Since the hash and hiter objects are coded in the underlying software, this restriction no longer exists, and so separate chaining is perhaps the most logical way to go. Its concrete implementation, however, has somewhat deviated from the classic scheme of connecting keys within each node into a link list. Instead, each new key hashing to a bucket is inserted into its binary tree. If there were, for simplicity, only 4 buckets, the scheme might roughly look like this:

```
     0         1         2         3
+---------+---------+---------+---------+
|    |    |    |    |    |    |    |    |
|   / \   |   / \   |   / \   |   / \   |
|  /\  /\ |  /\  /\ |  /\  /\ |  /\  /\ |
```

The shrub-like objects inside the buckets are AVL (Adelson-Volsky & Landis) trees. AVL trees are binary trees populated by such a mechanism that on the average guarantees their O(log(N)) search behavior regardless of the distribution of the key values.

The number of hash buckets is controlled by the HASHEXP parameter we have used above. The number of buckets allocated by the hash table constructor is 2**HASHEXP. So, if HASHEXP=8, HZISE=256 buckets will be allocated, or if HASHEXP=16, HSIZE=65536. As of the moment, it is the maximum. Any HASHSIZE specified over 16 is truncated to 16.

Let us assume HASHEXP=16 and try to see how, given a KEY, this structure facilitates hashing. First, a mysterious internal hash function maps the key, whether is it simple or composite, to some bucket. The tree in the bucket is searched for KEY. If it is not there, the key and its satellite data are inserted in the tree. If KEY is there, it is either discarded when ADD is called, or its satellite data are updated when REPLACE is called.

Just how fast all this occurs depends on the speed of search. Suppose that we have N=2**20, i.e. about 1 million keys. With HSIZE=2**16, there will be on the average 2**4 = 16 keys hashing to one bucket. Since N > HSIZE, the table is overloaded, i.e. its load factor is greater than 1. However, binary searching the 16 keys in the AVL tree requires only about 5 keys comparisons. If we had 10 million keys, it would require about 7 comparisons, which practically makes almost no difference.

Thus, the SAS object hash table behaves as O(log(N/HSIZE)). While it is not *exactly* O(1), it can be considered such for all practical intents and purposes, as long as N/HSIZE is not way over 100. Thus, by choosing HASHEXP judiciously, it is thus possible to tweak the hash table performance to some degree and depending on the purpose. For example, if the table is used primarily for high-performance matching, it may be a good idea to specify the maximum HASHEXP=16, even if some buckets end up unused. From our preliminary testing, we have not been able to notice any memory usage penalty exacted by going to the max, all the more that as of this writing, the Data step does not seem to report memory used by an object called through the DSCI. At least, experiments with intentionally large hash tables show that the memory usage reported in the log is definitely much smaller than the hash table must have occupied, although it was evident from performance that the table is completely memory-resident, and the software otherwise has no problem handling it. However, with several thousand keys at hand there is little reason to go over HASHEXP=10, anyway. Also, if the sole purpose of using the table is to eject the data in a key order using a hash iterator, even a single bucket at HASHEXP=0 can do just fine, as we saw earlier with the array sorting example. On the other hand, if there is no need for iterator processing, it is better to leave the table completely iterator-free by not specifying a non-zero ORDERED option. Maintaining an iterator over a hash table obviously requires certain overhead.

DYNAMIC DATA STEP STRUCTURE

The hash table object (together with its hiter sibling) represents the *first ever dynamic Data step structure*, i.e. one capable of acquiring memory and growing at run-time. There are a number of common situations in data processing when the information needed to size a data structure becomes available only at execution time. SAS programmers usually solve such problems by pre-processing data, either i.e. passing through the data more than once, or allocating memory resources for the worst-case scenario. As more programmers become familiar with the possibilities this dynamic structure offers, they will be able to avoid resorting to many old kludges. Remember, a hash object is instantiated during the run-time. If the table is no longer needed, it can be simply wiped out by the DELETE() method:

```
rc = hh.Delete () ;
```

This will eliminate the table from memory for good, but not its iterator! As a separate object related to a hash table, it has to be deleted separately:

```
rc = hi.Delete () ;
```

If at some point of a Data step program there is a need to start building the same table from scratch again, remember that the *compiler must see only a single definition of the same table* by the same token as it must see only a single declaration of the same array (and if the rule is broken, it will issue the same error message, e.g.: “Variable hh already defined”). Also, like in the case of arrays, the full declaration (table and its iterator) must precede any table/iterator references. In other words, this will NOT compile because of the repetitive declaration:

```
20 data _null_ ;
21         length k 8 sat $11 ;
22
23         dcl hash hh ( hashexp: 8, ordered: 1 ) ;
24         dcl hiter hi ( ‘hh’ ) ;
25         hh.DefineKey ( ‘k’ ) ;
26         hh.DefineData ( ‘sat’ ) ;
27         hh.DefineDone () ;
28
29         hh.Delete () ;
30
31         dcl hash hh (hashexp: 8, ordered: 1 ) ;
                    -
                    567
ERROR 567-185: Variable hh already defined.
32         dcl hiter hi ( ‘hh’ ) ;
                     -
                    567
ERROR 567-185: Variable hi already defined.
```

And this will not compile because at the time of the DELETE method call, the compiler has not seen HH yet:

```
39 data _null_ ;
40         length k 8 sat $11 ;
41         link declare ;
42         rc = hh.Delete() ;
                  557
                  68
ERROR 557-185: Variable hh is not an object.
ERROR 68-185: The function HH.DELETE is unknown, or cannot be accessed.
43         link declare ;
44         stop ;
```

```
declare:
        dcl hash hh ( hashexp: 8, ordered: 1 ) ;
        dcl hiter hi ( ‘hh’ ) ;
        hh.DefineKey ( ‘k’ ) ;
        hh.DefineData ( ‘sat’ ) ;
        hh.DefineDone () ;
return ;
stop ;
run ;
```

However, if we do not dupe the compiler and reference the object after it has seen it, it will work as designed:

```
data _null_ ;
       retain k 1 sat ‘sat’ ;
       if 0 then
      do ;
declare:
          dcl hash hh ( hashexp: 8, ordered: 1 ) ;
          dcl hiter hi ( ‘hh’ ) ;
          hh.DefineKey ( ‘k’ ) ;
          hh.DefineData ( ‘sat’ ) ;
          hh.DefineDone () ;
          return ;
       end ;
       link declare ;
             rc = hi.First () ;
             put k= sat= ;
             rc = hh.Delete () ;
             rc = hi.Delete () ;
       link declare ;
             rc = hh.Delete () ;
             rc = hi.Delete () ;
       stop ;
run ;

k=1 sat=sat
```

Of course, the most natural and trouble-free technique of having a table created, processed, cleared, and created from scratch again is to place the entire process in a loop. This way, the declaration is easily placed ahead of any object references, and the compiler sees the declaration just once. In a moment, we will see an example doing exactly that.

DYNAMIC DATA STEP DATA DICTIONARIES

The fact that hashing supports searching (and thus retrieval and update) in constant time makes it ideal for using a hash table as a dynamic Data step data dictionary. Suppose that during DATA step processing, we need to memorize certain key elements and their attributes on the fly, and at different points in the program, answer the following:

1. Has the current key already been used before?

2. If it is new, how to insert it in the table, along with its attribute, in such a way that the question 1 could be answered as fast as possible in the future?
3. Given a key, how to rapidly update its satellite?
4. If the key is no longer needed, how to delete it?

Examples showing how key-indexing can be used for this kind of task are given in [1]. Here we will take an opportunity to show how the hash object can help an unsuspecting programmer. Imagine that we have input data of the following arrangement:

```
data sample ;
   input id transid amt ;
   cards ;
1  11   40
1  11   26
1  12   97
1  13    5
1  13    7
1  14   22
1  14   37
1  14    1
1  15   43
1  15   81
3  11   86
3  11   85
3  11    7
3  12   30
3  12   60
3  12   59
3  12   28
3  13   98
3  13   73
3  13   23
3  14   42
3  14   56
;
run ;
```

The file is grouped by ID and TRANSID. We need to summarize AMT within each TRANSID giving SUM, and for each ID, output 3 transaction IDs with largest SUM. Simple! In other words, for the sample data set, we need to produce the following output:

```
id    transid    sum
 1       15      124
 1       12       97
 1       11       66
 3       13      194
 3       11      178
 3       12      177
```

Usually, this is a 2-step process, either in the foreground or behind the scenes (SQL). Since the hash object table can eject keyed data in a specified order, it can be used to solve the problem *in a single step*:

EXAMPLE: USING THE HASH TABLE AS A DYNAMIC DATA STEP DICTIONARY

```
data id3max (keep = id transid sum) ;
   length transid sum 8 ;
   dcl hash  ss  (hashexp: 3, ordered: ‘a’) ;
   dcl hiter si  ( ‘ss’ ) ;
   ss.defineKey  ( ‘sum’            ) ;
   ss.defineData ( ‘sum’, ‘transid’ ) ;
   ss.defineDone () ;
   do until ( last.id ) ;
      do sum = 0 by 0 until ( last.transid) ;
         set sample ;
         by id transid ;
         sum ++ amt ;
      end ;
      rc = ss.replace () ;
   end ;
   rc = si.last () ;
   do cnt = 1 to 3 while ( rc = 0 ) ;
      output ;
      rc = si.prev () ;
   end ;
run ;
```

The inner Do-Until loop iterates over each BY-group with the same TRANSID value and summarizes AMT. The outer Do-Until loop cycles over each BY-group with the same ID value and for each repeating ID, stores TRANSID in the hash table SS keyed by SUM. Because the REPLACE method is used, in the case of a tie, the last TRANSID with the same sum value takes over. At the end of each ID BY-group, the iterator SI fetches TRANSID and SUM in the order descending by SUM, and top three retrieved entries are written to the output file. Control is then passed to the top of the implied Data step loop where it encounters the table definition. It causes the old table and iterator to be dropped, and new ones - defined. If the file has not run out of records, the outer Do-Until loop begins to process the next ID, and so on.

BEYOND THE BASICS

NWAY SUMMARY-LESS SUMMARIZATION

The SUMMARY procedure is an extremely useful (and widely used) SAS tool. However, it has one notable shortcoming: It does not operate quite well when the cardinality of its categorical variables is high. The problem here is that SUMMARY tries to build a memory-resident binary tree for each combination of the categorical variables, and because SUMMARY can do so much, the tree carries a lot of baggage. The result is poor memory utilization and slow run times. The usual way of mitigating this behavior is to sort the input beforehand and use the BY statement instead of the CLASS statement. This usually allows running the job without running out of memory, but the pace is even slower - because now, SUMMARY has to reallocate its tree for each new incoming BY-group.

The hash object also holds data in memory and has no problem handling any composite key, but it need not carry all the baggage SUMMARY does. So, if the only purpose is, say, NWAY summarization, hash may do it much more economically. Let us check it out by first creating a sample file with 1 million distinct keys and 3-4 observations per key, then summarizing NUM within each group and comparing the run-time stats. For the reader’s convenience, they were inserted fro the log after the corresponding steps below where relevant:

EXAMPLE: SUMMARY-LESS SUMMARIZATION

```
data input ;
   do k1 = 1e6 to 1 by -1 ;
      k2 = put (k1, z7.) ;
      do num = 1 to ceil (ranuni(1) * 6) ;
         output ;
      end ;
   end ;
run ;
NOTE: The data set WORK.INPUT has 3499159 observations and 3 variables.

proc summary data = input nway ;
   class k1 k2 ;
   var num ;
   output out = summ_sum (drop = _:) sum = sum ;
run ;
NOTE: There were 3499159 observations read from the data set WORK.INPUT.
NOTE: The data set WORK.SUMM_SUM has 1000000 observations and 3 variables.
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           24.53 seconds
      user cpu time       30.84 seconds
      system cpu time     0.93 seconds
      Memory              176723k

data _null_ ;
   if 0 then set input ;
   dcl hash hh (hashexp:16) ;
    hh.definekey  (‘k1’, ‘k2’       ) ;
    hh.definedata (‘k1’, ‘k2’, ‘sum’) ;
    hh.definedone () ;
   do until (eof) ;
      set input end = eof ;
      if hh.find () ne 0 then sum = 0 ;
      sum ++ num ;
      hh.replace () ;
   end ;
   rc = hh.output (dataset: ‘hash_sum’) ;
run ;
NOTE: The data set WORK.HASH_SUM has 1000000 observations and 3 variables.
NOTE: There were 3499159 observations read from the data set WORK.INPUT.
NOTE: DATA statement used (Total process time):
      real time           10.54 seconds
      user cpu time       9.84 seconds
      system cpu time     0.53 seconds
      Memory                            58061k
```

Apparently, the hash object does the job more than twice as fast, at the same time utilizing 1/3 the memory. And at that, the full potential of the hash method has not been achieved yet. Note that the object cannot add NUM to SUM directly in the table, as is usually the case with arrays. Due to the very nature of the process, for each incoming key, SUM first has to be dumped into its host variable, then have the next value added, and finally reinserted into the table. There is a good indication that in the future, a means will be provided for automatically aggregating some statistics specified to the object constructor. In fact, in the beta of Version 9.2 such provisions seem to have already been made.

EXAMPLE: SPLITTING A SAS FILE DYNAMICALLY VIA OUTPUT METHOD

SAS programmers have now been lamenting for years that the Data step does not afford the same functionality with regard to output SAS data sets it affords with respect to external files by means of the FILEVAR= option. Namely, consider an input data set similar to that we have already used for Example 6, but with five distinct ID values, by which the input is grouped:

```
data sample ;
   input id transid amt ;
   cards ;
1  11   40
1  11   26
1  12   97
2  13    5
2  13    7
2  14   22
3  14    1
4  15   43
4  15   81
5  11   86
5  11   85
;
run ;
```

Imagine that we need to output five SAS data files, amongst which the records with ID=5 belong to a SAS data set OUT1, records with ID=2 belong to OUT2, and so on. Imagine also that there is an additional requirement that each partial file is to be sorted by TRANSID AMT.

To accomplish the task in the pre-V9 software, we need to tell the Data step compiler precisely which output SAS data set names to expect by listing them all in the DATA statement. Then we have to find a way to compose conditional logic with OUTPUT statements directing each record to its own output file governed by the current value of ID. Without knowing ahead of the time the data content of the input data set, we need a few steps to attain the goal. For example:

EXAMPLE: SAS FILE SPLIT IN THE PRE-V9-HASH ERA

```
proc sql noprint ;
   select distinct ‘OUT’ || put (id, best.-l)
   into : dslist
   separated by ‘ ‘
   from   sample
   ;
   select ‘WHEN (‘ || put (id, best.-l) || ‘) OUTPUT OUT’ || put (id, best.-l)
   into : whenlist
   separated by ‘;’
   from   sample
   ;
quit ;
proc sort data = sample ;
   by id transid amt ;
run ;
data &dslist ;
   set sample ;
   select ( id ) ;
      &whenlist ;
      otherwise ;
   end ;
run ;
```

In Version 9, not only the hash object is instantiated at the run-time, but its methods also are run-time executables. Besides, the parameters passed to the object do not have to be constants, but they can be SAS variables. Thus, at any point at run-time, we can use the .OUTPUT() method to dump the contents of an entire hash table into a SAS data set, whose very name is formed using the SAS variable we need, and write the file out in one fell swoop:

EXAMPLE: SAS FILE SPLIT USING THE HASH OUTPUT METHOD

```
data _null_ ;
   dcl hash hid (ordered: ‘a’) ;
   hid.definekey  (‘id’, ‘transid’, ‘amt’, ‘_n_’) ;
   hid.definedata (‘id’, ‘transid’, ‘amt’       ) ;
   hid.definedone ( ) ;

   do _n_ = 1 by 1 until ( last.id ) ;
      set sample ;
      by id ;
      hid.add() ;
   end ;
   hid.output (dataset: ‘OUT’ || put (id, best.-l)) ;
run ;
```

Above, the purpose of including _N_ in the hash key list is to make the key unique under any circumstances and thus output all records, even if they contain duplicates by TRANSID AMT. If such duplicates are to be deleted, we only need to kill the _N_ reference in the HID.DEFINEKEY() method. This step produces the following SAS log notes:

```
NOTE: The data set WORK.OUT1 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT2 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT3 has 1 observations and 3 variables.
NOTE: The data set WORK.OUT4 has 2 observations and 3 variables.
NOTE: The data set WORK.OUT5 has 2 observations and 3 variables.
NOTE: There were 11 observations read from the data set WORK.SAMPLE.
```

To the eye of an alert reader knowing his Data step, but having not yet gotten his feet wet with the hash object in Version 9, the step above must look like a heresy. Indeed, how in the world is it possible to produce a SAS data set, let alone many, in a Data _Null_ step? It is possible because, with the hash object, the output is handled completely by and inside the object when the .OUTPUT() method is called, the Data step merely serving as a shell and providing parameter values to the object constructor.

The step works as follows. Before the first record in each BY-group by ID is read, program control encounters the hash table declaration. It can be executed successfully because the compiler has provided the host variables for the keys and data. Then the DoW-loop reads the next BY-group one record at a time and uses its variables and _N_ to populate the hash table. Thus, when the DoW-loop is finished, the hash table for this BY-group is fully populated. Now control moves to the HID.OUTPUT() method. Its DATASET: parameter is passed the current ID value, which is concatenated with the character literal OUT. The method executes writing all the variables defined within the DefineKey() method from the hash table to the file with the name corresponding to the current ID. The Data step implied loop moves program control to the top of the step where it encounters the hash declaration. The old table is wiped out and a fresh empty table is instantiated. Then either the next BY-group starts, and the process is repeated, or the last record in the file has already been read, and the step stops after the SET statement hits the empty buffer.

BEYOND BEYOND THE BASICS

PEEK AHEAD: HASHES OF HASHES

In the Q&A part after the presentation at SUGI 29, one of the authors (*P.D.*) was asked whether it was possible to create a hash table containing references to other hash tables in its entries. Having forgotten to judge not rashly, he emphatically answered “No!” - and was all wrong! Shortly after the conference, one of the world’s best SAS programmers, Richard DeVenezia, demonstrated in a post to SAS-L that such construct is in fact possible, and, moreover, it can be quite useful practically. For examples of such usage on his web site, inquiring minds can point their browsers to

http://www.devenezia.com/downloads/sas/samples/hash-6.sas

Since after all, this is a tutorial, here we will discuss the technique a little closer to *ab ovo*. Let us imagine an input file as in the example 9 above, but with the notable difference that *it is not pre-grouped by ID.* Suppose it looks as follows:

```
data sample ;
   input id transid amt ;
   cards ;
5  11   86
2  14   22
1  12   97
3  14    1
4  15   43
2  13    5
2  13    7
1  11   40
4  15   81
5  11   85
1  11   26
;
run ;
```

Now the same file-splitting problem has to be solved *in a single Data step, without sorting or grouping the file beforehand* in any way, shape, or form. Namely, we need to output a SAS data set OUT1 containing all the records with ID=1, OUT2 – having all the records with ID=2 and so on, just as we did before. How can we do that?
When the file was grouped by ID, it was easy because the hash table could be populated from a whole single BY-group, dumped out, destroyed, and re-created empty before the next BY-group commenced processing. However, in the case at hand, we cannot follow the same path since pre-grouping is disallowed.

Apparently, we should find a way to somehow keep separate hash tables for all discrete ID values, and, as we go through the file, populate each table *according to the ID value* in the current record, and finally dump the table into *corresponding output files* once end of file is reached. Two big questions on the path to the solution, are:

1. How to tell SAS to create an aggregate collection of hash tables keyed by variable ID?
2. How to address each of these tables programmatically using variable ID as a key?

Apparently, the method of declaring and instantiating a hash object all at once, i.e.

```
dcl hash hh (ordered: ‘a’) ;
< key/data definitions >
  hh.definedone () ;
```

which has been serving us splendidly so far, can no longer be used, for now we need to make HH *a sort of a variable* rather than *a sort of a literal* and be able to reference it accordingly. Luckily, the combined declaration above can be split into two phases:

```
dcl hash hh () ;
hh = _new_  hash (ordered: ‘a’) ;
```

This way, the DCL statement only declares the object, whilst the _NEW_ method creates a new instance of it every time it is executed at the run time. This, of course, means that the following excerpt:

```
dcl hash hh () ;
do i = 1 to 3 ;
   hh = _new_  hash (ordered: ‘a’) ;
end ;
```

creates three instances of the object HH, that is, three separate hash tables. It leads us to the question #2 above, or, in other words, to the question: *How to tell these tables apart?* The first, crude, attempt, to arrive at a plausible answer would be to try displaying the values of HH with the PUT statement:

```
26  data _null_ ;
27     dcl hash hh () ;
28     do i = 1 to 3 ;
29        hh = _new_ hash (ordered: ‘a’) ;
30        put hh= ;
ERROR: Invalid operation for object type.
31     end ;
32  run ;
```

Obviously, even though HH *looks* like a ordinary Data step variable, it certainly is not. This is further confirmed by an attempt to store it in an array:

```
33  data _null_ ;
34     array ahh [3] ;
35     dcl hash hh () ;
36     do i = 1 to 3 ;
37        hh = _new_ hash (ordered: ‘a’) ;
38        ahh [i] = hh ;
ERROR: Object of type hash cannot be converted to scalar.
```

```
39     end ;
40  run ;
```

So, we cannot store the collection of “values” HH assumes anywhere in a regular Data step structure. Then, where can we store it? In another hash table, of course! In a hash table, “data” can mean numeric or character Data step variables, *but it also can mean an object.* Getting back to the current problem, the new “hash of hashes” table, let us call it HOH, say, will contain the hashes pertaining to different ID values as its data portion. The only other component we need to render HOH fully operative is a key. Since we need to identify different hash tables by ID, this is the key variable we are looking for. Finally, in order to go through the hash table of hashes and select them for output one by one, we need to give the HOH table an iterator, which below will be called HIH. Now we can easily put it all together:

EXAMPLE: SPLITTING AN *UNSORTED* SAS FILE A HASH OF HASHES AND THE OUTPUT METHOD

```
data _null_ ;
   dcl hash  hoh  (ordered: ‘a’) ;
   dcl hiter hih  (‘hoh’       ) ;
   hoh.definekey  (‘id’        ) ;
   hoh.definedata (‘id’, ‘hh’  ) ;
   hoh.definedone () ;
   dcl hash hh () ;
   do _n_ = 1 by 1 until ( eof ) ;
      set sample end = eof ;
      if hoh.find () ne 0 then do ;
         hh = _new_ hash (ordered: ‘a’) ;
         hh.definekey  (‘_n_’) ;
         hh.definedata (‘id’,’transid’, ‘amt’) ;
         hh.definedone () ;
         hoh.replace () ;
      end ;
      hh.replace() ;
   end ;
   do rc = hih.next () by 0 while ( rc = 0 ) ;
      hh.output (dataset: ‘out’|| put (id, best.-L))  ;
      rc = hih.next() ;
   end ;
   stop ;
run ;
```

As in the case of the sorted input, the program reports in the SAS log thus:

```
NOTE: The data set WORK.OUT1 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT2 has 3 observations and 3 variables.
NOTE: The data set WORK.OUT3 has 1 observations and 3 variables.
NOTE: The data set WORK.OUT4 has 2 observations and 3 variables.
NOTE: The data set WORK.OUT5 has 2 observations and 3 variables.
NOTE: There were 11 observations read from the data set WORK.SAMPLE.
```

Let us delve into the inner mechanics of HOH: How does it all work in concert?

1. The hash of hashes table HOH is declared, keyed by ID and instantiated using the usual "combined" method.
2. A hash object HH, whose future instances are intended to hold partial ID-related data from the input file, is declared, *but not yet instantiated*.
3. The next input record is read.
4. .FIND() method searches HOH table using the current ID as a key. If it does not find an HH hash object with this key, it has not been instantiated yet. Hence, it is now instantiated and stored in HOH by means of the HOH.REPLACE() method. Otherwise, an existing hash instance is copied from HOH into its 'host variable' HH to be reused.
5. The values from the record are inserted via HH.REPLACE() method into the hash table whose instance HH currently holds. Once again, _N_ is used as part of the composite key into HH hashes to discriminate between duplicates by ID and TRANSID.
6. If not end of file, the flow returns to line #3 and the next record is read .
7. Otherwise the iterator HIH (belonging to HOH table) is used to loop through the table one ID at a time.
8. Each time, HIH.NEXT() method extracts a new instance of HH from HOH, then the corresponding table is dumped into a data set with the name formed using the current value of ID.
9. The step is terminated.

Although the described mechanism of HOH operation may seem intimidating at once, in actuality it is not. Simply remember that instances of a hash object can be stored in a hash table (itself an object) as if they were data values, and they can be retrieved from the table when necessary to tell SAS on which one to operate currently. Another term for 'hash', 'associative array' is not at all arbitrary. Just as you can store a collection of related items of the same type in an array, you can store them in a hash - only the storage and retrieval mechanisms are different, and the hash can store a wider variety of types. Thinking through the control flow of the sample program above may be a quick way to bring the (at first, always vague) idea of how hashes of hashes operate in cohesion.

MULTI-LEVEL HASHES OF HASHES

The example above should serve as a good introduction into hash of hashes from the didactic point of view. However, it is still, well, introductory. It is possible to use this capability on a much wider scale and to a much greater benefit. Also, the depth of hash-of-hash storage can exceed that of a single level. But how can it be utilized practically? Let us consider the following real-world problem presented to SAS-L as a question by Kenneth Karan. Assume that we have a data set:

```
data a ;
   input sta: $2. gen: $1. mon: $3. Ssn: 9. ;
cards ;
NY        M       Jan    123456789
NY        M       Feb    123456789
NY        M       Mar    123456789
NY        M       Jan    987654321
NY        M       Feb    987654321
FL        M       Mar    987654321
NY        F       Jan    234567890
NY        F       Feb    234567890
FL        F       Jan    345678901
;
run ;
```

Now imagine that we have to produce a data set printed as follows:

```
Sta    Gen    Mon     _TYPE_    Count
```

```
-----------------------------------
                         0        4
             Feb         1        3
             Jan         1        4
             Mar         1        2
       F                 2        2
       M                 2        2
       F     Feb         3        1
       F     Jan         3        2
       M     Feb         3        2
       M     Jan         3        2
       M     Mar         3        2
FL                       4        2
NY                       4        3
FL           Jan         5        1
FL           Mar         5        1
NY           Feb         5        3
NY           Jan         5        3
NY           Mar         5        1
FL     F                 6        1
FL     M                 6        1
NY     F                 6        1
NY     M                 6        2
FL     F     Jan         7        1
FL     M     Mar         7        1
NY     F     Feb         7        1
NY     F     Jan         7        1
NY     M     Feb         7        2
NY     M     Jan         7        2
```

In other words, we need to produce output similar to what PROC SUMMARY would create without the NWAY option (i.e. for all possible interactions of the CLASS variable values), only the COUNT column would contain the number of distinct SSNs per each categorical keys' combination, rather than the number of observations (_FREQ_) per each combination. Of course, it can be easily accomplished by a series of sorts combined with appropriate BY-processing or by SQL explicitly listing the categorical combinations in the GROUP clause (SQL will still sort behind-the-scenes). *But is it possible to attain the goal via a single pass through the input data?* The answer is "yes"; and the Data step hash object is what affords us such a capability.

EXAMPLE: USING MULTI-LEVEL HASH OF HASHES TO CREATE A SINGLE-PASS EQUIVALENT OF COUNT DISTINCT OUTPUT FITTED TO THE PROC SUMMARY NON-NWAY TEMPLATE

```
%let blank_class = sta gen mon ;
%let comma_class = %sysfunc (tranwrd (&blank_class, %str( ), %str(,))) ;
%let items_class = %sysfunc (countw (&blank_class)) ;

data model ;
   array _cc_ $ 32 _cc_1-_cc_&items_class ;
   do over _cc_ ;
      _cc_ = scan (‖&blank_class=, _i_) ;
   end ;
run ;
proc summary data = model ;
   class &blank_class ;
   output out = types (keep = _cc_: _type_) ;
run ;
data count_distinct (keep = &blank_class _type_ count) ;
   dcl hash  hhh() ;
   hhh.definekey  (‘_type_’) ;
   hhh.definedata (‘hh’,’hi’) ;
   hhh.definedone () ;
   dcl hash  hh() ;
   dcl hiter hi   ;
   dcl hash  h()  ;

   do z = 0 by 0 until (z) ;
      set types end = z nobs = n ;
      array _cc_ _cc_: ;
      hh = _new_ hash (ordered: ‘a’) ;
      do over _cc_ ;
         if _type_ = 0 then _cc_ = ‘_n_’ ;
         if missing (_cc_) then continue ;
         hh.definekey (_cc_) ;
         hh.definedata(_cc_) ;
      end ;
      hi = _new_ hiter (‘hh’) ;
      hh.definedata (‘_type_’,’count’,’h’) ;
      hh.definedone () ;
      hhh.add() ;
   end ;
   do z = 0 by 0 until (z) ;
      set test end = z ;
      pib_ssn = put (ssn, pib4.) ;
      do _type_ = 0 to n - 1 ;
         hhh.find() ;
         if hh.find() ne 0 then do ;
            count = 0 ;
            h = _new_ hash () ;
            h.definekey (‘pib_ssn’) ;
            h.definedone() ;
         end ;
         if h.check() ne 0 then do ;
            count ++ 1 ;
            h.add() ;
         end ;
         hh.replace() ;
      end ;
   end ;
   do _type_ = 0 to n - 1 ;
      call missing (&comma_class) ;
      hhh.find() ;
      do _iorc_ = hi.next() by 0 while (_iorc_ = 0) ;
         output ;
         _iorc_ = hi.next() ;
      end ;
   end ;
   stop ;
run ;
```

Let us try to reverse-engineer the inner mechanics of this code:

1. SSN is converted into its PIB4. image just to cut hash table memory usage by about half.
2. H-tables contain only 4-byte SSN images. There are as many H-tables per _TYPE_ as there are unique _TYPE_ keys; in other words, for each new value of STA, a separate instance of H-table is created, and it will contain only SSN values attributed to that key. The same is true for all other CLASS key combinations.
3. HH-tables are keyed by CLASS key combinations, each key pointing to _TYPE_, count, its own H-table and its iterator object as the data. This way, when a CLASS key of any _TYPE_ comes with the input data, the program knows which H-table, containing its own SSN values, to search.
4. When we need to output the H-table, we then know which particular iterator to use.
5. HHH-table is keyed by _TYPE_, so in this case, it contains 8 HH-tables (in turn, containing H-tables), enabling us to go from one instance of HH-table to another in a simple DO-loop.
6. In the end, _TYPE_ is used as a key to go over each instance of HHH-table and spit out the content of each HH-table to the data set COUNT_DISTINCT.
7. Note that _N_ is used as a single-value (_N_=1) key into the _TYPE_=0 table to give the table a dummy key because without a key, a hash table cannot be defined.

Readers willing to master this rather unusual technique are strongly encouraged to think thoroughly through the details of the code, using the Data step debugger if necessary.

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.

It has been proven through testing and practical real-life application that direct-addressing methods can be great efficiency tools if/when used wisely. Before the advent of Version 9, the only way of implementing these methods in a SAS Data step was custom-coding them by hand.

The Data step hash object provides an access to algorithms of the same type and hence with the same high performance potential, but it does so via a pre-canned routine. It makes it unnecessary for a programmer to know the details, for great results—on par or even better than hand-coding, depending on the situation—can be achieved just by following syntax rules and learning which methods cause the hash object to produce coveted results. Thus, along with improving computer efficiency, the Data step hash objects may also make for better programming efficiency.

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

We would like to than Ilene Brill and Harry Droogendyk for inviting us to present in the Hands-on Workshop section and for their patence in dealing with us.

REFERENCES

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

Dorfman, P and Shajenko, L . 2006 "*Data Step Hash Objects and How to Use Them"*
Proceedings of Annual Northeast SAS Users Group Conference,

Dorfman, P and Shajenko, L . 2006 "Crafting Your Own Index: Why, When, How"
Proceedings of the Twenty-seventh SAS Users Group International Meeting

Dorfman, P and Snell, G. 2002. "*Hashing Rehashed"*.
Proceedings of the Twenty-seventh SAS Users Group International Meeting

Dorfman, P and Snell, G. 2003. "Hashing: Generations."
Proceedings of the Twenty-seventh SAS Users Group International Meeting

Dorfman, P and Vyverman, K. 2005 "*DATA Step Hash Objects as Programming Tools*."
Proceedings of the Thirtieth SAS Users Group International Meeting

Dorfman, P and Vyverman, K. 2009 "The SAS Hash Object in Action"
Proceedings of the SAS Global Forum 2009 Conference

Eberhardt, P 2010 "The SAS Hash Object: It's Time to .find() your way around"
Proceedings of the SAS Global Forum 2010 Conference

Knuth, D . *The Art of Computer Programming,*

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

Secosky, J. 2007 "*Getting Started with the DATA step Hash Object*"
Proceedings of the SAS Global Forum 2007 Conference

Standish, T. *Data Structures, Algorithms & Software Principles in C*.

CONTACT INFORMATION

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

Paul M. Dorfman
4437 Summer Walk Ct.
Jacksonville, FL 32258
(904) 260-6509
Sashole@gmail.com

Peter Eberhardt
Fernwood Consulting Group Inc.
288 Laird Drive
Toronto, ON, M4G 3X5 Canada
(416)429-5705
peter@fernwood.ca
www.fernwood.ca
www.BrewingMadeEasy.com

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

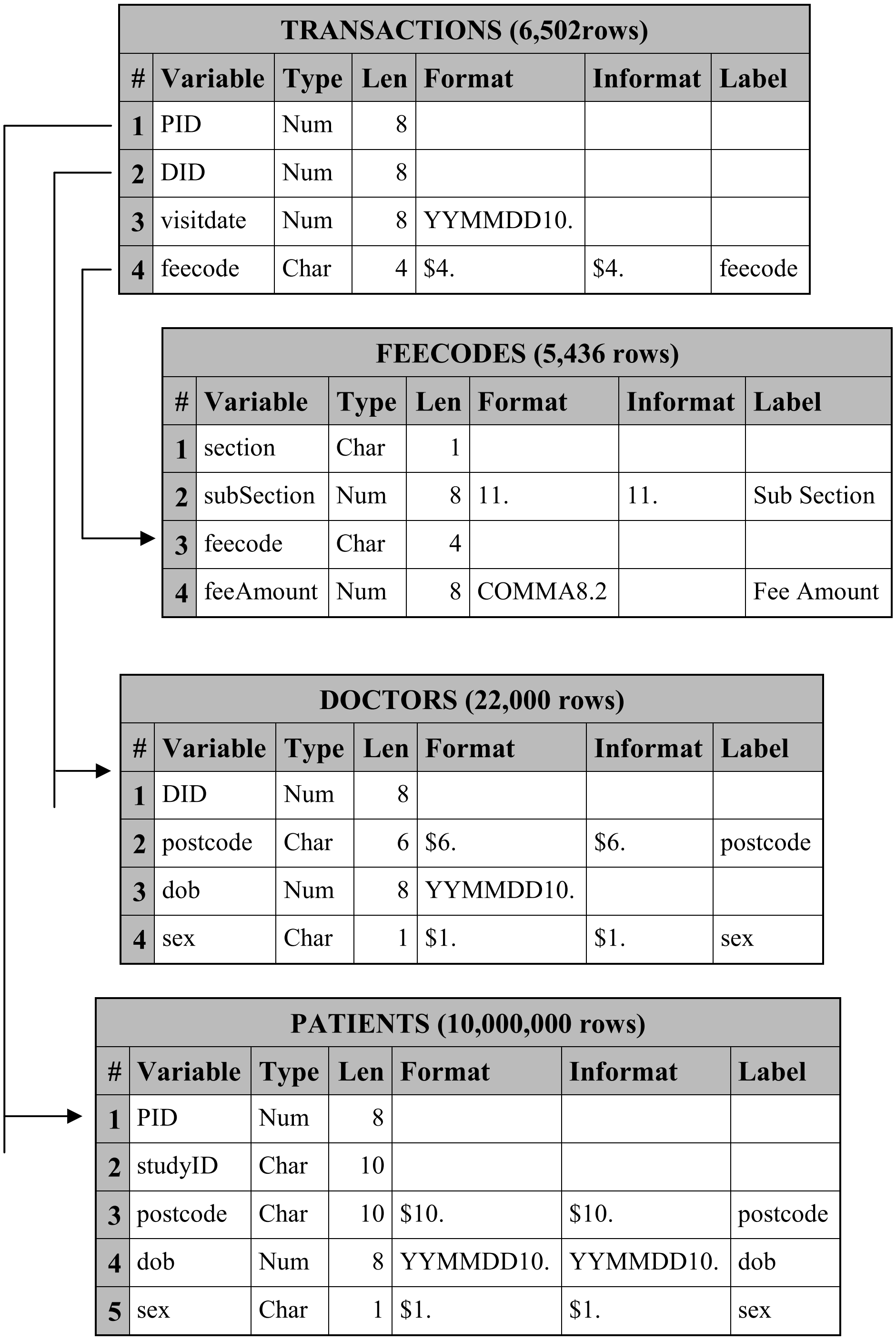
DOCTORS TABLE:

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