# A Hash Alternative to the PROC SQL Left Join

Kenneth W. Borowiak, Howard M. Proskin & Associates, Inc., Rochester, NY

#### **ABSTRACT**

Prior to the introduction of SAS® Version 9, access to a pre-packaged hash routine was only available through PROC SQL. This fast table look-up technique is used when performing an inner join between two tables and one of the tables will fit into memory. Unfortunately, PROC SQL does not use hashing when an outer join is requested. With the introduction of the Hash Object in SAS V9, hashing-made-easy methods are now available in the DATA step. This paper demonstrates how a slight augmentation to some established hash table look-up code can be used as an efficient alternative to the PROC SQL left join.

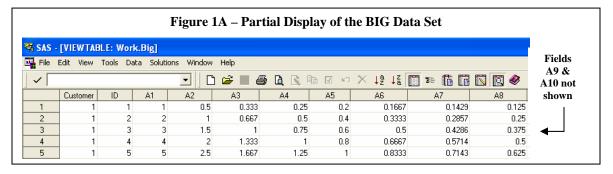
#### INTRODUCTION

The SAS implementation of the Structured Query Language via PROC SQL provides both a complimentary and alternative set of tools vis-à-vis the DATA step for the programmer, depending on the task. Prior to SAS Version 9, PROC SQL held a distinct advantage over the DATA step merge when implementing an *inner join* with equality conditions. This type of join brings together information from two or more tables together where the join condition between them is satisfied. If the smaller of the tables could be judiciously loaded into memory<sup>1</sup>, the SQL optimizer would choose *hashing* to perform the join. A hash join brings information from two tables together *without having to first sort the tables*.

Unfortunately, PROC SQL does not use hashing when executing an outer join. Prior to the introduction of SAS® Version 9, access to a pre-packaged hash routine was only available through PROC SQL. With the introduction of the Hash Object in SAS V9, hashing-made-easy methods are now available in the DATA step. This paper demonstrates how a slight augmentation to some established hash table look-up code can be used as an efficient alternative to the PROC SQL left join.

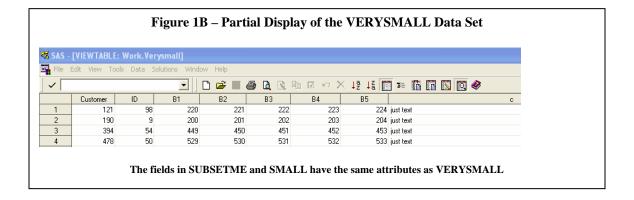
### **PRELIMINARIES**

The examples in this paper will use four data sets. The first is BIG (4 million observations) that has the numeric fields CUSTOMER and ID as the key variables and the numeric fields A1 through A10 as the satellite fields. The second data set is SUBSETME (4.4 million observations) which also has CUSTOMER and ID as the key fields and B1 through B5 as the numeric satellite variables. The keys in BIG and SUBSETME are unique, but there are observations in BIG and SUBSETME where there is no corresponding match with the keys in the other table. The data sets VERYSMALL and SMALL are randomly selected observations from SUBSETME, which contain 1,000 and 800,000 observations, respectively. The program which generated the data sets can be found in the Appendix at the end of the paper. The data sets should be considered unsorted, though they were generated in order by the key fields. Figures 1A and 1B contain partial displays of the BIG and VERYSMALL data sets, respectively.



\_

<sup>&</sup>lt;sup>1</sup> According to Lavery[2005-A], this means that 1% of the needed variables from the smaller table could fit into one memory buffer. Adjusting the buffersize option can influence the optimizer's decision to use a hash join.



## PROC SQL JOIN TECHNIQUES

The type of join requested in the FROM or WHERE clauses of PROC SQL affects the technique the optimizer decides upon to execute the join. To see which type of join technique is used in a query, the SAS undocumented option *\_method* can be used. For example, consider the query below where an inner join is requested to merge information from the BIG and VERYSMALL tables.

```
Figure 2 – Example of a PROC SQL Inner Join Using
                                the Hash Join Technique
/*-- Inner join of the BIG and VERYSMALL data sets --*/
proc sql _method ;
  create table sql_hj as
  select
  from
            Big T1,
            VerySmall T2
            T1.customer=T2.customer and
  where
            T1.id=T2.id;
  quit ;
NOTE: SQL execution methods chosen are:
                                                             sqxihsh is the code
                                                            indicating that a hash
      saxcrta
                                                              ioin was used to
          sqxjhsh
                                                           perform the operation
              sqxsrc( WORK.BIG(alias = T1) )
              sqxsrc( WORK.VERYSMALL(alias = T2) )
NOTE: Table WORK.SQL_HJ created, with 313 rows and 18 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time
                         3.24 seconds
      cpu time
                         1.95 seconds
```

Perusing the SAS log we find information regarding SQL execution techniques courtesy of the\_method option. In particular, sqxjhsh is the code indicating use of a hash join. This should not come as too much of a surprise considering VERYSMALL is, well, smallish and would likely meet the condition of fitting into memory. More evidence to support the conjecture that a hash join was implemented is the fast run time. Had VERYSMALL been too large to load into memory and in the absence of a compound index on the key variables, a  $merge\ join$  would have been employed requiring both BIG and VERYSMALL to be sorted behind the scenes. Even with all the recent efforts (e.g. threaded sort) SAS has invested in its sorting routines, sorting the 4 million observations of BIG in 3.24 seconds with typical modern day computing resources is unlikely.

Now let us consider the task where a left join is requested between BIG and VERYSMALL. For any observation where the key fields in BIG are found in VERYSMALL, the satellite variables in VERYSMALL will accompany the satellite variables in BIG. If there is not a match between keys in BIG and VERYSMALL, the observation from BIG is kept and the satellite fields from VERYSMALL will all have missing values. Since the keys are unique in both of the data sets, the resulting table of a left join between the two will have the same number of observations as BIG (i.e. 4 million).

```
Figure 3 – Example of a PROC SOL Left Join Using
                                the Merge Join Technique
/*-- Left Join of BIG and VERYSMALL tables --*/
proc sql _method ;
  create table lj1 as
  select
             Big T1
  from
             left join
             VerySmall T2
             on T1.customer=T2.customer and
                 T1.id=T2.id;
  quit;
NOTE: SQL execution methods chosen are:
                                  sqxjm is the code
      sqxcrta
                                                                 A join merge needs
                               indicating a join merge
          saxim
                                                                the tables to be sorted
              saxsort
                                                                and will perform the
                  sqxsrc( WORK.VERYSMALL(alias = T2) )
                                                                 operation if needed
                  sqxsrc( WORK.BIG(alias = T1) )
NOTE: Table WORK.LJ1 created, with 4000000 rows and 18 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time
                          55.24 seconds
      cpu time
                          15.20 seconds
```

By applying the \_method option on the PROC SQL statement, we find in the log that the optimizer has decided upon a join merge (*sqxjm*) to execute the left join. In order to do so, both of the tables needed to be sorted (*sqxsort*). The sorting of the tables happens behind the scenes, possibly unbeknownst to the unsuspecting PROC SQL user. From the authors experience querying SAS data sets<sup>2</sup> with PROC SQL, the join merge is the only join technique used to execute any outer join (left, right and full)<sup>3</sup>. When sorting of the tables is needed, particularly if at least one of the tables is large, possibly a more efficient alternative exists.

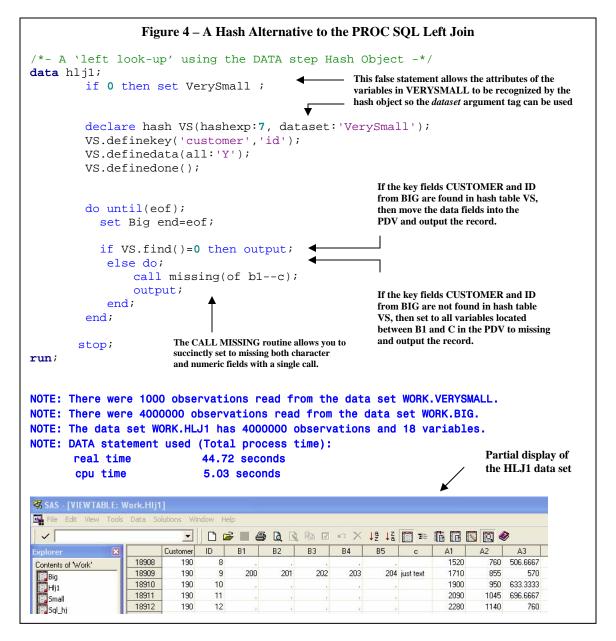
### **DATA STEP HASH OBJECT**

Refuge from PROC SQL's join merge that requires sorting is sought using the Hash Object made available with the advent of SAS Version 9. Using dot notation syntax, a pre-packaged hash routine is at the disposal of the programmer and using the Hash Object is gaining prominence among SAS users. Excellent references on how the Hash Object can be used and details of its inner workings are referenced at the end of the paper. The code in Figure 4 on the next page demonstrates how some well-established hash table look-up code can be augmented to mimic the functionality of the PROC SQL left join.

-

<sup>&</sup>lt;sup>2</sup> This specifically excludes queries using the Pass-Through facility of PROC SQL.

<sup>&</sup>lt;sup>3</sup> That is, without intervening on the optimizer's behalf. More discussion on this later in the paper.



The first step in using the Hash Object is to instantiate it, which is accomplished with the DECLARE statement. The hash table in this example is named VS. The hash table is going to be populated with contents of the VERYSMALL dataset by means of the dataset argument tag, which appears in the parentheses of the DECLARE statement. Assigning a value of 7 to the hashexp argument tag says the hash table should have 2\*\*7=128 top level buckets. Next, the key and data fields of the hash table need to be defined. The DEFINEKEY method is used to indicate that CUSTOMER and ID are the key fields. The DEFINEDATA method is used to indicate what fields associated with the key fields should be loaded into the hash table. The short-cut ALL:Y is used in this example to indicate that all fields from VERYSMALL should be loaded in as data in the hash object, including the key fields CUSTOMER and ID<sup>4</sup>. The DEFINEDONE method concludes the key and data definitions.

<sup>&</sup>lt;sup>4</sup> In this example, loading CUSTOMER and ID into the hash table as data elements in the hash table is unnecessary and consumes memory resources. However, using the ALL argument tag in the DEFINDATA method allows you to avoid typing in a long list of satellite variables.

Now we proceed onto the table look-up using an explicit loop to iterate through the BIG data set. The FIND method is used to see if the current values of the CUSTOMER and ID in the BIG data set are found as keys in the hash table VS. If they are, the value returned by the FIND method is 0 and the data elements in the hash table associated with the keys are moved into the program data vector (PDV) and the record is dumped to the output data set. If we wanted to output only records from BIG that had corresponding keys in the hash table, we would be all done. This is the standard table look-up that is analogous to an inner join in PROC SQL. However, we are interested in mimicking the behavior of a left join. We do not want exclude any records from BIG where there is no corresponding match in the hash table. If the value from FIND method is something other than 0, then no new information is brought into the PDV from the hash table. So we need to set to missing all the values of the variables that could be contributed by the VERYSMALL data set via the hash object. This is accomplished in a very succinct way using the CALL MISSING routine with a name range list, which allows you to assign missing values to both character and numeric fields in the list.

The results from the 'left look-up' using the DATA step hash object in Figure 4 shows a modest decrease in run time of about ~20% (55.24 vs. 44.72 seconds). The decrease in run time comes at the expense of an increase in memory usage, as the hash table resides in RAM and sorting from SQL takes place on disk. But in this example, essentially the only source of the sorting burden was from the BIG data set. Consider using the DATA step hash object to execute a left look-up on the SMALL data set and its 800,000 observations.

```
Figure 5 –A Left Look-up Using the SMALL Data Set
/*-- Results of the SQL left join of BIG and SMALL tables --*/
NOTE: SQL execution methods chosen are:
      sgxcrta
          sqxjm
              saxsort
                                                                       A second
                  sqxsrc( WORK.SMALL(alias = T2) )
                                                                     relatively large
              sqxsort
                                                                    table now needs
                  sqxsrc( WORK.BIG(alias = T1) )
                                                                     to be sorted ...
NOTE: Table WORK.LJ2 created, with 4000000 rows and 18 columns.
                                                                     ... resulting in
NOTE: PROCEDURE SQL used (Total process time):
                                                                     an increase in
      real time
                           1:15.34 seconds
                                                                       run time.
      cpu time
                           15.84 seconds
      Results of the left look-up using the DATA step Hash Object --*/
NOTE: There were 800000 observations read from the data set WORK.SMALL.
NOTE: There were 4000000 observations read from the data set WORK.BIG.
NOTE: The data set WORK.HLJ2 has 4000000 observations and 18 variables.
NOTE: DATA statement used (Total process time):
                           48.16 seconds 👞
       real time
                                                         The run time increased only
                            5.03 seconds
       cpu time
                                                          marginally when the much
                                                          larger SMALL data set is
                                                          loaded into the hash table
```

The results in Figure 5 show that the PROC SQL left join took 1:15 seconds to execute when the table SMALL replaced VERYSMALL in the query in Figure 3. The additional 20 seconds (~36%) to execute the query can be attributed to having to sort a relatively larger second data set. On the other hand, the DATA step hash object solution only increased run time by a few seconds. The loading of a larger data set into the hash table was a bigger task. However, the time it takes to find the keys in the hash object with 1,000

entries is marginal to finding the keys in a hash table of 800,000. Hashing is a hybrid technique that leverages the speed of a direct address look-up technique and the not-as-fast but memory-conserving approach of a binary search at the end stage of the algorithm that executes in O(log(# of hash entries/ # hash buckets)) time. Translation: The time it takes to find keys in a hash table is relatively insensitive to the number of entries in it.

To put using the Hash Object to the test for performing a left look-up, let us consider loading SUBSETME into the hash table. Loading the keys and associated data from VERYSMALL and its 1,000 observations into a hash table is not very taxing on memory resources. However, loading the six satellite variables of SUBSETME's 4.4 million records is another story. Paul Dorfman and Lessia Shajenko [2006] show that the memory can be used more judiciously by loading only the keys and a record id into the hash table and then using the POINT= option on a SET statement to read the satellite fields from the look-up data set. Figure 6 below shows how the DATA step in Figure 4 can be modified as such.

```
Figure 6 – Hashing only the Keys and Record ID Pointer
/*-- Left look-up that does not hash the satellite --*/
data hlj3C;
  declare hash Sub(hashexp:7);
                                               Only a field with a record identifier of
   Sub.definekey('customer','id');
                                                SUBSETME will be loaded into the
   Sub.definedata('n');
                                                  data portion of the hash table
   Sub.definedone();
  do until(eof_SubsetMe) ;
       set SubsetMe end=eof_SubsetMe ;
       n+1 ;
                                                      Populate the hash table
       Sub.add();
  end;
  do until(eof);
                                                       If the values of CUSTOMER and
       set Big end=eof;
                                                        ID from BIG are found in the
                                                       hash table, move record n from
       if Sub.find()=0 then do;
                                                        SUBSETME into the PDV and
               set SubsetMe point=n;
                                                             output its content
               output;
           end;
         else do;
                                                      If the key fields are not found in hash
               call missing(of b1--c);
                                                       table Sub, then set to all variables
               output;
                                                     located between B1 and C in the PDV to
          end;
                                                         missing and output the record.
  end;
 stop;
run;
NOTE: There were 4400000 observations read from the data set WORK.SUBSETME.
NOTE: There were 4000000 observations read from the data set WORK.BIG.
NOTE: The data set WORK.HLJ3C has 4000000 observations and 18 variables.
NOTE: DATA statement used (Total process time):
       real time
                            1:09.66 seconds
       cpu time
                            15.63 seconds
/*-- Results of the SQL left join of BIG and SUBSETME tables --*/
NOTE: Table WORK.SQL_LJ3 created, with 4000000 rows and 18 columns.
NOTE: PROCEDURE SQL used (Total process time):
                                                           Requires the 4M and 4.4M
                           1:54.28 seconds
      real time
                                                           observation data sets to be
                           25.90 seconds
      cpu time
                                                           sorted to execute the join
```

Rather than using the dataset argument tag to directly load SUBSETME into the hash table, the keys from each observations are loaded one at time with the first DO loop and the ADD method. The variable n is created and serves as the observation number from SUBSETME and is the only data portion of the hash table. The second DO loop iterates through BIG, where the values of CUSTOMER and ID are searched for in the hash table. If a record is found, the variable n gets moved into the PDV. Record n of SUBSETME is retrieved using the POINT= option and the satellites fields from SUBSETME are brought into the PDV and the record is output. If the values of the key fields in BIG are not found in the hash table, the values of the satellite fields contributed by SUBSETME are set to missing. Inspecting the run times of the DATA step and the PROC SQL statement (which is not explicitly shown), we find that the ratio of the run times is about 1.5. In the very first example where a 'large' (i.e. BIG) and a 'small' (i.e. VERYSMALL) table were involved, the ratio of the run times was close to 1. This indicates that there is at least some range where as the size of the data sets involved grow, a left look-up hash solution outperforms the PROC SQL left join.

#### OTHER CONSIDERATIONS

- The examples covered in the paper worked under the assumption that the table on the left side of the join was unsorted. In the case where that table is sorted by the key fields, there may not be sufficient reason to stray from the PROC SQL left join.
- Using the DATA step Hash Object in Version 9.1 requires the attributes of the key variables (i.e. name, field type, and length) in the hash table to match that of key fields providing the values to be searched. This will not result in error your SAS log if this condition is not met, but you will not get the result you were expecting. The 'parameter matching' burden is on the shoulders of the programmer.
- The Hash Object does not accept duplicate key values. Dorfman and Shajenko [2005-B] demonstrate how to work around this limitation by creating an auxiliary key field (e.g. observation number) and using a second hash table.
- If the table on the right side of a PROC SQL left join is indexed on the key fields, an index join (execution code *sqxjndx*) may be employed if the IDXNAME data set option is used. By searching the disk-resident index on right hand table, this removes the need to sort the table on the left side.

#### **CONCLUSION**

When performing an outer join in PROC SQL, if the tables involved are not sorted then they will be sorted behind the scenes. This can lead to run times that are suboptimal. This paper demonstrated how some well-established hash table look-up code in the DATA step can be augmented to mimic the functionality of the PROC SQL left join where no sorting is needed. In the examples explored in the paper, when at least one of the tables is large and unsorted then there exists potential to reduce run-time using the DATA step Hash Object. When both of the tables are large and unsorted, the benefits of the using hashing increase.

## REFERENCES

Dorfman, P., Snell, G. (2003), "Hashing: Generations", *Proceedings of the 28<sup>th</sup> Annual SAS Users Group International* 

Dorfman, P., Vyverman, K. (2004), "Hash Components Objects: Dynamic Data Storage and Table Look-Up", *Proceedings of the 29<sup>th</sup> Annual SAS Users Group International* 

Dorfman, P., Vyverman, K. (2005-A), "Data Step Hash Objects as Programming Tools", *Proceedings of the 30<sup>th</sup> Annual SAS Users Group International* 

Dorfman, P., Shajenko, L. (2005-B), "Crafting Your Own Index: Why, When, How", *Proceedings of the 18th Annual Northeast SAS Users Group Conference* 

Lavery, R. (2005-A), "The SQL Optimizer Project: \_Method and \_Tree in V9.1", *Proceedings of the 30<sup>th</sup> Annual SAS Users Group International* 

Lavery, R. (2005-B), "An Animated Guide: Hashing in V9.1", Proceedings of the 18<sup>th</sup> Annual Northeast SAS Users Group Conference

#### ACKNOWLEDGEMENTS

The author would like to thank his colleagues at Howard M. Proskin & Associates for their comments, suggestions, and attention during a presentation on this topic as the paper was being written and Mona Kohli for reviewing the paper.

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

## CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Kenneth W. Borowiak Howard M. Proskin & Associates, Inc. 300 Red Creek Drive, Suite 220 Rochester, NY 14623

E-mail: KBorowiak@hmproskin.com EvilPettingZoo97@aol.com

**Data Manipulation** 

### **APPENDIX**

```
/*- Create the data sets used in the examples throughout the paper -*/
/*-- Create the BIG data set --*/
%let dim=10;
data Big(drop=_: sortedby=_null__);
 length Customer ID 8;
 array A[&dim] ;
 do customer=1 to 40000 ;
   do id=1 to 100;
     do _i=1 to &dim;
       a[_i]=id*customer/_i;
       end;
       output;
      end;
  end;
run;
NOTE: The data set WORK.BIG has 4000000 observations and 12 variables.
/*-- Create the SUBSETME data set --*/
%let dim2=%eval(&dim/2);
data SubsetMe(drop=_: sortedby=_null_) ;
 length Customer ID 8;
 array B[&dim2] ;
  c='just text
 do customer=1 to 120000 by 3;
    do id=110 to 1 by -1;
       do _i=1 to &dim2;
         b[_i]=customer+id+_i;
        end;
     output;
      end;
  end;
run;
NOTE: The data set WORK.SUBSETME has 4400000 observations and 8 variables.
/*-- Make VERYSMALL by taking a random sample from SUBSETME --*/
proc surveyselect data=SubsetMe out=VerySmall
                      method=srs n=1000 seed=771133;
NOTE: The data set WORK.VERYSMALL has 1000 observations and 8 variables.
/*-- Make SMALL by taking a random sample from SUBSETME --*/
proc surveyselect data=SubsetMe out=Small
                      method=srs n=800000 seed=771133;
run;
NOTE: The data set WORK.SMALL has 800000 observations and 8 variables.
```