

A Gentle Introduction to Hash Tables

Kevin Martin, Kevin.Martin2@va.gov

Dept. of Veteran Affairs

July 15, 2009

ABSTRACT

Most SAS programmers fall into two categories. Either they've never heard of hash tables or they don't understand the use and significance of these wonderful new features in the SAS9 platform. The intent of this talk is to move you into a third category, those who use/love hash tables. Having used SAS on a daily basis for the last 15 years, I can honestly say these are one of the best tools I've seen for optimizing old programs.

INTRO

So what exactly is a hash table? SAS describes this as a component object that's available inside the parameters of the normal data step. If you've used certain programming languages such as Java and/or JavaScript, you're probably familiar with objects and their associated properties and methods. However, most people don't have this background so there is a learning curve involved with both the new syntax and what can be accomplished with the process. This particular paper will focus on just one example but the references to other papers on the SAS support site (<http://support.sas.com/>) contain many other ways to use these features in different projects.

To peak your interest, here are some things you can find in past SUGI papers.

- 1) The ability to join 2 datasets. No big deal you say. Okay, but this join occurs without having to pre-sort either file!!
- 2) How about joining 3 datasets, again without sorting, all in one step? Oh yeah!
- 3) Conditional joins on those multiple tables? A great example exists on merging in a ZIP9 table first and then using a ZIP5 table if no match could be found in the first attempted (ZIP9) join.
- 4) Recursive Lookups, such as when new ID schematics are generated as a result of an old system converting to a new platform. I haven't

- found a need for something like this (as yet) but could see where it would be useful in certain work environments.
- 5) Outputting N Datasets, where the N is unknown ahead of time. The hash table can dynamically figure out what N happens to be as it processes thru your data file so there's no need for you to figure out what # happens to be when trying the old method of DATA FILE_1-FILE_# ;
 - 6) Outer Joins. The data step is great at inner joins with the MERGE statement and IN data step operators but PROC SQL was the only way to try an outer join (till now!!).

OUR EXAMPLE SITUATION

We had a big program responsible for building 18 different dimension tables for OLAP cube structures on our SQL server boxes. Dimension tables are essentially like SAS formats, where there is a key column containing the unique values mapping into other displayed values in your final product. 1="Yes", 0="No" is a simple example of this.

To optimize each dimension table, the underlying fact table(s) should be queried to determine the unique values within each important demographic column. This is done to keep the available options in the cube from displaying possible demographic values than don't exist in the underlying fact table.

Our project actually has two fact tables, one currently having 35+ million records and the other with 10+ million records.

OLD CODING APPROACH

The original shell used two basic approaches to get the unique listing of values for each fo the 18 variables. (1) PROC SUMMARY NWAY ; CLASS *variable_of_interest* ; OUTPUT OUT=TEMP; RUN; or (2) PROC SORT NODUPKEY ; BY *variable_of_interest* ; RUN; To make matters worse, the original programmer had tried to pre-process the intended fields of interest prior to doing the PROC SORT step and then failed to remove any of these temporary work files with 35M records in them. By the time the shell had ran to completion, the work folder for that one job had over 30 tables in it which totaled over 600 million records. Ouch!

The shell was consistently terminating prior to completion so I was asked for my opinion on how to improve things. The logs I studied always had at least 3 steps left when the server finally ran out of available resources. The constant I/O reads into temporary WORK tables and the PROC SORTs on the huge fact tables were the primary culprits. The resources issue could be addressed indirectly by using PROC DELETE to get rid of each temporary work table when it was no longer needed. But since its common knowledge that PROC SORT is a memory/time hog, I wanted to replace that functionality when I knew a better method existed.

NEW CODING APPROACH

By switching to a hash table approach for each of the two different fact tables, we can greatly reduce the I/O because the process only reads each table a **SINGLE** time while allowing these new tools to dynamically keep track of the unique values for all the columns we need to study. We'll start with a single hash table definition in the first set, to fully describe what each step is doing.

Our fact table contains a 5-digit FIPS code identifier for the home counties of all the patients being discharged from inpatient VA facilities, contained in the column name of HOMEcnty. Obviously, there are more FIPS codes than number of unique counties experiencing an inpatient discharge so we want to determine this smaller set existing in our data file. I will list out the full data step with numbered comments on each line and then describe what each is accomplishing.

```
Data _null_; /* #1 */
  Declare hash county (ordered: 'a'); /* #2 */
  RC=county.definekey( 'homecnty' ); /* #3 */
  RC=county.definedata( 'homecnty' ); /* #4 */
  RC=county.definedone(); /* #5 */
  Do until( eof1 ); /* #6 */
    Set IN.MYFACT1 (keep=homecnty) end=eof1; /* #7 */
    County.replace(); /* #8 */
  End; /* #9 */
  County.output( dataset: 'unique_FIPS' ); /* #10 */
  Stop; /* #11 */
  Run; /* #12 */
```

- 1) While this process will be creating an output file, we'll be doing so via a method of the hash object (see description #10) and thus specify the `_NULL_` step here.
- 2) The hash table only exists for the life of this data step (similar to arrays) but you need the keywords of `DECLARE HASH` to instruct the SAS engine that an internal table in memory is being created...and named `COUNTY` in this instance. It's best to give it a unique name that isn't an exact match of your field name, just to keep things straight. There are also several options/arguments you can specify for this table and we are instructing this step to order the records in an Ascending layout. We do this now so there's no need to use `PROC SORT` later on, in the event we wanted to post-process the results further and merge against an outside file.
- 3) The `RC` field is simply short-hand for `ReturnCode` and could really be any valid SAS variable name. It is one way to allow you to subsequently test the success/failure of a given step if you so desired...but these initial steps are pretty standard so nothing more to say on `RC`. The important part is the syntax to the right of the equal sign. The hash object (`COUNTY`) is calling a method (`DEFINEKEY`) for the field name (`HOME CNTY`) that will be present inside the data set vector. Keys are the unique listings/values that will be kept track of by the process, directly into memory.
- 4) Same `RC` description as in #3. The `DEFINEDATA` method is designating which field(s) should also be stored in the direct memory of the hash table and this example just happens to be using the same field as the key defined in #3. You list the field names inside matching quotes and delimit with a comma if more than one is cited.
- 5) The `DEFINEDONE` method requires no arguments but just informs the step that the definitions of the hash table are complete.
- 6) `DO UNTIL` will iterate at least one time, or until the condition in the parentheses is true. The `EOF1` condition is this situation actually comes from line #7 and relates to when the designated set file is on the last record.
- 7) Specifying a normal `SET` statement to read in a permanently stored SAS file, while only `KEEPing` the one field of interest. Note the field happens to be the same one as specified in the `DEFINEKEY` and `DEFINEDATA` elements of line #3 and #4.
- 8) The first true call to our defined hash table, which occurs with the dot syntax used in object oriented programming. The `REPLACE()` method is being used in this instance (there are several other methods

- available but replace gives us exactly what we need). As the data step moves thru the records of the fact table, the REPLACE() method looks to see if the value read in from the current record happens to exist in the internal memory of the hash table. If it does, no action is taken since the key value already exists. If it doesn't, the value gets added to the COUNTY hash.
- 9) The process in #8 continually repeats itself as the SET statement reads thru the entire designated file, till it gets to the last record. At that point, the temporary EOF1 value is set to 1 and the END statement will stop the processing of the DO loop. Our hash table COUNTY now contains a unique internal listing that's sorted alphabetically with all the FIPS codes contained in our 35+ million fact table.
 - 10) However, the lifespan of the hash table only exists for the duration of the current data step and we need to get these results written out to a normal SAS file so we can use this information in later DATA steps and/or PROCedure calls. The OUTPUT() method of the hash table allows us to accomplish this task quite nicely. Note how you simply specify a name:value pair to the method with the value argument inside matching quotes. The Unique_FIPS data file will have a single field that contains a unique listing of the FIPS values.
 - 11) The STOP statement is a safety mechanism to immediately stop the processing of the current DATA step and move focus to the statements after the end of said step. I feel it's somewhat redundant but every example I've seen contains it so I go with the flow.
 - 12) RUN simply gives closure to the overall DATA _NULL_ step.

FURTHER ENHANCEMENTS

So you now have the basic principles of how to accomplish a keyed index creation. Comparing the run times of this step should be similar too and probably somewhat better than the normal PROC SORT NODUPKEY approach you normally use. So why go to all this trouble you ask? Because we can easily define multiple hash objects within the same instance of a given data step, which allows the system memory to keep track of multiple topics while only reading a source data file a SINGLE time. Taking the original shell and adding a second hash table for Patient Priority code would look something like this. I will include a second data element in the DEFINEDATE() method to provide a syntax example.

```

Data _null_;
  Declare hash county (ordered: 'a' );
  RC=county.definekey( 'homecnty' );
  RC=county.definidata( 'homecnty' );
  RC=county.definidone();
  Declare hash PP (ordered: 'a' );
  RC=PP.definekey('Priority');
  RC=PP.definidata('Priority' , 'e_prio1_8' );
  RC=PP.definidone();
  Do until( eof1 );
    Set IN.MYFACT1 (keep=homecnty priority e_prio1_8 ) end=eof1 ;
    County.replace();
    PP.replace();
  End;
  County.output( dataset: 'unique_FIPS' );
  PP.output( dataset: 'Patient_Priority' );
  Stop;
  Run;

```

So the 2nd hash definition largely mimics the first, with the DEFINEDATA section keeping two data fields in the final output (both those fields are also added to the KEEP statement from the big fact table). I wrote a little more code but I've still only read my source data a single time and now I have two unique listings (Unique_FIPS and Patient_Priority data sets).

Our main project continued in this manner for the remaining fields that required dimension tables, with 18 hash objects for the 35+ million table and 6 hash objects for the 10+ million table. Final wall clock time on the new finished product was 19 minutes as opposed to the 51+ minutes using the original PROC SORT method (which never fully completed anyway). Memory consumption was but a fraction of the original job as well, since each created table only contained a fraction of the records in the original approach.....and the hash tables are all removed from memory at the completion of each data step.

CONCLUSION

While somewhat intimidating, these features are absolutely great for many types of projects. The listed references will hopefully give you ideas on other ways to tackle problems. Merging data without having to sort anything is among my favorite hash table uses and one of the first things mentioned in the Judy Loren paper.

REFERENCES

How Do I Love Hash Tables? Let Me Count the Ways!

(<http://www2.sas.com/Proceedings/Forum2008/029-2008.pdf>)

Judy Loren, SAS Global Forum 2008 Proceedings

Hashing Performance Time with Hash Tables

(<http://www2.sas.com/Proceedings/forum2007/039-2007.pdf>)

Elena Muriel, SAS Global Forum 2007 Proceedings

Search on “hash tables” on the <http://support.sas.com/> site

Search SAS on-line documentation via:

Contents tab → expand SAS Products → expand Base SAS → expand SAS 9.2 Language Reference : Dictionary → expand Dictionary of Component Object Language Elements

Search online for any papers written by Paul Dorfman. Admittedly a hard read but he truly understands how the hash objects work