Paper 306-2008

Better Hashing in SAS® 9.2

Robert Ray and Jason Secosky
SAS Institute Inc., Cary, NC

ABSTRACT

The DATA step hash object is one of the most versatile new features of Base SAS® programming. Since its introduction in SAS® 9, we have numerous accounts from users where the hash object has drastically reduced data processing time for complex data join operations. Based on feedback, we have further enhanced the hash object. For SAS® 9.2, we introduce the ability to store duplicate keys in a hash object and have added a find frequency counter. In this paper, we leverage the duplicate key capability to implement true SQL-like joins as well as partial-key look-ups. In addition, we explore uses for the find frequency counter. Go beyond SAS 9.1 and see why hashing in SAS 9.2 improves how you process data.

INTRODUCTION

A time-consuming part of many SAS programs is looking up a value from one data set in another data set. SAS® 6 lookup methods, like SET with KEY= or a format, are good for many applications. However, in SAS 9, there is a better tool, the DATA Step hash object. The hash object provides a fast, easy way to perform in-memory table lookups without sorting or indexing (Warner-Freeman, 2007). We refer people to Secosky and Bloom (2007) and Loren (2007) for papers on getting started with the hash object.

In this paper, we explore use cases for two enhancements in the hash object in SAS 9.2:

- Duplicate keys, also known as the ability to add multiple data tuples (records) per key
- A "find" frequency counter that is incremented when a key is found

The new duplicate key, or MULTIDATA feature that is introduced in SAS 9.2, is the most significant enhancement to the hash object since its introduction in SAS 9.0. With MULTIDATA, the hash object becomes a much more flexible component that can function as an in-memory index, a sort heap, or even a general-purpose list object. By specifying MULTIDATA:'YES' when creating a hash object, the hash object can store multiple data tuples with a single key entry. It is no longer necessary to choose to keep the first or last occurrence of a key, the hash can keep them all. In addition, the hash object allocates memory as needed. There is no need to pre-size the hash, like an array.

To navigate these new data lists, four new hash object navigation methods are added: HAS_NEXT, FIND_NEXT, HAS_PREV, and FIND_PREV. Because the best documentation for any language feature is example code, here is a small example of using MULTIDATA:'YES'. Bold text represents new features in SAS 9.2.

```
data dup;
  input shoe $ sales;
datalines;
Boot     10
Sandal   11
Slipper  15
Sport    20
Boot     16
Boot      9
Boot    100
Slipper   5
Sandal    5
Slipper   6
Slipper  99
;

data _null_;
  length shoe $ 8 sales 8;
  dcl hash h(dataset:'dup', multidata: 'y');
```

```
     h.defineKey('shoe');
     h.defineData('shoe', 'sales');
     h.defineDone();

     do shoe = 'Boot', 'Casual', 'Sandal', 'Slipper', 'Sport';
       rc = h.find();
       if (rc = 0) then do;
         put shoe= @15 sales=;
         h.has_next(result: r);
         do while(r ne 0);
           h.find_next();
           put shoe= @15 sales= @25 '(dup)';
           h.has_next(result: r);
         end;
       end;
     end;
   run;
```

Results:

```
   shoe=Boot     sales=10
   shoe=Boot     sales=16  (dup)
   shoe=Boot     sales=9   (dup)
   shoe=Boot     sales=100 (dup)
   shoe=Sandal   sales=11
   shoe=Sandal   sales=5   (dup)
   shoe=Slipper  sales=15
   shoe=Slipper  sales=5   (dup)
   shoe=Slipper  sales=6   (dup)
   shoe=Slipper  sales=99  (dup)
   shoe=Sport    sales=20
```

This example uses a DO WHILE loop and the HAS_NEXT and FIND_NEXT methods to loop over duplicates, adding the '(dup)' tag to each duplicate in the output. The order tuples are returned when using FIND_NEXT and FIND_PREV differ between the "classic" release of SAS 9.2 and the "platform" release of SAS 9.2. A bug was fixed where tuples were not stored in the order they were added.

The first section of this paper looks at how MULTIDATA combines an SQL-like join in the DATA step with other DATA step logic. Then, we investigate a technique where MULTIDATA is used to look up data in a hash object when only a partial key is available. The find frequency counter is introduced in the third section with an example that includes summary statistics in report headers. We also compare sum performance against other SAS procedures that compute summary statistics. The last two sections present techniques for emulating lists and sets with a hash object. We highlight how the hash object is not pre-sized and dynamically allocates memory as needed.

"JOINS" WITH DUPLICATE KEYS

A coding pattern we observe is post-processing the results of an SQL procedure query with a DATA step. This is an effective practice when applying a language where it is best suited. The DATA step hash object enables one to go beyond the DATA step MERGE statement and perform SQL-like joins with DATA step logic, all from one step. Combining multiple steps into one can make programs easier to understand and faster by avoiding unnecessary temporary table I/O or multiple passes over large data sets.

In SAS 9.1, the hash object can implement certain types of joins. Namely, equijoins that do not contain duplicate keys and fit into memory. In SAS 9.2, the hash object supports duplicate keys, enabling a more general join mechanism in the DATA step. This section of the paper shows how an SQL join with duplicate keys and a DATA step can be coded in one DATA step using a hash object in SAS 9.2.

There are two common cases where duplicate keys are used in an SQL join. The first is when a Cartesian product is requested. Cartesian products occur when a join condition isn't specified. Cartesian products are usually a mistake that result in large result sets. They are helpful when investigating all-to-all combinations to find an optimal permutation or to ensure that sparse data has necessary columns (Pike, 2007).

A more common use of duplicate keys is when performing a range or "fuzzy" join. Bhat and Suligavi (2001) present an example of this, where they join two tables by patient number, and a date from one table must be within a date range of the other table. We extend their problem by following the join with a DATA step to split the result into separate data sets based on the value of a variable. Let us look at the implementation with PROC SQL and a DATA step and then with one DATA step.

The first two steps in the example create sample data sets, AE and VISIT. AE stores a patient number (PATNO) and the date an adverse event occurs (AEDT). VISIT stores PATNO and the prior and current date (PVISDT and VISDT) of a clinic visit along with other patient data. We want to combine AE and VISIT by PATNO, where AEDT is between PVISDT and VISDT. The PROC SQL step combines the data. A DATA step splits the data into separate data sets based on the value of the variable PHASE.

```
data ae;
  input patno aedt:MMDDYY10.;
datalines;
1 01/10/1996
1 01/12/1996
1 01/15/1996
1 05/12/1996
1 08/16/1996
2 03/15/1996
2 05/20/1996
;

data visit;
  input patno visit phase:$1. pvisdt:MMDDYY10. visdt:MMDDYY10.;
datalines;
1 1 A          . 01/05/1996
1 1 B 01/05/1996 02/04/1996
1 1 C 02/04/1996 03/06/1996
1 2 A 03/06/1996 04/05/1996
1 2 B 04/05/1996 05/05/1996
1 3 A 05/05/1996 06/04/1996
1 3 B 06/04/1996 07/04/1996
1 3 C 07/04/1996 08/03/1996
1 3 D 08/03/1996 09/02/1996
2 1 A          . 01/10/1996
2 1 B 01/10/1996 02/09/1996
2 2 A 02/09/1996 03/11/1996
2 2 B 03/11/1996 04/10/1996
2 2 C 04/10/1996 05/10/1996
2 3 A 05/10/1996 06/09/1996
;

proc sql;
  create table sql_join as
  select *
  from ae, visit
  where ae.patno = visit.patno and
        ae.aedt >  visit.pvisdt and
        ae.aedt <= visit.visdt;
quit;

data Phase_A Phase_B Phase_C Phase_D Unknown;
  set sql_join;
  select(phase);
    when('A') output Phase_A;
    when('B') output Phase_B;
    when('C') output Phase_C;
    when('D') output Phase_D;
```

```
      otherwise output Unknown;
    end;
  run;
```

This solution works. However, if SQL_JOIN is a large temporary data set, the I/O to write it from PROC SQL, and then read it from a DATA step, is significant. With large data sets, PROC SQL might choose a disk-based algorithm to combine the data sets instead of an in-memory algorithm (Lavery 2005). If one of the data sets can fit into memory, a hash object-based solution might be faster. Let's explore a solution that combines the PROC SQL step and DATA step into one DATA step that performs an SQL-like join with a hash object.

A hash object solution loads the smaller of the two data sets, AE, into a hash object named AE_HASH. The smaller of the two data sets is loaded into the hash to use less memory. The key is PATNO. We have a duplicate key situation because the value of PATNO is the same in many rows. In SAS 9.2, the hash object stores duplicate keys when MULTIDATA:'YES' option is specified when the hash object is created.

After the hash object is loaded, we read a PATNO and date range from VISIT. Then, we retrieve each AEDT for the PATNO from AE_HASH. This is done in a DO WHILE loop with the FIND and FIND_NEXT methods. If AEDT is in the date range read from VISIT, a valid combination has been found, and the data are output to a data set based on the variable PHASE.

```
data Phase_A Phase_B Phase_C Phase_D Unknown;
  drop rc;
  if _N_ = 1 then do;
    dcl hash ae_hash(dataset:'ae', multidata:'yes');
    ae_hash.defineKey('patno');
    ae_hash.defineData('aedt');
    ae_hash.defineDone();
  end;

  set visit;

  rc = ae_hash.find();
  do while (rc = 0);
    if pvisdt < aedt <= visdt then
      select(phase);
        when('A') output Phase_A;
        when('B') output Phase_B;
        when('C') output Phase_C;
        when('D') output Phase_D;
        otherwise output Unknown;
      end;
    rc = ae_hash.find_next();
  end;
run;
```

The following notes are seen in the log, indicating the join and data set creation occurred.

```
NOTE: There were 7 observations read from the data set WORK.AE.
NOTE: There were 15 observations read from the data set WORK.VISIT.
NOTE: The data set WORK.PHASE_A has 2 observations and 6 variables.
NOTE: The data set WORK.PHASE_B has 4 observations and 6 variables.
NOTE: The data set WORK.PHASE_C has 0 observations and 6 variables.
NOTE: The data set WORK.PHASE_D has 1 observations and 6 variables.
NOTE: The data set WORK.UNKNOWN has 0 observations and 6 variables.
```

While the code is slightly longer, an in-memory implementation without temporary data sets is guaranteed, producing a fast execution time. Storing duplicate keys in a hash object enables more types of SQL-like joins to be coded along side DATA step code in one DATA step.

PARTIAL KEY LOOKUPS

Hashing data items by a compound key, such as the elements of a customer address (city, county, and state), is common practice. The DATA step hash object makes forming such compound keys easy for the user, hiding the complexity of blending character and numeric variables into a compound key. However, there are times when it is useful to be able to use a subset of a compound key to search for multiple matching records. The same technology that makes multi-key hashing so fast also makes it impossible to do partial-key lookups. While the hash object does not directly support partial-key lookups, it is possible to create auxiliary indexes into a main hash object when creating a hash object with MULTIDATA:'YES'. This facilitates efficient lookups with any predefined subset of the full key set.

An example of this technique features a sample data set containing customer data including the customer's state, county and city of residence. All three of these are used in combination to hash the remaining customer information into the main hash table. Many customers can reside in the same city, so the main hash table must be created with MULTIDATA:'YES'.

To facilitate searches based on a subset of the keys, additional hash tables are constructed containing only key information. Each of these is an auxiliary "index" into the master hash table. These indexes map unique, full-key combinations to a subset of the original key. For example, if our master table has keys A, B and C, an auxiliary hash object might map all combinations of B and C that occur for each unique value of A. Because multiple entries can fall under any key subset, MULTIDATA:'YES' must be used when creating these as well.

To populate these secondary hash objects, the master key data needs to be collapsed into unique combinations of all three keys. For that, a standard hash table is built from the input data set and is then iterated over to populate the secondary hash objects. Once populated, it is possible to build complete keys for all the main table rows that contain the specified sub-key by using the FIND_NEXT method on these secondary hash objects. When that loop surrounds a FIND_NEXT loop on the master hash object, all rows containing the key subset can be extracted efficiently. The example code follows, with bold text indicating SAS 9.2 enhancements.

```
data customer;
  length state $2 county $10 city $15 lname $10 fname $10 mi $1 custID $7;
  input state 1-2 county 4-13 city 14-27 lname 28-37 fname 38-47
        mi 48-49 custID 50-57;
datalines;
NC Wake      Raleigh       Adams     Sam       A 0123456
NC Wake      Cary          Jones     Jenny     J 5432100
NC Wake      Raleigh       Smith     John      S 3434345
NC Wake      Apex          Hart      David     P 4545456
NC Wake      Cary          Jones     Robert    R 9090987
NC Durham    Durham        Ball      Norbert   L 7766653
TX Bexar     San Antonio   Johnson   James     C 2323456
TX Bexar     San Antonio   Antonio   Ray       M 9878645
TX Madina    Hondo         Walls     Jeffery   J 2343435
;

data _NULL_;
  if 0 then set customer; /*-- Load variable attributes --*/

  dcl hash master(dataset:'customer', multidata:'yes');
  master.defineKey('state', 'county', 'city');
  master.defineData(all:'yes');
  master.defineDone();

  dcl hash unique_keys(dataset:'customer');
  unique_keys.defineKey('state', 'county', 'city');
  unique_keys.defineData('state', 'county', 'city');
  unique_keys.defineDone();

  dcl hiter iter('unique_keys');

  dcl hash st(multidata:'yes');
  st.defineKey('state');
  st.defineData('county', 'city');
  st.defineDone();

  dcl hash ct(multidata:'yes');
  ct.defineKey('state', 'county');
  ct.defineData('city');
  ct.defineDone();

  dcl hash cy(multidata:'yes');
  cy.defineKey('state', 'city');
  cy.defineData('county');
  cy.defineDone();

  /*-- Load auxiliary hash objects --*/
  rc = iter.first();
  do while (rc = 0);
    st.add();
    ct.add();
    cy.add();
    rc = iter.next();
  end;

  state = 'NC';
  put; put "Find all customers in the state of " state; put;

  /*-- load first county and city for selected state --*/
  rc = st.find();
  do while (rc = 0);
    rc2 = master.find();
    do while (rc2 = 0);
      put @1 state @4 county @14 city @28 fname @38 lname @48 custid;
      rc2 = master.find_next();
    end;
    rc = st.find_next();
  end;

  state = 'NC';
  county = 'Wake';
  put; put "Find all customers in " county " county, " state; put;

  /*-- load first city for selected state and county --*/
  rc = ct.find();
  do while (rc = 0);
    rc2 = master.find();
    do while (rc2 = 0);
      put @1 state @4 county @14 city @28 fname @38 lname @48 custid;
      rc2 = master.find_next();
    end;
    rc = ct.find_next();
  end;
  stop;
run;
```

When this code is executed, the following output is sent to the log.

```
Find all customers in the state of NC

NC Durham    Durham        Norbert   Ball      7766653
NC Wake      Apex          David     Hart      4545456
NC Wake      Raleigh       Sam       Adams     0123456
NC Wake      Raleigh       John      Smith     3434345
NC Wake      Cary          Jenny     Jones     5432100
NC Wake      Cary          Robert    Jones     9090987

Find all customers in Wake  county, NC

NC Wake      Apex          David     Hart      4545456
NC Wake      Raleigh       Sam       Adams     0123456
NC Wake      Raleigh       John      Smith     3434345
NC Wake      Cary          Jenny     Jones     5432100
NC Wake      Cary          Robert    Jones     9090987
```

FIND FREQUENCY COUNTER

NWAY summarization of data combines values for all combinations of class values or keys. The SUMMARY procedure or PROC SQL with GROUP BY are usually used to compute summarization data. In SAS 9.1, if one would like to use summary data within a DATA step, a PROC SUMMARY or PROC SQL step is required before the DATA step. In SAS 9.2, the hash object can perform additive summarization, enabling programmers to combine summing with DATA step logic in one DATA step instead of two steps. There are also situations where the hash object is faster computing sums than PROC SUMMARY or PROC SQL. This section highlights combining two steps into one DATA step and hash object performance.

SUMS WITH THE HASH OBJECT

Providing summary data in report headings can make reports easier to understand. When creating a report from the DATA step that that includes summary data in headings, a PROC SUMMARY step is usually run before the DATA step to create the summary data. Then, the DATA step reads the summary data, including them in report headings. In SAS 9.2, the hash object can compute sums, removing the need to code two steps. Sums are computed with a hash object when the SUMINC:'*var*' parameter is specified when the hash object is created.

The following code uses PROC SUMMARY and a DATA step to produce a simple report that presents summary data in report headings. The output data set produced by PROC SUMMARY includes the region name and total sales for the region. The DATA step reads the detail data and, for the first row of detail data for a region, outputs a heading that includes the total sales for the region.

```
proc summary data=sashelp.shoes nway;
  class region / missing groupinternal;
  var sales;
  output out=shoes_sum(drop=_:) sum=total_sales;
run;

data _null_;
  file print;
  set sashelp.shoes;
  by region;

  if first.region then do;
    set shoes_sum;
    put _page_;
    put region @30 '(Total Sales: ' total_sales DOLLAR10. ')' //
        'Subsidiary' @15 'Product' @31 'Sales' / 45*'-';
  end;

  put subsidiary @15 product @31 sales;
run;
```

This is sample output from first page of the report. Note the summary data in the report heading.

```
Africa                       (Total Sales: $2,342,588)

Subsidiary    Product         Sales
---------------------------------------------
Addis Ababa   Boot            $29,761
Addis Ababa   Men's Casual    $67,242
Addis Ababa   Men's Dress     $76,793
```

Let's examine how a hash object computes total sales for a region. When the following program starts, a hash object is created to store REGION as the key, and an internal accumulator for sales. To sum values of SALES, the SUMINC:'SALES' parameter is used when creating the hash object. When SUMINC:'*var*' is specified, each time a key is found, the value in the variable *var* is added to an internal accumulator associated with the key. In this example, each time a region is found, the value in SALES is added to the internal accumulator for the region.

This program uses the REF method to add regions to the hash and increment their accumulator. The REF method is new in SAS 9.2. It looks for a key in the hash object. If the key isn't found, it is added. In this program, if the region isn't found in the hash object, it is added, and the accumulator is initialized to the value of SALES. If the region is found, SALES is added to the accumulator.

The second part of the program uses the sums that were computed in the first part of the program. For the first observation read for a region, we get the total sales for a region from the hash by using the SUM method. The SUM method looks up the key and sets the variable after SUM: to the value of the internal accumulator. In this case, TOTAL_SALES gets the value of the internal accumulator for a region. TOTAL_SALES is output in the report header.

```
data _null_;
  if _N_ = 1 then do;
    dcl hash h(suminc:'sales');
    h.defineKey('region');
    h.defineDone();

    do while(not done);
      set sashelp.shoes end=done;
      h.ref();
    end;
  end;

  file print;
  set sashelp.shoes;
  by region;

  if first.region then do;
    h.sum(sum:total_sales);
    put _page_;
    put region @30 '(Total Sales: ' total_sales DOLLAR10. ')' //
        'Subsidiary' @15 'Product' @31 'Sales' / 45*'-';
  end;

  put subsidiary @15 product @31 sales;
run;
```

An array could have been used to accumulate and store the total sales for each region. The difficult part with an array is that we need to map a region name to a number in order to index the array. Another difficulty is that we might not know the number of unique regions or BY groups, making it necessary to write additional code to size the array. A hash object can use a character key, and it dynamically grows to use as much memory as is necessary, eliminating the need to know the number of BY groups.

FASTER SUMMING WITH A SINGLE CPU

PROC SUMMARY is a simple and efficient way to compute summary statistics. However, when running SAS on a single CPU and when there are few values to sum per combination of class values or key, PROC SUMMARY uses more time and memory than summing with a hash object.

Using SAS 9.1, Dorfman and Vyverman (2006) highlight this case and compare PROC SUMMARY and hash object execution times. We add to their work by presenting the multi-CPU case and compare summing with a hash object with SUMINC and PROC SQL with a GROUP BY. In an 8-CPU case, multi-threaded PROC SUMMARY and PROC SQL are comparable to a hash object. In the single CPU case, the hash object with SUMINC is the fastest summing method of the methods shown here.

Dorfman and Vyverman point out that the accumulator has to be moved into a DATA step variable to increment it and put back into the hash object to update it with SAS 9.1. In SAS 9.2, accumulating a value is faster when SUMINC is used. SUMINC adds an internal accumulator for each key, removing the overhead of moving the accumulator around. The example code below extends Dorfman and Vyverman's performance tests by adding a hash object with SUMINC and PROC SQL with GROUP BY.

First, we create a sample data set with 1 million rows and 1 to 6 values per key. The key is made up of a number (K1) and a character value (K2). The variable summed is NUM. Sums are computed using PROC SUMMARY, PROC SQL with a GROUP BY, a DATA step without SUMINC, and a DATA step with SUMINC. We compare the performance running on a multi-CPU system with and without the NOTHREADS option. The NOTHREADS option limits SAS to using algorithms that do not use threads; that is, SAS runs on a single CPU.

```
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;

proc summary data=input nway;
  class k1 k2 / missing groupinternal;
  var num;
  output out=summ_sum(drop=_:) sum=my_sum;
run;

proc sql;
  create table sql_sum as
  select k1, k2, sum(num) as my_sum
  from input
  group by k1, k2;
quit;
```

The next DATA step runs in SAS 9.1 and does not use SUMINC to sum values. Each time the key is found, the variable MY_SUM is incremented, and the REPLACE method puts the new value of MY_SUM back into the hash object.

```
data _null_;
  if 0 then set input;

  dcl hash h(hashexp:20);
  h.defineKey('k1', 'k2');
  h.defineData('k1', 'k2', 'my_sum');
  h.defineDone();

  do while(not done);
    set input end=done;
    if h.find() ne 0 then
      my_sum = 0;
    my_sum + num;
    h.replace();
  end;

  h.output(dataset:'hash_sum');
  stop;
run;
```

The next DATA step sums the values using an internal accumulator for each key. Specifying SUMINC:'NUM' when the hash object is created tells the hash object to add the value of variable NUM to a key's accumulator each time the key is found. We use the REF method, new in SAS 9.2, in a DO WHILE loop to both load the hash object and increment a key's accumulator. The REF method does a key lookup. If REF finds the key, the accumulator is incremented by the value in NUM. If the key isn't found, the key is added to the hash object, and the accumulator is set to NUM.

A key's accumulator is retrieved with the SUM method. A DO WHILE loop and a hash iterator are used to loop over all the values in the hash. The SUM method retrieves a key's accumulator, and we output the accumulator to the output data set. The hash object OUTPUT method isn't used to write the hash object to a data set because the key accumulator isn't output. We are investigating a way to include the key accumulator in OUTPUT method output.

```
  data hash_suminc_sum(keep=k1 k2 my_sum);
    if 0 then set input;

    dcl hash h(suminc:'num', hashexp:20);
    h.defineKey('k1', 'k2');
    h.defineDone();

    do while(not done);
      set input end=done;
      h.ref();
    end;

    dcl hiter hi('h');
    rc = hi.first();
    do while (rc = 0);
      h.sum(sum: my_sum);
      output;
      rc = hi.next();
    end;
    stop;
  run;
```

These programs were run on an 8-CPU Solaris machine with 8 GB of RAM. System options SUMSIZE and SORTSIZE were set to 1G. The CPU times for PROC SUMMARY and PROC SQL are higher than the real times because more than one CPU is used and the CPU time reflects the total CPU time from all CPUs used. The real times for all the methods are similar.

Method	Real Time (sec)	CPU Time (sec)	Memory (MB)
PROC SUMMARY	6.95	14.95	236
PROC SQL with GROUP BY	6.44	11.33	192
Hash Object	7.98	7.96	99
Hash Object with SUMINC	7.08	7.05	99

Next, we ran with the system option NOTHREADS to limit the multi-threaded PROCs to using a non-threaded algorithm on a single CPU. In this case, the hash object times are faster than PROC SUMMARY and PROC SQL. PROC SUMMARY is a general-purpose PROC that computes more than additive NWAY statistics. It uses data structures that are efficient at computing multi-way summarizations when there are numerous class variables. When using formats for classification, PROC SUMMARY formats values in parallel while a similar operation in a DATA step is a sequential algorithm. PROC SQL sorts the key values before summing them. Sorting requires more memory and time than hashing.

The hash object uses less memory than the PROC methods and performs fewer operations to look up and increment its internal accumulator. The hash object with SUMINC is faster than a hash object without SUMINC. SUMINC is faster because the internal accumulator is incremented inside the hash object with the FIND and REF methods instead of moving the value out of the hash to increment it in a DATA step variable.

Method	Real Time (sec)	CPU Time (sec)	Memory (MB)
PROC SUMMARY	12.25	12.20	229
PROC SQL with GROUP BY	9.74	9.69	166
Hash Object	8.17	8.14	99
Hash Object with SUMINC	6.77	6.75	99

A hash object with SUMINC is a tool for combining a PROC SUMMARY and DATA step into one DATA step, potentially avoiding unnecessary temporary data set I/O. In some situations, hash-object summing can be faster than PROC SUMMARY and PROC SQL.

EMULATING LISTS

When MULTIDATA:'YES' is specified when a hash object is created and a duplicate key is added to the hash object, the data are stored in a list off the key. Items that are added first are at the beginning of the list. Items added last are at the end of the list. This list property enables manipulating items in a hash object based on their entry order.

An example of this is modifying a data set in memory. Often, using an in-memory hash operation on data can produce more straight-forward programs than directly manipulating data in a data set. In this example, a meeting of nine people is planned. Eight have reservations at the Hyatt and one at the Marriott. A month before the meeting, the Hyatt phones and reports construction delays mean they cannot accommodate the guests. The Marriott says they can accommodate four more guests. Of the nearby hotels, the Emily Morgan can accommodate two guests and the La Quinta can take the rest. We want to write a program to reassign rooms based on the original order that people made reservations.

The reservation agency sent a data set, HOTEL_TXACT, that contains the transactions when people added or dropped a room reservation, in the order they were received. The DATA step program to reassign rooms consists of two parts. The first part reads HOTEL_TXACT and consolidates the ADD/DROP list into a hash object that represents all people with reservations. For an "add" observation, the ADD method is called to add the person to the hash object. For a "drop" observation, we retrieve the current list of reservations for the hotel from the hash object. Then, we loop over the reservations until we find the name we need to remove and call the REMOVEDUP method.

The second part of the program steps through the reservations for the Hyatt, removes the reservation and then reassigns the person to a different hotel. We use the FIND method to loop over all the hash entries with the key 'Hyatt', in the order they were received. For each entry, the REMOVEDUP method is called to remove the Hyatt reservation. Then, the ADD method is called to assign the first four to the Marriott, the next two to the Emily Morgan, and the rest to the La Quinta. The OUTPUT method writes the hash object to the data set, REASSIGNED, in the new reservation order.

```
data hotel_txact;
  input action $ name $ hotel $;
datalines;
ADD  Alfred   Hyatt
ADD  Ronald   Hyatt
ADD  Louise   Marriott
DROP Louise   Marriott
ADD  Janet    Hyatt
ADD  Carol    Marriott
ADD  Louise   Hyatt
ADD  Jane     Hyatt
ADD  Thomas   Hyatt
ADD  William  Hyatt
DROP Thomas   Hyatt
ADD  Judy     Hyatt
DROP Carol    Marriott
ADD  John     Hyatt
ADD  Thomas   Marriott
;

data _null_;
  dcl hash h(multidata:'yes');
  h.defineKey('hotel');
  h.defineData('hotel', 'name');
  h.defineDone();

  /* ADD and DROP reservations for each hotel */
  do while(not done);
    set hotel_txact end=done;

    if action = 'ADD' then
      h.add();

    else if action = 'DROP' then do;
      drop_name = name;
      rc = h.find();
      do while(rc = 0);
        if drop_name = name then do;
          h.removeDup();
          leave;
        end;
        rc = h.find_next();
      end;
    end;
  end;

  h.output(dataset:'before_reassign');

  /* Reassign hotel guests to other hotels */
  i = 1;
  rc = h.find(key:'Hyatt');
  do while(rc = 0);
    h.removeDup();

    if i <= 4 then hotel = 'Marriott';
    else if i <= 6 then hotel = 'E.Morgan';
    else hotel = 'LaQuinta';

    h.add();

    i = i + 1;
    rc = h.find(key:'Hyatt');
  end;

  rc = h.output(dataset:'reassigned');
  stop;
run;
```

Here is the PRINT procedure output of the data sets BEFORE_REASSIGN and REASSIGNED. The first four people to register at the Hyatt (Alfred, Ronald, Janet, and Louise) are reassigned to the Marriott, after Thomas, who was already registered at the Marriott. The next two people, Jane and William, are assigned to the Emily Morgan. The last two to register at the Hyatt, Judy and John, are at the La Quinta.

```
             BEFORE_REASSIGN               |               REASSIGNED
-------------------------------------------|------------------------------------------
         Obs    hotel       name           |          Obs     hotel      name
                                           |
           1     Hyatt       Alfred        |            1     E.Morgan    Jane
           2     Hyatt       Ronald        |            2     E.Morgan    William
           3     Hyatt       Janet         |            3     Marriott    Thomas
           4     Hyatt       Louise        |            4     Marriott    Alfred
           5     Hyatt       Jane          |            5     Marriott    Ronald
           6     Hyatt       William       |            6     Marriott    Janet
           7     Hyatt       Judy          |            7     Marriott    Louise
           8     Hyatt       John          |            8     LaQuinta    Judy
           9     Marriott    Thomas        |            9     LaQuinta    John
```

This example shows how lists are manipulated in the hash object. Without the hash object, multiple steps would be required to consolidate the ADD/DROP records and then reassign people to hotels. This solution is easy to think through the in-memory operations that the hash object performs on the list of data.

EMULATING SETS

Dynamic data structures use memory as needed. The hash object is the first dynamic data structure in the DATA step. Unlike an array, which must be completely allocated during DATA step compilation, a hash object takes little memory when the DATA step begins and allocates memory on-demand as the program executes. Allocating memory as needed enables the hash object to be used as a set data structure or to build lists in memory.

This section covers two examples, one using the hash object as a set to pick M random integers from a range of 1..N without replication. We use this method to choose a play list of M songs from a library of N songs. The second example extends our play list analogy. We use the hash object as a set to store a play list of random songs until a maximum play time is met.

RANDOM SONG SELECTION

We want to select a play list of M songs from a library of N, where M < N. To avoid a repetitive play list, we don't want any duplicates. This is called random selection without replication. Floyd (1987) created a random selection algorithm without replication that is fast, efficient, and appropriate for large data sets. Floyd's algorithm uses a set, S, to track which integers have been chosen. We use Floyd's algorithm and a hash object to represent the set.

Floyd's algorithm makes one call to RANUNI for every random integer selected. When choosing five items from ten, we first choose four from nine and pick one more item from those left. When choosing four from nine, we choose three from eight and pick one more item from those left. This algorithm recurs until one item is chosen from six. The beauty of this algorithm is that each item has the same probability of being chosen and RANUNI is called once for each item chosen. To avoid recursion, an iterative version of this algorithm is in this pseudo code:

```
initialize set S to empty
for J = N - M + 1 to N
  T = int(ranuni(0)*J)+1;
  if T is in S then
    T = J
  insert T in S
```

The SAS code to implement this algorithm comes in two parts. We use one DATA step to create a library of songs. Then, we use another DATA step to pick M out of N random numbers, and select observations from the song data set for our play list.

```
data songs;
  length artist title $ 32;
  format time time.;
  input artist& title& time:STIMER.;
datalines;
Seal              Bring It On                   3:58
Rod Stewart       Every Picture Tells A Story   6:00
```

```
The Cars          Just What I Needed            3:46
Eric Clapton      Wonderful Tonight             3:43
Styx              Blue Collar Man               4:06
Don Henley        The Boys Of Summer            4:51
Steely Dan        Rikki Don't Lose That Number  4:33
Steve Winwood     Roll With It                  5:20
Jackson Browne    Running On Empty              5:30
Sons of Champlin  Hold On                       8:33
Toto              We Made It                    3:59
Tracy Chapman     Fast Car                      4:56
Robert Cray       Right Next Door               4:21
Bonnie Raitt      Something To Talk About       3:48
;

/* Random selection, pick M=5 of N=14 songs */
%let m=5;
data playlist(keep=artist title time);
  dcl hash s();
  s.defineKey('t');
  s.defineDone();

  /* Select M random numbers from 1..N */
  /* N is set by the NOBS= option on the SET statement */
  do j = n-&m.+1 to n;
    t = int(ranuni(0) * j) + 1;
    /* If T is in S (T is already chosen), then T = J */
    if s.find() = 0 then
      t = j;
    s.add();
  end;

  /* Subset SONGS based on the random play list in S */
  dcl hiter iter('s');
  rc = iter.first();
  do while (rc = 0);
    set songs nobs=n point=t;
    output;
    rc = iter.next();
  end;
  stop;
run;
```

Here is PROC PRINT output of the result data set, PLAYLIST.

```
Obs    artist           title                           time

 1     Rod Stewart      Every Picture Tells A Story    0:06:00
 2     Seal             Bring It On                    0:03:58
 3     Tracy Chapman    Fast Car                       0:04:56
 4     Bonnie Raitt     Something To Talk About        0:03:48
 5     Steve Winwood    Roll With It                   0:05:20
```

The two loops in the DATA step could be combined. They were left apart to distinguish using the hash as a set from iterating over it to select songs.

An array of N items could have represented the set. If N is small, this is a fine solution. If N is large, time is spent allocating memory that goes unused if M is much smaller than N. The hash object allocates memory for only M items.

The SURVEYSELECT Procedure, part of SAS/STAT® software, uses Floyd's algorithm for random sampling. PROC SURVEYSELECT could have been used instead of a hash object. In the next section, we see that a hash object is more flexible than PROC SURVEYSELECT because a random sample can be chosen and manipulated in one step when using a hash object, avoiding I/O to a temporary data set and combining sampling with DATA step logic.

RANDOM SONG SELECTION WITH TIME CONSTRAINT

We want to select any number of random songs to fill a particular block of time. Our implementation continues to treat a hash object as a set. RANUNI is called to select a random song. The hash object H stores the songs that have been chosen so duplicates do not appear in the play list. The program continues to select songs, decrementing the time remaining for songs, until the program is within 60 seconds of the total time. Using the SONGS data set from the prior section, here is the code to implement this solution.

```
%let playtime_in_minutes=30;
data playlist(keep=artist title time);
  dcl hash h();
  h.defineKey('t');
  h.defineDone();

  /* convert minutes to seconds */
  total_time = &playtime_in_minutes * 60;

  /* N is set by the NOBS= option on the SET statement */
  do j = 1 to n;
    t = int(ranuni(0) * n)+1;
    if h.find() = 0 then
      continue;
    h.add();

    set songs nobs=n point=t;

    /* Ignore songs that put us over */
    total_time = total_time - time;
    if total_time < 0 then
      total_time = total_time + time;
    else
      output;

    /* If we are within 60 seconds of total time, then stop */
    if total_time < 60 then
      leave;
  end;
  stop;
run;
```

Here is the PROC PRINT output of the resulting data set, PLAYLIST.

```
Obs    artist          title                              time

  1    Seal            Bring It On                      0:03:58
  2    Steely Dan      Rikki Don't Lose That Number     0:04:33
  3    Don Henley      The Boys Of Summer               0:04:51
  4    The Cars        Just What I Needed               0:03:46
  5    Bonnie Raitt    Something To Talk About          0:03:48
  6    Robert Cray     Right Next Door                  0:04:21
  7    Eric Clapton    Wonderful Tonight                0:03:43
                                                        =======
                                                        0:29:00
```

This program demonstrates using a hash object as a set to ensure there isn't duplication in a random selection of songs. PROC SURVEYSELECT could have been used for a random selection. It couldn't be used to fill an arbitrary block of time. The general purpose nature of the hash object and DATA step enables programmers to read and process a group of items in memory from one step. This avoids unnecessary I/O to temporary tables and consolidates algorithms into fewer steps, making them easier to understand.

CONCLUSION

The examples in this paper highlight the two main features added to the hash object in SAS 9.2, multiple data tuples per key and the find frequency counter. Multiple data tuples per key, or MULTIDATA, enables using the hash object for SQL-like queries, the ability to perform partial key lookups, and to be able to treat a hash object as a list. The find frequency counter provides a way to compute some summary statistics from within a DATA step. Both of these new features enable simplifying code by combining multiple steps into one and improve performance by removing I/O to temporary data sets and the furthering the ability to combine data without sorting.

REFERENCES

Bentley, Jon Louis, and Robert Floyd. 1987. "A Sample of Brilliance." *Communications of the Association for Computing Machinery*, 30, 754–757.

Bhat, Gajanan, and Raj Suligavi. 2001. "Merging Tables in DATA Step vs. PROC SQL: Convenience and Efficiency Issues." *Proceedings of the Twenty-Sixth SAS Users Group International Meeting.* Cary, NC: SAS Institute Inc. Available at www2.sas.com/proceedings/sugi26/p104-26.pdf.

Dorfman, Paul, and Koen Vyverman. 2006. "DATA Step Hash Objects as Programming Tools." *Proceedings of the Thirty-First SAS Users Group International Meeting.* Cary, NC: SAS Institute Inc. Available at www2.sas.com/proceedings/sugi31/241-31.pdf.

Lavery, Russ. 2005. "The SQL Optimizer Project: _Method and _Tree in SAS 9.1." *Proceedings of the Thirtieth SAS Users Group International Meeting.* Cary, NC: SAS Institute Inc. Available at www2.sas.com/proceedings/sugi30/101-30.pdf.

Loren, Judy. 2006. "How Do I Love Hash Tables? Let Me Count The Ways!" *Proceedings of the Nineteenth Northeast SAS Users Group Meeting.* Cary, NC: SAS Institute Inc. Available at www.nesug.info/Proceedings/nesug06/ap/ap11.pdf.

Pike, Greg. 2007. "Cartesian Products: Don't Fear this Reaper." (Accessed February 24, 2008) Available at www.singlequery.com/?p=16.

Secosky, Jason, and Janice Bloom. 2007. "Getting Started with the DATA Step Hash Object." *Proceedings of the First SAS Global Forum.* Cary, NC: SAS Institute Inc. Available at www2.sas.com/proceedings/forum2007/271-2007.pdf.

Warner-Freeman, Jennifer. 2007. "I Cut my Processing Time by 90% Using Hash Tables - You Can Do It Too!" *Proceedings of the Twentieth Northeast SAS Users Group Meeting.* Cary, NC: SAS Institute Inc. Available at www.nesug.info/Proceedings/nesug07/bb/bb16.pdf.

ACKNOWLEDGMENTS

Many thanks to Al Kulik, who implemented the DATA step hash object and hash iterator. We also thank Janice Bloom for reviewing the paper.

CONTACT INFORMATION

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

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