Paper BB-75

Hash: Is it Always the Best Solution?

David Izrael and Elizabeth Axelrod, Abt Associates Inc.

ABSTRACT

When you get a new hammer, everything looks like a nail. That's how we felt about the Hash object when we started to use it: Wow, this is fantastic - It can solve everything! But… we soon learn that everything is *not* a nail, and sometimes a hammer is *not* the best tool for the job. In SAS® Version 9, direct addressing with the Hash object was introduced, and this enabled users to perform look-ups much faster than traditional methods of joining or merging. Even beyond look-ups, we can now use the HASH object for summation, splitting files, array sorting, and fuzzy matching - just to name a few. What an all-purpose hammer! But… is it always the best tool to use? After a brief review of basic HASH syntax, we will pose some problems and provide several solutions, using both the Hash object and more traditional methods. We will compare real- and CPU–time, as well as programmer's time to develop the respective programs. Which is the better tool for the job? Our recommendations will be revealed through our results.

INTRODUCTION

As a user of SAS, you know there are always multiple solutions to any problem. With the introduction of the HASH object in SAS, we now have even more ways to solve our problems. The authors were curious about how well the HASH object stacks up against some other processing methods, and in this paper we will present some comparisons.

This paper serves as a guideline, or starting place, for how you might want to think about using the HASH object. We do not attempt to teach you how to use the HASH object – instead we refer you to some excellent papers on this topic, referenced at the end our paper. Authors of these papers have inspired us to delve deeper into the HASH object; they have served as our teachers, and we encourage you to explore their papers.

What constitutes a 'best' solution? Is it the one that runs the quickest? Not necessarily. Consider also the time and effort it takes to code, debug, and validate your results. Some solutions are absolutely elegant, and even if they take a bit longer to execute, we might prefer them for their sheer beauty. Determining a 'best' solution is a choice, based on personal preference, programmer skill, platform, etc.

HOW LONG DOES IT TAKE?

All of the examples presented here were run on a 64-bit Windows server, using SAS Version 9.4. Of course, your results may vary, and this emphasizes the non-trivial task of making choices based on a number of factors. When we talk about time, we consider the REAL time (rather than CPU time) to be of most interest to us, since this is how long we spend twiddling our thumbs, waiting for our job to complete. Our jobs share resources with other jobs and system activities, so running the same job on different days or different times of day can yield very different results. A rigorous benchmarking is beyond the scope of this paper; rather, we share with you our findings based on a set of run times that were run on a system with very little other activity and that we feel are representative of the relative performance metrics of each of the methods we used.

BASIC HASH PROGRAM STRUCTURE

In most of our examples, we define the attributes and declare our hash tables using this basic construct:

```
if _n_=0 then
       /* this sets attributes of vars used in hash tables */
       set foo;

if _n_=1 then do;
      /* declare and fill the hash table here – just once */
end;

set foo; /* now we can read the data and do lookups or other hash methods */
```

We start with some straightforward look-up tasks, and then move into fuzzy matching, summation, and splitting files – all of which can be solved with both HASH and other methods.

LOOKUPS

TASK 1 – SIMPLE LOOKUP – TASK DESCRIPTION

Let's start with a simple lookup example: We have a finder file with just over 2.8 million unique IDs (BENE_ID), and each record also has two date fields (DATE1 and DATE2). We want to select records from a file of inpatient claims (with close to 32 million records and over 300 variables) that match on BENE_ID. We also want to grab those two date fields, and append them to each matching record in the claims file. We will compare a traditional merge, PROC SQL, FORMAT lookups, and HASH.

TASK 1 – SIMPLE LOOKUP – SOLUTION USING TRADITIONAL MERGE

In order to use merge, both files must be pre-sorted. In this example, our finder file is already sorted, but we have to sort the big claims file, and this is where we anticipate that MERGE will lose ground.

```
proc sort data=inpat_claims  /* dataset with inpatient claims */
       out=inpat_srt;
       by bene_id;
run;

NOTE: PROCEDURE SORT used (Total process time):
       real time           2:08.59
       cpu time            2:32.19

data task1_merge;
    merge inpat_srt (in=in1 )
          finder     (in=inx);
    by bene_id;
    if inx;
run;

NOTE: DATA statement used (Total process time):
       real time           39.60 seconds
       cpu time            39.43 seconds
```

If our big file was already sorted, merge would be a terrific option, and probably the one that we'd all gravitate towards. But we lost a lot of time with that sort!

TASK 1 – SIMPLE LOOKUP – SOLUTION USING FORMATS

What if we didn't have to sort the big file? We can avoid sorting by using a format with a put statement as a lookup. We create a user-defined format of all the IDs from the finder file, using the CNTLIN= option on PROC FORMAT. We will place our date fields in the format labels.

```
data fmt;
    set finder (rename=(bene_id=start)) end=eof;
    retain fmtname "$getdate";
    length label $20;
    label = put(date1,mmddyy10.) || put(date2,mmddyy10.);
    output;
    if eof then do;
        start=' ';
        label=' ';
        hlo='O';
        output;
    end;
run;

NOTE: DATA statement used (Total process time):
       real time           0.99 seconds
       cpu time            0.95 seconds

proc format cntlin=fmt;
run;
NOTE: PROCEDURE FORMAT used (Total process time):
       real time           4.89 seconds
       cpu time            4.85 seconds
```

Now we can select desired records from our un-sorted claims file, but keep in mind that we also have to convert the character dates back to SAS dates. Here's the code:

```
data task1_fmt;
    set inpat_claims;
    length string $20.;
    string=put(bene_id,$keep.);
    if string>' ';
    date1=input(substr(string,1,10),mmddyy10.);
    date2=input(substr(string,11,10),mmddyy10.);
    output;
    drop string;
    format date1 date2 mmddyy8.;
run;

NOTE: DATA statement used (Total process time):
      real time           47.90 seconds
      cpu time            47.89 seconds                    :
```

Even with the time it took to create the format, we've certainly gained some efficiency by not having to sort our big file. As a side experiment, we were curious how using the data step WHERE option compares to using a subsetting IF. Word on the street is that WHERE is a more efficient construct, but our little experiment doesn't yield a significant difference with the subsetting IF. Here's the code:

```
data task1_fmt_where;
    set inpat_claims (where=(put(bene_id,$keep.)>' '));
    length string $20.;
    string=put(bene_id,$keep.);
    date1=input(substr(string,1,10),mmddyy10.);
    date2=input(substr(string,11,10),mmddyy10.);
    output;
    drop string;
    format date1 date2 mmddyy8.;
run;

NOTE: DATA statement used (Total process time):
      real time           44.39 seconds
      cpu time            43.97 seconds
```

TASK 1 – SIMPLE LOOKUP – SOLUTION USING SQL

PROC SQL turns out to be a rather nice solution, and as with our format solution, we do not have to pre-sort our big file. We use the _METHOD option [8], which shows us what SAS is doing behind the scenes, in the notes below.

```
proc sql _method;
     create table task2_sql as
        select i.*, date1, date2
        from   finder      as f,
               inpat_claims   as i
        where  f.bene_id = i.bene_id
        ;
NOTE: SQL execution methods chosen are:

sqxcrta
     sqxjhsh
        sqxsrc( WORK.INPAT_CLAIMS(alias = I) )
        sqxsrc( WORK.FINDER(alias = F) )
quit;

NOTE:  PROCEDURE SQL used (Total process time):
       real time           47.55 seconds
       cpu time            47.39 seconds
```

The information from the _METHOD option shows us that PROC SQL is actually using a HASH algorithm to do the join (SQXJHSH). And indeed – it's quite an efficient process.

TASK 1 – SIMPLE LOOKUP – SOLUTION USING HASH

Finally… let's see how the HASH object stacks up. Using HASH to lookup our IDs, select matching records, and grab those two date fields, we can use our un-sorted claims file.

```
data task2_hash;
    if _n_=0 then
       set finder;
    if _n_=1 then do;
        declare hash f (dataset: "finder");
        f.definekey('bene_id');
        f.definedata('date1','date2');
        f.definedone();
        call missing(bene_id,date1,date2);
    end;
    set inpat_claims;
    rc = f.find();
    if rc=0;
    drop rc;
run;

NOTE: DATA statement used (Total process time):
      real time           42.80 seconds
      cpu time            42.29 seconds
```

Not too shabby! The Hash object within the data step does a great job. Although the data step using HASH takes a bit longer than the one using MERGE, once we add in the time used for the PROC SORT, our HASH solutions wins. This is one reason it is such a great solution for lookups: No need to pre-sort your big data files! You'll notice that using the FORMAT as a lookup tool took about the same amount of time as our HASH lookup. But we had to spend a bit of programmer time coding to build the format, and then to convert the character dates back to SAS dates.

HASH, SQL, and formats with a put statement all gave equivalent performances. But merge is a definite loser, due to the necessary step of pre-sorting the big data file. Merge therefore becomes extremely inefficient if you have to do lookups from multiple files, as in the next example.

TASK 2 – MULTIPLE LOOKUPS – TASK DESCRIPTION

In this example, we want to:

- identify which claims to keep based on our finder file, as in task 1;
- flag records that have a primary diagnosis code that match an external lookup file of diagnosis codes;
- flag records that have provider IDs that match an external lookup file of provider IDs.

If we solve this using merge, we'll have to sort our large file 3 times – by BENE_ID, then by primary diagnosis code, and then by Provider ID. Surely there's a better way! Let's start with a hash solution.

TASK 2 – MULTIPLE LOOKUPS – SOLUTION USING HASH

Here we create three hash tables: one for the finder file (to determine which records to keep and to grab the date vars), one for the diagnosis code list and one for the provider list. We can achieve everything we want - all in one data step – and no pre-sorting is necessary.

```
data task2_hash;
    if _n_=0 then do;
       set finder;
       set diag_list;
       set prov_list;
    end;
```

```
    if _n_=1 then do;
        declare hash f (dataset: "finder");
        f.definekey('bene_id');
        f.definedata('date1','date2');
        f.definedone();
        call missing(bene_id,date1,date2);

        declare hash d (dataset: "diag_list");
        d.definekey('prim_dx');
        d.definedone();
        call missing(prim_dx);

        declare hash p (dataset: "prov_list");
        p.definekey('provider');
        p.definedone();
        call missing(provider);
    end;
    set inpat_claims;
    rc_ben = f.find();
    if rc_ben=0;

    rc_dx = d.find();  /* lookup the dx code */
    found_dx=(rc_dx=0);

    rc_prov = p.find(); /* looup the provider id */
    found_prov=(rc_prov=0);
run;

NOTE: DATA statement used (Total process time):
      real time           50.04 seconds
      cpu time            49.42 seconds
```

Wow! Using the hash object to do multiple lookups in one data step is fantastic! It's fast and elegant. *And* it's intuitive: Read the file once, decide whether or not to keep the record, then make some other decisions about a few variables while we're at it. Our program is easy to follow and we don't have to keep track of a bunch of intermediate files, sorted every-which-way. It is hash's ability to swiftly solve problems like this that makes us absolutely swoon over this method.

TASK 2 – MULTIPLE LOOKUPS – SOLUTION USING SQL

We really wanted to see how SQL would perform on this problem, but after many feeble attempts at crafting the SQL code (including a terrible cartesian failure that brought our system to its knees), we reached out to one of our SQL experts who provided us with this code:

```
proc sql _method;
    create table task2_sql as
    select i.*,
        case
            when ( i.prim_dx in ( select prim_dx from diag_list ) )
                then 1
                else 0
            end as dx_found,
        case
            when ( i.provider in ( select provider from prov_list ) )
                then 1
                else 0
            end as prov_found,
        f.date1,
        f.date2
    from inpat_claims i, finder f
    where i.bene_id = f.bene_id;
quit;


SQL execution methods chosen are:
```

```
sqxcrta
    sqxjhsh
        sqxfil
            sqxsrc( WORK.INPAT_CLAIMS(alias = I) )
        sqxsrc( WORK.FINDER(alias = F) )
    sqxsubq
        sqxsrc( WORK.DIAG_LIST )

SQL subquery execution methods chosen are:
    sqxsubq
        sqxsrc( WORK.PROV_LIST )
     quit;

NOTE: PROCEDURE SQL used (Total process time):
       real time           1:54.94
       cpu time            1:54.78
```

Ta-da! As we can see from the notes generated from the _METHOD option, SQL is using a hash algorithm, although this step does take more than twice as long as our data step using hash.

LAST WORDS ON LOOKUPS

In all of the examples above, the times shown are based on runs we did very early in the morning – we wanted to minimize the influence of other interference (especially contention for I/O resources) and system processes. . We believe this is why the REAL times are so close to the CPU times. As stated in our introduction, your mileage will surely vary – even ours did when we ran these same jobs at other times. But our overall findings were consistent: Using just one lookup table, hash, SQL, and format solutions were pretty equivalent. With multiple lookup tables, a hash solution for us was a clear winner – easier to code up, fast, elegant, and just plain satisfying.

Note that the SQL code presented for multiple lookups works only for our specific purpose of *flagging*. To actually pick up variables from a look up table, the SQL code would need to be significantly different. The beauty of hash, however, is that we'd be able to accomplish this modification with minimal editing to our hash code just adding the names of desirable data into respective *definedata()* and *call missing ()*

Let's ramp up the complexity of our problems, and move on to other uses of hash beyond lookups.

FUZZY MATCHING

We have two data sets (F1 and F2), with roughly one million observations each, and our task is to link these files by performing some fuzzy matching on last name, first name, and date of birth. We'll solve this problem using both hash and SQL. In both of these we will use the SOUNDEX function to match the name variables (although in our actual work we used a more complicated cost function), and we'll consider dates of birth to match if they are within 7 days of each other.

TASK 3 – FUZZY MATCHING – SOLUTION USING SQL

```
proc sql;
create table task3_sql as
    select *
    from  F1 (keep = first_name1 last_name1 birth_date1) ,
          F2 (keep = first_name2 last_name2 birth_date2)
    where soundex(first_name1) = soundex(first_name2) and
          soundex(last_name1)  = soundex(last_name2)  and
           abs(birth_date1 – birth_date2)<7 ;
 quit;


NOTE: PROCEDURE SQL used (Total process time):
       real time           2.24 seconds
       cpu time            2.24 seconds
```

For SQL to solve this problem, it compares each record from F1 to each record in F2. We can use the same logic in a data step using the hash iterator.

TASK 3 – FUZZY MATCHING – SOLUTION USING TWO HASH TABLES

To solve this problem using a hash method, we will load each file (F1 and F2) into its own hash table. Rather than

using hash's direct access capability, we will use the hash iterators to walk through each record from both tables, comparing our keys with the same fuzzy matching logic we used above.

```
data task3_hash;
     if _n_ = 0 then  do;
          set F1 (keep = first_name1 last_name1 birth_date1) ;
          set F2( keep = first_name2 last_name2 birth_date2) ;
     end;

     DECLARE hash F1( dataset:"F1(keep=first_name1 last_name1 birth_date1)",
          MULTIDATA:"Y",
          HASHEXP: 15);

     F1.DEFINEKEY ('FIRST_NAME1');
     F1.DEFINEDATA('first_name1', 'last_name1', 'birth_date1' );

     DECLARE hiter hi_f1("F1");
     F1.DEFINEDONE ( );

     DECLARE hash F2(dataset:"data.F2
          (keep=first_name2 last_name2 birth_date2 )",
          MULTIDATA:"Y", HASHEXP: 15);

     F2.DEFINEKEY ('FIRST_NAME2');
     F2.DEFINEDATA ('first_name2', ' last_name2',  'birth_date2' );
     DECLARE hiter hi_f2("F2");
     F2.DEFINEDONE ( );

     rc1 = hi_f1.first();  /* comment needed */
     rc2 = hi_f2.first();

     do while (rc1=0);     /* cycle through each … */
          do while (rc2 = 0) ;  /* cycle through each … */
             if soundex(first_name1) = soundex(first_name2) and
                soundex(last_name1) = soundex(last_name2)   and
                abs(birth_date1 – birth_date2)<7 then
                    output;
                rc2 = hi_bop.next() ;
          end;
          rc2 = hi_f2.first();
          rc1 = hi_f1.next ();
     end;
stop;
run;
```

We create two hash tables – F1 and F2 based on our matching data sets. We use hash iterators – HI_F1 and HI_F2 – to cycle through and compare each record from F1 with each record from F2, and perform our fuzzy comparisons. We expected that the time used by the hash method would be less than, or at least comparable to the SQL method. To our surprise, this was not the case. After almost five minutes of processing time, we decided to kill the job.

```
ERROR: User asked for termination
NOTE: The SAS System stopped processing this step because of errors.
NOTE: SAS set option OBS=0 and will continue to check statements.
      This might cause NOTE: No observations in data set.
NOTE: DATA statement used (Total process time):
      real time           4:59.43
      cpu time            4:43.31
```

Running this code on smaller data sets (1,000 observations each) performed instantly. We found that the interaction of two hash tables when performing a comparison of each record from the first one to each record from the second one appears to be crucially sensitive to their sizes, although we are not sure why this is. We therefore concluded that for our fuzzy matching task, PROC SQL does a good job and we would not recommend deviating from it.

SUMMATION

For our next task, we will sum three variables from our DRUGS data set, which contains 150 million observations. We'll sum *weight, daysupply,* and *dose*, by several other categorical variables *(outlet_state, main_cat)*.
We compared summation of variables using PROC SUMMARY and using hash tables.

TASK 4 – SUMMATION – SOLUTION USING PROC SUMMARY

```
proc summary data = drug (keep = outlet_state main_cat weight daysupply dose)
                  nway noprint ;
     class  outlet_state main_cat;
     var    weight daysupply dose ;
     output out = task4_summary (drop = _:)
            sum =  ;
run ;

NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           6:15.82
      cpu time            1:18.81
```

TASK 4 – SUMMATION – SOLUTION USING HASH

We can use hash to sum variables by loading the file with the keys and the variables to be summed into a hash table, accumulating sums much like we do in a dataset with a retain, and outputting just the resulting sums.

```
data _null_ ;
     if 0 then
          set drugs (keep = outlet_state main_cat  weight daysupply dose);

     dcl hash hh (hashexp:20, ordered: 'a') ;
     hh.definekey ('outlet_state', 'main_cat' ) ;
     hh.definedata ('outlet_state', 'main_cat', 'sum_weight', 'sum_daysupply',
                    'sum_dose') ;
     hh.definedone () ;

     do until (eof) ;
          set drug (keep = outlet_state main_cat weight daysupply dose
                    where= (outlet_state ne ' ' and main_cat ne ' '))
                    end = eof ;

          if hh.find () ne 0 then do;
             sum_weight = 0;
             sum_daysupply=0;
             sum_dose=0;
          end;

          sum_weight + weight;
          sum_daysupply + daysupply;
          sum_dose + dose;

          hh.replace ();
     end ;
     rc = hh.output (dataset: 'summmary') ;
run ;

NOTE: DATA statement used (Total process time):
      real time           9:10.35
      cpu time            1:27.18
```

We'll create a hash table with the keys *outlet_state* and *main_cat*. The data part of the hash table includes the fields to be summarized *(weight, daysupply, and dose)* as well as the fields that accrue the sums *(sum_weight, sum_daysupply, and sum_dose).*

The HH.FIND() method searches the HH hash table using current values of our keys: *outlet_state* and *main_cat*. If the current key is not found, the variables *sum_weight, sum_daysupply,* and *sum_dose* are set to 0 and new keys (*outlet_state* and *main_cat*) are added to HH. The values of *weight, daysupply, and dose* are added to the cumulative variables and new cumulative record replaces the previous one in HH by the method HH.REPLACE().

PROC SUMMARY clearly works better for this task, and users who are a dab hand at PROC SUMMARY will certainly benefit from using it.

SPLITTING SAS DATA SETS

Suppose we want to split our DRUGS data set by state, writing out a SAS file for each value of *outlet_state*. We do not know beforehand what states are in the data sets, nor do we know if the data set is sorted by *outlet_state*.

We will solve this problem using a macro to cycle through some SQL and traditional data step code, and then by using a 'hash of hashes' method suggested by Dorfman in [2].

TASK 5 - SPLITTING SAS DATA SETS – SOLUTION USING MACRO, SQL, AND DATA STEP

Using PROC SQL, we first put the list of existing states into a macro variable. We'll reference this macro later in our data step, where we write an observation to a data set that matches the value of *outlet_state* in a given observation.

```
%macro split;

     proc sql noprint ;
          select distinct outlet_state
              into: statelst
                    separated by ' '
          from drugs (keep = outlet_state);
     quit ;

     data &statelst;
          set drugs (keep=outlet_state main_cat weight daysupply dose) ;
          %do i=1 %to &sqlobs;
              %let curst = %scan(&statelst, &i, %str ( ));
              %if &i = 1 %then
                    if outlet_state = "&curst" then output &curst;
              %else
                    else if outlet_state = "&curst" then output &curst;
          %end;
     run;
%mend;

%split;

NOTE: The SAS System used:
      real time           9:05.17
       cpu time           2:31.74
```

Here's how this macro works: In the first step, we use SQL to grab all values of *outlet_state* and place them in a macro variable called *statelst*. The macro variable &sqlobs is automatically created in this step, providing us with the number of states in our list. In the next data step, we use the macro variable *statelst* to generate the names of our output files. We read the DRUGS file and write out observations to the respective output dataset, based on the current value of *outlet_state.*

TASK 5 - SPLITTING SAS DATA SET USING HASH OF HASHES

The 'hash of hashes' provides a very interesting solution here. We encourage the reader to refer to Dorfman's paper [2] for a full description of how this works. But the basic idea is that it enables you to use a *.definedata()* where one of the variables is itself a hash table! As also demonstrated in [2], we can build a new instance of a hash table with the same name, using the operator *_new_.*

```
data _null_ ;
     dcl hash states (ordered: 'a') ;
     dcl hiter his('states' ) ;
     states.definekey('outlet_state' ) ;
     states.definedata('outlet_state', 'states_inst' );
     states.definedone() ;

     dcl hash states_inst () ; *** will have instances of all states data;
```

```
      do _n_ = 1 by 1 until ( eof ) ; *** we use _n_ as counter to use it as
                                        an augmentation for key in HH;

          set drugs(keep = outlet_state main_cat  weight daysupply dose)
                     end = eof ;
          if states.find () ne 0 then do ;
              states_inst = _new_ hash (ordered: 'a');
              states_inst.definekey ('outlet_state', '_n_');
              states_inst.definedata ('outlet_state', 'main_cat', 'weight',
                                        'daysupply', 'dose') ;
              states_inst.definedone ();
              states.replace ();
          end ;
          states_inst.replace();
      end;

      rc = his.first();

      do while (rc=0);
          states_inst.output (dataset: 'out'|| outlet_state );
          rc = his.next();
      end ;

      stop;
run;

NOTE: The SAS System used:
      real time           7:13.04
      cpu time            3:05.93
```

In this example, the hash table *states* is loaded and checked for existence of a current key which is *outlet_states* using the method *states.find.*

The hash *states_inst* will hold the instances, each of which contains the data for a single state. If *states.find()* method does not find the current state in *states* hash it loads it into *state* using *states.replace();* a new instance of "hash-data" *states_inst* is loaded with the newly ~~came~~ read state data using *states_inst.replace().*

If *states.find()* does find the current state in *states* hash, it loads the current data into the already existing instance of *states_inst* hash using again *states_inst.replace().* After all observations of *drugs* have been read, each record of *states* hash of hashes will contain one state (*outlet_state*) - 51 in total if there are all states in the database - and *states_inst* hash will represent the 51 instances with all their records . As for the *his* hash iterator, the *his.next*() method will point to the next record in *states (i.e.,* next state) and, therefore, at the next instance of the *states_inst* hash table. With this understanding, we use method *states_inst.output* to write the data sets for each *outlet_state.*

One can see almost the same CPU time (traditional method is slightly ahead) and some difference in real time – 9 minutes vs. 6 - between our first solution to the problem using traditional SAS methods, and the hash of hashes method.

The hash solution is elegant, and provides aesthetic enjoyment for the programmer's soul. Having said that, is this approach the first solution most of us would grab from our available tool belt? Probably not. Most of us probably deal with this stuff rather rarely, so one has to either recollect the syntax, search for previous programs, or google for similar solutions and then figure out how to adjust them to the task at hand. And this takes *your* time, which is far more valuable than computer time (setting aside, of course, the extreme cases when a program runs for 1 hour while an alternative one runs for 3 days). So, be aware of this when weighing the merits and effectiveness of your options. Of course, it is often worth the time invested to learn a new approach –that's how *we* learned the hash method!

CONCLUSION

We looked at five programming problems, comparing various solutions to a hash solution. So, how does hash fare? Is it always the best solution? Well, … It depends!! For simple single-table lookups, hash was pretty equivalent to some of other methods we tried. For multi-table lookups hash was the winner. But in all lookup situations, the authors prefer methods that don't require pre-sorting, and we particularly like using hash solutions for their elegance

and ease of coding. In most of our tasks, PROC SQL is generally comparable with the hash solution – and this should come as no surprise, since we showed that for lookup operations, PROC SQL uses the hash method itself. As far as the other tasks are concerned, we see either negligible advantage of the hash method or clear advantage of traditional methods and procedures over the hash.

Of course, in all cases, one must consider the programmer's time required to develop, debug, and test the program. If there is no great performance advantage to one method over another, the deciding factor should probably be how skilled you are with the method you choose. You should be able to write the code, validate your results, and understand what's really going on.

And speaking of time, we remind you of the elusive nature of REAL time – how greatly it varies depending on many factors, especially on a shared environment – time of day, day of the week, contention with other resources and system activities. During the development of this paper, we also learned some surprising things about how the *sequence* of your data steps can have a huge impact on real time. Although beyond the scope of this paper, we consider visiting this interesting topic in the future.

The hash object can solve problems that have been traditionally addressed by SQL or traditional data step code. If you are not yet familiar with how hash can address these problems, we encourage you to further explore this valuable tool. We find it to be elegant and satisfying!

ACKNOWLEDGDEMENTS

The authors wish to thank Jack Shoemaker and Paul Grant for providing us with valuable input and technical guidance.

REFERENCES

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

2. Paul M Dorfman, Koen Vyverman. 2005. "Data Step Hash Objects as Programming Tools." *Proceedings of the SAS Global Forum 2005 Conference.*

3. Eberhardt, Peter. 2011. "The SAS® Hash Object: It's Time to .find() Your Way Around." *Proceedings of the 2011 SAS Global Forum Conference.*

4. Loren, Judy. 2008. "*How* Do I Love Hash Tables? Let Me Count The Ways!" *Proceedings of the 2008 SAS Global Forum Conference.*

5. Secosky, Jason and Janet Bloom. "Getting Started with the DATA Step Hash Object." Available at http://support.sas.com/rnd/base/datastep/dot/iterator-getting-started.pdf

6. Robert Ray and Jason Secosky. 2008. "Better Hashing in SAS® 9.2." *Proceeding of the 2008 SAS Global Forum Conference.*

7. Izrael, David. 2012. "Working with a Large Pharmacy Database: Hash and Conquer." *Proceedings of NorthEast SAS User Group conference, Baltimore, MD, November 2012.*

8. Shipp, Charles Edwin and Kirk Paul Lafler. 2013. "Exploring the PROC SQL _METHOD Option." *Proceedings of SAS Global Forum 2013 Conference.*

CONTACTS

David Izrael
Abt Associates Inc.
Phone (w): 617.349.2434
E-mail: david_izrael@abtassoc.com

Elizabeth Axelrod
Abt Associates Inc.
E-mail: elizabeth_axelrod@abtassoc.com