Avoid Expensive Sorts – Use a Dynamic Lookup Table

By Bill Fehlner, SAS Institute

April 2006

SAS now provides two pre-defined component objects for use in a DATA step: the hash object and the hash iterator object. These objects enable you to quickly and efficiently store, search, and retrieve data based on lookup keys.

The DATA step component object interface enables you to create and manipulate these component objects by using statements, attributes, and methods. You use the DATA step object dot notation to access the component object's attributes and methods in order to:

- Provide in-memory data storage and retrieval with the hash object.
- Define a data component and key component (think table and index).
- Load data from a SAS table into the hash object (input data does not have to be sorted).
- Lookup a data row based on key values.
- Extract data in sort order with the hash iterator object.
- Add or delete data rows dynamically.

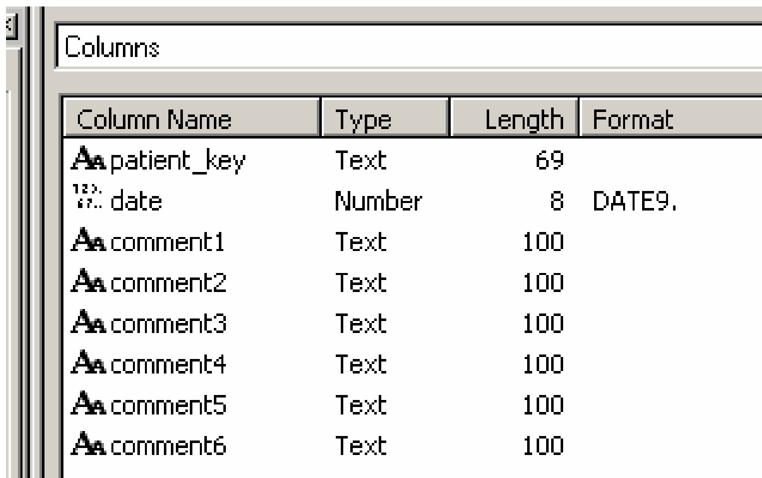
The hash and hash iterator objects have one attribute, fourteen methods, and two statements associated with them. Full documentation is included in the SAS online doc. This article provides practical examples of using these methods and statements.

Assigning surrogate keys with a hash object

Surrogate keys are defined and used when the natural key for a row in a table causes difficulties. For example, even if an account number is retired when an account is closed, the old account number may be reused for a new account after a waiting period. In this case, the natural key, the account number, is not a unique key over time. What is needed is a surrogate key that is unique over time.

In this example, we start with an existing SAS data set, *datamart.oldmaster* that contains health data for each patient on several dates. Each patient is identified by a unique patient key with 69 characters. Although there is no problem with uniqueness of this patient key, a knowledge worker wants to replace this long key with a unique numeric surrogate key before analyzing the data.

The structure of *datamart.oldmaster* for this example is:.


Columns

Column Name	Type	Length	Format
patient_key	Text	69	
date	Number	8	DATE9.
comment1	Text	100	
comment2	Text	100	
comment3	Text	100	
comment4	Text	100	
comment5	Text	100	
comment6	Text	100	

Surrogate keys are generated by:
1. Recognizing a patient key that has not been seen before.
2. Assigning a unique integer value, the patient ID, to this patient key.
3. Applying this same patient ID to all other rows belonging to this patient key.

For small amounts of data, one can group the input data set by patient key using Proc Sort. In a subsequent data step, a different integer value is assigned to the first row in each group and transferred to all the other rows in the group. The code to do this is displayed at the end of this section.

For large amounts of data, the Proc Sort step is costly. In contrast, a hash object can efficiently identify the start of a new group without needing to sort the original data set. The larger the number of rows per patient, the more efficient the hash object becomes.

The SAS code to implement this process with a hash object is as follows:

```
data datamart.newmaster(drop=nextID patient_key rc) ;
     retain nextID 1;
     length rc 8 patient_ID 8 patient_key $ 69;
     if _n_ = 1 then do;
          /* 1. create an empty Hash object */
         declare hash patients( );
         patients.definekey ("patient_key");
         patients.definedata ("patient_ID");
         patients.definedone( );
     end;
          /* 2. read the input data set sequentially*/
     set datamart.oldmaster;
          /* 3. look for a matching patient_id in Hash object */
     rc = patients.find( );
     if rc = 0 then do;
          /* 4. match found, store row with this surrogate key */
         output;
     end;
     else do;
          /* 5. match not found, create and store surrogate key */
         patient_ID = nextID;
         output;
         patients.add( );
         nextID + 1;
     end;
run;
```

Note the commented statements in the code.

- Comment 1:
  - The empty hash object is created once; hence the "if _n_=1 then do;" statement.
  - The "declare hash …" statement assigns the name "patients" to the hash object.
  - Methods are used to create the empty hash object.
    - "Patients.definekey(…)" assigns a key variable to the hash object.
    - "Patients.definedata(…)" assigns a data variable to the hash object.
    - "Patients.definedone( )" creates the empty hash object in memory.
- Comment 2:
  - The master data set (in this case, datamart.oldmaster) is read in sequentially.
- Comment 3:
  - Another method, "patient.find( )", is used to search for a match to patient_key.
- Comment 4:
  - If rc = 0, the search was successful and the output row contains a value for the variable patient_ID.
- Comment 5:
  - nextID contains the next value for the surrogate key, that is, patient_ID.
  - Another method, "patient.add( )" adds a new row to the hash object.

A hash object provides a dynamic storage area in memory that can store each unique patient key – patient ID pair as it is created. As each patient row is read in, the hash object is queried to determine if that patient key is already stored in the hash object. If the patient key is found, the corresponding patient ID value is returned. Otherwise, a new patient ID value is generated and added to the dynamically growing hash object.

In this particular example, the hash object behaves almost like a very fast SAS format. But unlike a format, the hash object can be modified as it is being used.

So how efficient is the hash object approach to generating surrogate keys? A benchmark was done with several sample data sets having various numbers of patients, various numbers of rows per patient, and various numbers of variables. Both the hash approach and the Sort were run with SAS®9 on Windows XP.

As the following table of cpu usage shows, resources needed for the hash object approach go up linearly with the number of rows in the input data set, but climb much faster with the sort based approach.

Number of rows	***Number of patients***	***Cpu - Hash***	***Cpu - Sort***
4374	729	0.03	0.06
86400	1728	0.17	1.17
168750	3375	0.32	2.85
532400	10648	1.06	8.58

The difference is even greater when the real (or elapsed) time is compared. Part of the difference is due to I/O resources used during the sort, and part reflects I/O needed to read in a large data set twice.

Number of rows	***Number of patients***	***Real time - Hash***	***Real time - Sort***
4374	729	0.03	0.06
86400	1728	0.17	4.46
168750	3375	0.32	9.20
532400	10648	1.09	20.81

For completeness, the SAS code used to create surrogate keys with a Proc Sort step is as follows:

```
proc sort data=datamart.oldmaster out=sorted;
    by patient_key;
run;
data datamart.newmaster;
   set sorted;
   by patient_key;
   retain patient_id;
   if first.patient_key then patient_id = _n_;
   drop patient_key;
run;
```

As mentioned at the beginning of this section, this code groups the input data set by patient key using Proc Sort. In the data step, patient_id is given a new integer value whenever the first row in a group is read. The RETAIN statement makes this value available to all the other rows in the group.

Finding a Minimum Value in a Lookup Table

In this example, we start with client locations and branch locations. The challenge is to determine the closest branch for each client. A location is given as the latitude and longitude coordinates derived from the postal code in the address.

Id and location of each client is stored in a data set *Work.clients* that has the following structure:

Columns

Column Name	Type	Length	Format
clientID	Number	8	
lat	Number	8	
long	Number	8	

Transit number and location of each branch is stored in a data set *Work.banks* that has the following structure:

Columns

Column Name	Type	Length	Format
transit	Text	4	
latbank	Number	8	
longbank	Number	8	

Various metrics can be used to determine the distance between two points. In this example, we will use a simplified version of the exact formula based on latitude and longitude coordinates.

For small amounts of data, an SQL query can be used. Small here means at most a few thousand clients and a couple of hundred branches. The SQL query forms a Cartesian product of all clients with all banks, calculates the distance between each client and each branch location, and orders the result. The data step extracts the row for each client with the shortest distance. Unfortunately, although the SQL code is straightforward to write, it is inefficient and eventually runs out of work space as the number of clients grows towards a million. The code is displayed at the end of this section.

The efficient alternative is to use either several arrays or a single hash object to store the branch data and loop over the branches in memory for each client. A distance is still calculated for each branch, but the minimum value is generated on the fly. Thus very little work space is required, not matter how many clients there are.

The SAS code to implement this example with a hash object is as follows:

```
2  data closest_bankH(keep = clientid mindist transit);
3       length rc 8 minbank $ 4;
4         /* create and load hash object */
5       if _n_ = 1 then do;
6           if 0 then set work.banks(keep=transit latbank longbank);
7           declare hash banks(dataset:"work.banks");
8           banks.definekey ("transit");
9           banks.definedata ("transit","latbank","longbank");
10          banks.definedone( );
11          declare hiter retrieve('banks');
12      end;
13        /* search for a minimum distance */
14    set work.clients;
15    mindist=100000000;
16    rc = retrieve.first();
17    do while(rc = 0);
18      dist = sqrt((lat - latbank)**2 + (long - longbank)**2);
19      if dist lt mindist then do;
20         mindist = dist;
21         minbank = transit;
22      end;
23      rc = retrieve.next();
24    end;
25         /* output the result for this client */
26    transit = minbank;
27    mindist = round(mindist,.01);
28    output;
29  run;
30
```

Notes refer to the line numbers on the left

(5-12) "if _n_ = 1" insures that the Hash object is created and loaded with data once.

(6) "if 0 then set" tells the SAS compiler to add the variables transit, latbank and longbank to the program data vector (PDV) with the correct attributes.

(7) "declare hash" statement names the hash object and requests that data from the work.banks data set be loaded into it.

(8) "banks.definekey" method specifies the key variable for lookups.

(9) "banks.definedata" method specifies data variables returned from hash object.

(10) "banks.definedone" method creates the Hash object and loads the data.

(11) "declare hiter" creates a hash Interator object linked to the banks hash object.

(14) Read the client data sequentially.

(16) "retrieve.first" method returns the first row in the hash object linked to the hash iterator object. Return code (rc) is 0 if the read is successful.

(17-24) Continue processing until all rows of the hash object have been returned.

(18) Calculate distance between points using a metric formula.

(19-22) If the new distance is minimum, store the new values.

(23) "retrieve.next" method returns the next row in the hash object linked to the hash iterator object. Return code (rc) is 0 if the read is successful.

(26-28) Output the closest branch for this client.

A benchmark was done with several sample data sets having various numbers of clients. One run was done with three times as many branches. The following table of cpu usage compares the hash based technique with an Array based technique and the SQL technique:

No. clients	***No. branches***	***Cpu-Hash***	***Cpu-Array***	***Cpu-SQL***
1000	200	0.08	0.05	0.54
2000	200	0.14	0.08	1.08
8000	200	0.48	0.24	4.54
25000	200	1.43	0.70	14.46
25000	600	4.19	2.20	44.12

The differences in real (or elapsed) time follow a similar trend:

No. clients	***No. branches***	***real-Hash***	***real-Array***	***real-SQL***
1000	200	0.08	0.05	0.45
2000	200	0.14	0.08	1.20
8000	200	0.48	0.24	7.93
25000	200	1.44	0.70	31.20
25000	600	4.20	2.20	49.47

Note that in this particular example, arrays are more efficient than a hash object. Here we do not need to locate rows in the Hash object based on a character key. There are other data steps where it is necessary to process all the elements in a collection at one point in the code and to locate individual elements based on a key at another point. In such a situation, a hash object will be ideal.

The SAS code to implement this example with several arrays is as follows:

```
 1
 2 data closest_bank(keep = clientid mindist transit);
 3        /* set up arrays to hold branch information */
 4     array transitno (1000) $ 4 _temporary_;
 5     array latitude (1000) _temporary_;
 6     array longitude (1000) _temporary_;
 7       /* store bank information in arrays */
 8     if _n_=1 then do bank = 1 to totobs;
 9        set work.banks nobs=totobs;
10        transitno(bank) = transit;
11        latitude(bank)= latbank;
12        longitude(bank)= longbank;
13     end;
14       /* search for a minimum distance */
15     set work.clients;
16     mindist=100000000;
17     do bank = 1 to totobs;
18       dist = sqrt((lat - latitude(bank))**2 +
19                  (long - longitude(bank))**2);
20       if dist lt mindist then do;
21          mindist = dist;
22          minbank = bank;
23       end;
24     end;
25        /* output the result for this client */
26     transit = transitno(minbank);
27     mindist = round(mindist,.01);
28     output;
29 run;
```

Notes on arrays refer to the line numbers on the left

(4-6) Set up several arrays, since one array cannot store both numeric and character data values. This is not a restriction with a hash object.

(8-13) Load arrays from data set. Note that the maximum size of the arrays must be known in advance. This is not necessary with a hash object.

(18-19) Each array reference returns a single value. A find method on a hash object or a next method on a hash iterator object returns an entire row of values.

The SAS code to implement this example with SQL is as follows:

```
2  proc sql;
3     create table work.temp as
4     select clientid, transit,
5            sqrt((lat - latbank)**2 + (long - longbank)**2)
6                as distance
7     from work.clients, work.banks
8     order by clientid, distance
9        ;
10 quit;
11
12 data closest_bank;
13    set work.temp(rename=(distance = mindist));
14       by clientid;
15    if first.clientid then do;
16       mindist = round(mindist,.01);
17       output;
18    end;
19 run;
20
```

This is typically concise SQL code. However, the use of a Cartesian product makes this inefficient or impossible to use for large data sets.

Conclusion:

The hash object in SAS®9 provides an efficient, convenient mechanism for quick data storage and retrieval. The dynamic nature of a hash object, as well as the ability to use character keys and composite keys, sets it apart from arrays. The two examples here and the example from the July 2005 issue of “Insight” demonstrate how the use of a hash object can eliminate the need for a large sort step.