

Join, Merge or Lookup? Expanding your toolkit

Judy Loren, Health Dialog, Portland, ME

Sandeep Gaudana, Health Dialog, Portland, ME

ABSTRACT

People who are learning SAS® often ask: Is the DATA step or PROC SQL better? The answer depends on several factors. This paper will provide guidance to help new-to-intermediate users make an informed choice in merge/join situations. The syntax and the theory underlying 5 different techniques for bringing together data from 2 (or more) datasets will be covered. The techniques to be included are: sort and merge; SQL; user-defined formats; hash tables; and indexed lookup. The methods will be compared for suitability in a variety of specific merge scenarios, and benchmarks of resource use will be shown. A macro that can be used to develop similar benchmarks at any site on any data is included. This should be a handy resource for anyone who is often faced with new merge challenges.

INTRODUCTION

The “best” join technique for a given scenario depends primarily on your situation and your goals. Is this a one-time join done for analysis or exploration? Or is it part of a production stream that is executed daily? Are you constrained by clock time, development time or access to computer resources? Can you afford to trade CPU utilization for space savings, or vice versa? Do you have knowledgeable programmers to devote to development or are you relying on SAS neophytes?

In this paper, we provide an overview of several join techniques with enough syntax to give the reader some starting code. We show some benchmarks that we produced that might help others in making the decision about which technique to apply in a given situation. We also share a macro we developed that will allow you to test various techniques on your site using your data and easily compare resource usage.

TECHNIQUE: SORT-AND-MERGE

The sort-and-merge technique consists, at its most elementary level, of two **proc sorts** and a **data** step with a **merge** statement. Consider the following case: You have a small dataset of hospital admission records and you want to look up information about the patient in a large dataset of all insured members. If you use the sort-and-merge technique, you will do the following;

```
proc sort data = liba.members;
  by member_id;
run;

proc sort data = libc.admissions;
  by member_id;
run;

data both;
  merge liba.members(in=in1)
        libc.admissions(in=in2);
  by member_id;
  if in1 and in2 then output;
run;
```

Note the use of **in1** and **in2** to control which records are kept, based on which datasets contain data about that member.

One or both of the datasets may already be sorted by the necessary variable (in this case, **member_id**). If SAS

can detect that the current sortation of a dataset matches that required by the **by** statement, it will avoid devoting resources to re-sorting it.

ADVANTAGES AND DISADVANTAGES

The sort-and-merge technique is straightforward and takes advantage of common tasks that almost any SAS programmer will be familiar with. If additional processing of the result set is necessary, the data step can include that as well. However, if one or both of the datasets is very large and the sort required for the merge is different from the sort ultimately needed for that dataset, this technique can be expensive, sometimes prohibitively so.

TECHNIQUE: PROC SQL

Proc SQL was added to SAS in Version 6. This procedure makes it easy for users to transition from other database tools to SAS. SAS Institute has been and remains committed to ANSI standard SQL, to the extent that certain SAS-specific DATA step functions don't work in SQL processing (e.g., the **sum(of x1-x9)** functionality that sums the elements of an array).

To accomplish the same join described above using SQL:

```
proc sql;
  create table both as
  select a.*
         ,b.*
  from liba.members a
       join
       libc.admissions b
  on a.member_id = b.member_id;
quit;
```

This join is an “inner join.” Provided at least one of the datasets has only one record per match key, this results in the same dataset as the **data** step merge above. Only records where the **member_id** exists on both datasets will be kept in dataset **both**. Variations on this such as **left join**, **right join** or **outer join** allow you to keep only **member_ids** in the left dataset (**members**), only **member_ids** in the right dataset (**admissions**) or all **member_ids** in either dataset, respectively.

ADVANTAGES AND DISADVANTAGES

One advantage of proc SQL has been mentioned already: if you already know some SQL as a result of using another database tool, you can jump into SAS with proc SQL fairly easily. The different flavors of SQL that exist in different products can create bumps in the road as you move from one brand to another, but the way of thinking about data in various locations is the same. Another advantage is that SQL can accomplish a lot in one step. The user codes the result set desired, not the steps to execute. In the above example, because there is no need to code the proc sort explicitly, some people would say that SAS doesn't sort the datasets. In fact, though, behind the scenes, some way of matching the datasets has to happen. This leads to one disadvantage of proc SQL.

SQL was written specifically to avoid the user having to (or getting to, depending on how you look at it) determine the right way to do the job of matching data. Each database product that implements SQL, and SAS is no exception, has to have an optimizer: a program whose job is to decide how to accomplish the end result specified by the user. Optimizers work very well much of the time. But they have only certain information available to them, so they will sometimes make the wrong decision. Also, the optimizer will make the decision at run time. If you have SQL in production code, it may run very well most of the time but occasionally change in performance. Much more information about the SQL optimizer is available in Russ Lavery's SUGI 30 paper (see References section of this paper).

TECHNIQUE: USER-DEFINED FORMATS

User-defined formats are the duct tape of SAS. They have loads of uses and do a lot of things well. You may be familiar with user-defined formats as a way of printing descriptions rather than codes, or as a way of grouping continuous numeric variables into categories for display or analysis. But user-defined formats can also be used as a way of joining data. Consider once again the case where you have a list of **member_ids** and you want to

extract their records from a large member dataset. To avoid the cost of sorting the large dataset, you can create a format that maps all the desired **member_ids** to the same value (for example, the word “YES”) and all other **member_ids** to a different value (perhaps the word “NO”). Then you process the large dataset in its existing sort and choose records based on what value the **member_id** maps to when you apply your format.

Enhancing this example, suppose that in addition to selecting members you want to keep one piece of data from the small dataset. Let’s use admission date as an example. In this case, instead of mapping all the desired **member_ids** to the word “YES,” you map each desired **member_id** to its admission date. You can still accomplish the selection by mapping **other** to “NO”.

Here is the code:

```
** Create a dataset to feed proc format **;
data key;
  set libc.admissions(rename=(member_id = start));
  retain fmtname '$key';
  label = put(admit_dt, yymmdd10.);
  hlo = ' ';
  output;
  if _n_ = 1 then do;
    start = 'other';
    hlo = 'o';
    label = 'NO';
    output;
  end;
run;

** Create the format $key in the work library **;
proc format cntlin=key;
run;

** Process the large dataset choosing only desired members **;
data both;
  set liba.members;
  where put(member_id,$key.) ne 'NO';
  admit_dt = input(put(member_id,$key.), yymmdd10.);
run;
```

Once again, only **member_ids** found on both datasets will be kept in the dataset **both**. The SAS date value **admit_dt** on the **admissions** dataset is translated to a label for purposes of the format, then translated back into a SAS date value for **both**.

ADVANTAGES AND DISADVANTAGES

The main advantage of this technique is the efficiency gain in avoiding the sort of the large dataset. In this case we are selecting certain records from the large dataset, as well as appending a piece of information to each selected record. The format technique also works very well for mass lookup situations, where the sole reason for joining two datasets is to lookup information about one variable on the main dataset from a separate dataset. One example of this might be looking up the specialty of a doctor when processing a large volume of health care claims. Rather than sorting the entire claims dataset by the provider ID in order to merge it with a provider dataset to pick up the specialty code, you can put the provider ID and specialty into a format and look it up during an existing pass of the claims data.

The format technique is limited to inner or one-sided joins. In the example shown here, like the previous 2 illustrations, the result set **both** contains only records where the **member_id** exists on both **admissions** and **members**. Unlike the earlier techniques, however, the format technique does not support keeping all the **admissions** regardless of whether the member is on the **members** dataset.

Another constraint on the format technique is size. Like the hash technique below, since formats must fit into existing memory, there is a limit to the size of dataset a format can handle. Also, using the format technique in this way requires passing the entire large dataset. Only the indexed access technique (shown below) allows you to avoid that.

TECHNIQUE: HASH OBJECTS

Hash objects were introduced in Version 9 of SAS. To learn more about these amazingly useful tools, we highly recommend that you track down one or all of Paul Dorfman's papers about them (see the References section of this paper for one example). Here we will present the basic code needed for a join, but there is much more that can be done with hash objects.

Hash objects or hash tables are **data** step constructs. Much like arrays, they exist only for the duration of the **data** step in which they are created. Like formats, they are loaded into memory. Combine this with the hashing strategy, and you get lookups that are extremely fast.

Here is the code to accomplish the merge in question:

```
data both(drop=rc);
  length member_id $ 12 admit_dt discharge_dt 8;

  declare associativearray hh ();

  rc = hh.DefineKey ('member_id');
  rc = hh.DefineData ('admit_dt', 'discharge_dt');
  rc = hh.DefineDone ();

  do until (eof1) ;
    set libc.admissions end = eof1;
    rc = hh.add ();
  end;

  do until (eof2) ;
    set liba.members end = eof2;
    rc = hh.find ();
    if rc = 0 then output;
  end;
stop;
run;
```

ADVANTAGES AND DISADVANTAGES

Hash tables are as fast as formats, and they allow joins between datasets where you want to keep multiple data fields from both datasets. They do not require that either dataset be sorted. They take advantage of memory, which is often abundant.

But memory is not unlimited. Since hash tables must fit into existing memory, huge joins may not work using the hash technique.

In this version of the hash technique, as in all the examples above, only **member_ids** found on both datasets will be kept in the dataset **both**. All records from the **members** table can be kept if desired, but the use of hash tables exhibited above does not easily keep records from **admissions** that are not found on **members** (in other words, it is not set up for outer joins).

Hash tables are new to SAS users, and more complex to code than the straightforward **proc sort** and **merge** we started out with. You also have to be careful that variable lengths are established correctly.

TECHNIQUE: INDEXED LOOKUP

Indexed lookup refers to a technique which simulates direct access to a specific record buried somewhere within a large dataset. It is implemented through an index structure, which is a second dataset sorted by the indexed variable(s) containing a pointer to the relative location of the record in the main dataset. Direct access is obtained using the **key=** option on the **set** statement during a **data** step. As each record is read from a small dataset in a **data** step, SAS looks at the value(s) of the variable(s) on which the large dataset is indexed, finds the corresponding entry in the index, then pulls in the record from the large dataset that contains the desired value(s) of the indexed variable(s). If there are multiple records with the same value(s) of the indexed variable(s) on the large dataset, other options specified on the **set** statement determine which record is returned.

Here is the code:

```
*establish the index on the large dataset;
proc datasets lib=liba;
  modify members;
  index create member_id;
quit;

*process the small dataset and pull in matching records from large dataset;
data both;
  set libc.admissions;  *This is the main set statement controlling the data step;

  set members
    key= member_id / unique;

  if _iorc_ ne 0 then delete;  *_iorc_ is non-zero when lookup unsuccessful *;

  _error_=0;                 *Failed lookups set _error_=1;
                             *Reset to avoid data dump in log:
run;
```

To use this technique, the main dataset must be indexed on one (or several) key variable(s). If this index already exists at the time you run your join, there is no need to re-create it (and in fact an error will be generated if you try). Once the index exists, the dataset can be accessed using **key=** on the **set** statement. In the **data** step shown here, the main **set** loop points to the **admissions** table. Records from the **admissions** table will be brought in one at a time. Each time, the value of **member_id** on the **admissions** record is used to extract a record from the **members** dataset via the index. The **unique** option specifies that there is only one record on **members** for each value of the index variable, and multiple attempts to lookup the same value should return the same record. Without this option, multiple attempts to find the same value will search for sequential records. Second and subsequent attempts at the same value will fail if no additional records with that value are found.

This code is set up to delete records from the **admissions** dataset if the **member_id** is not found on the **members** table. This is not necessary; all records from the **admissions** dataset can be kept. The indexed lookup technique does not provide for moving all records from the **members** dataset to **both**.

ADVANTAGES AND DISADVANTAGES

Indexed lookup is ideally suited for situations where you need to do multiple very small lookups against a very large table in between updates to the large table. The main cost is in creating and maintaining the index on the large table. This overhead can be well-justified by the quick turnaround on multiple single (or small volume) lookups. The disk access required by first reading the index dataset and then finding the record on the large dataset makes this technique perform with decreasing efficiency as the number of values requested increases.

PERFORMANCE TESTING

The question of which technique to apply requires the coder to consider several factors. Some of those factors have to do with how the technique will perform. Until it is executed, it is impossible to know exactly what computer resources will be required (How long will the job take? How much Memory is needed?) But guidelines can be developed that allow the user to estimate the relative resource utilization of various techniques under different scenarios.

In the process of developing these benchmarks for our site and for this paper, we developed a macro that runs any technique on a series of datasets to compare CPU time, real time and memory utilization. This macro (shown in Appendix A) does the following:

Starting with a large dataset specified by the user, the macro generates a series of samples of that large dataset to use as the small dataset in a join. The smallest desired size, the largest desired size and the increment for each size in between are specified by the user as well. For example, if the user wants to test a technique using a sample dataset of sizes ranging from 5% to 45% of the full dataset provided, the parameters 5 and 45 are supplied, as well as an increment such as 10 so that samples will be generated of size 5%, 15%, 25%, 35% and 45%.

These sample datasets are made available iteratively, under the name ***small_dataset***, to a program specified by the user. The user feeds the macro the path and name of the program, which the user has written to join ***small_dataset*** with another dataset using whichever technique the user wants to test.

The macro then runs the user-supplied program on each of the samples, writing the log to a name also specified by the user. The log will be used as input to the **%logparse** macro. The **%logparse** macro will extract the performance measures from the log and store them in a dataset which can be analyzed and reported (an example is shown in the next section). See Mike Raithel's SUGI 30 paper, cited in the References section of this paper, for details on how to use the **%logparse** macro and its associated macro **%passinfo**. These macros are available from SAS Institute and are experimental in Version 9.1. See the References section for the URL from which to download the **%logparse** macro.

An example of the application of the macro, including a possible program the user might supply, is shown in Appendix B. Note that this macro and associated programs were developed and tested for a Windows environment. The logic and most of the SAS code would apply to other OS's, but would need to be adjusted.

RESULTS

To develop a systematic set of benchmarks, we ran the performance macro on all the techniques described above using small datasets ranging in size from 0.01% of a large dataset to 95%. We did this under several scenarios: large dataset not sorted or indexed; large dataset indexed; large dataset sorted. We also ran separate benchmarks for single variable key and multi-variable key joins. Space constraints prevent us from listing all the results from all of our tests. They would be of limited value, since they were all run on the same system. The configuration of each system and OS would cause results to fluctuate from site to site. We will share some of our results here to show how the process works and what kind of differences you might see, and encourage you to try the same approach at your site.

These benchmarks were done on a server with 2 Intel® Xeon™ MP CPUs (3.00 GHz) with 3.8 GB RAM running Microsoft® Windows® 2000 5.00.2195 Service Pack 4 with 2.40 TByte attached disk for permanent storage and a separate 273 GByte capacity attached disk for work space.

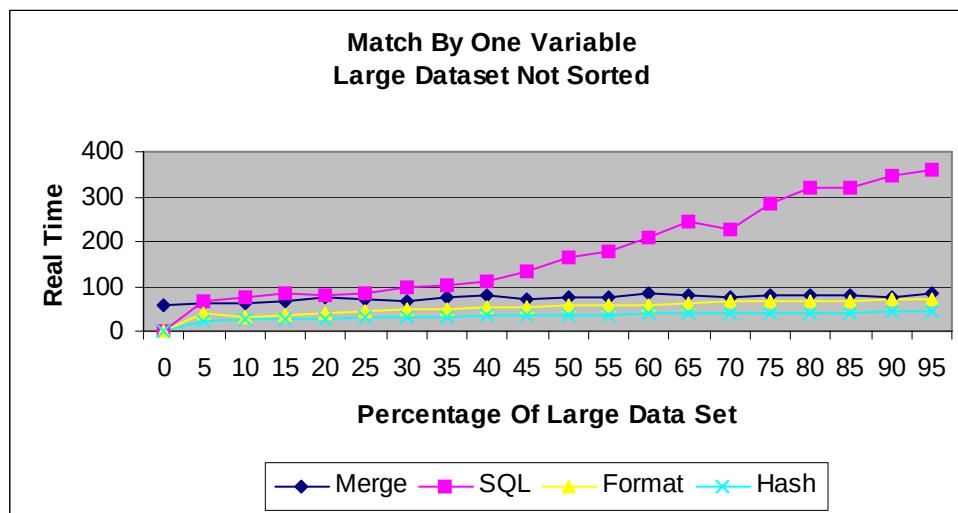
The large dataset used for all of these benchmarks contains 5,010,996 observations and 86 variables.

Our first set of tests explored performance when the large dataset was not sorted. Table 1.0 focuses on the elapsed or clock time to run single variable joins on a large nonsorted dataset using small datasets from 5% to 95% the size of the large dataset. [See below for explanation of the last row on Key Indexing.] To keep the table size manageable, we display here only a few of the size increments. The graph in Figure 1.0 is built from the complete set of measurements.

Table 1.0					
Real Time					
Small Data Set Sizes	5%	25%	50%	75%	95%
# Records	249,802	1,252,635	2,505,927	3,760,439	4,760,735
Merge	64	72	74.36	82.02	86.01
SQL	64.92	83.15	163.53	282.46	361.32
Format	39.17	43.66	57.2	67.28	70.64
Hash	22.96	30.40	35.31	40.12	44.3
Key Indexing	132.37	654.85	1347.02		

These results shows that when the small dataset was anywhere from 5% to 95% of the large dataset, three of the techniques (sort-merge, formats and hash tables) showed very small increases in elapsed time for significant increases in the size of the small dataset. SQL, on the other hand, slowed dramatically as the small dataset grew. This illustrates that SQL chooses different methods depending on what it knows about the datasets to be joined.

Fig. 1.0

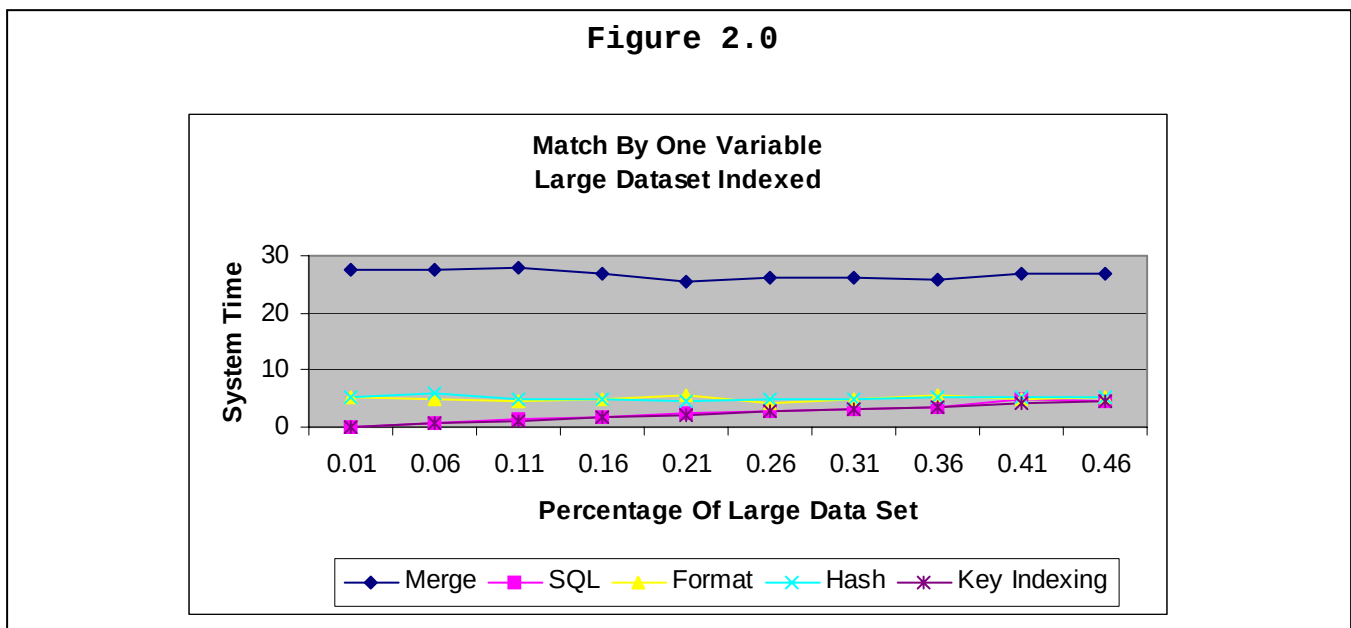


A note about the row in Table 1.0 referring to the performance of Key Indexing: Although the rest of Table 1.0 refers to the situation where the large dataset is neither sorted nor indexed, an index is required for applying the Key Indexing technique. The cost of developing the index is not included in the results shown in this Table. You can see that this technique takes much longer than any other, even at sizes as small as 5% of the large dataset. But we had read quite a bit in other publications about the gains available through indexed access. So we decided to try much smaller volumes to find where the technique is applicable.

Table 2.0 displays the CPU time in seconds for each of the techniques applied to the problem of an extremely small dataset being joined to a large, indexed dataset. We chose to focus on CPU rather than elapsed time in this analysis just to show that both are possible using this process. The CPU times are summed across all the steps necessary to accomplish the technique. For example, in the sort-merge technique, the time shown in the table includes the time to accomplish the sorts plus the time to complete the data step merge. These same results are shown in graph form in Figure 2.0

Table 2.0										
System Time										
Small Data Set	0.01%	0.06%	0.11%	0.16%	0.21%	0.26%	0.31%	0.36%	0.41%	0.46%
# Records	485	2,874	5,521	7,914	10,579	13,066	15,560	18,060	20,513	22,975
Key Indexing	0.09	0.56	1.10	1.67	2.15	2.95	3.21	3.45	4.28	4.54
SQL	0.12	0.70	1.27	1.66	2.54	2.82	3.14	3.43	4.95	4.70
Hash	5.29	5.78	4.81	4.73	4.60	4.87	4.84	5.28	5.26	5.10
Format	5.34	4.74	4.45	4.86	5.65	4.34	4.75	5.56	4.76	5.38
Merge	27.49	27.41	27.79	26.81	25.58	26.23	26.11	25.83	26.90	26.91

These results highlight the difference in performance of various techniques under different scenarios. While SQL and Key indexing might not have been the techniques of choice in the first test, they win the performance test hands down at the very small end of the spectrum. The sort-merge took six times the as long as the best of the other techniques. Using formats or hash tables gave about the same performance, and that didn't change much for the different sizes of samples tried. SQL and key indexed access, however, perform noticeably better than any other technique at the small end, but at the large end of this range they approach the times for using formats or hashing. The graph in Figure 2.0 makes these assessments clear.



The cutoff point for the use of indexed access is a function of both the percentage of the large dataset you need to find and the raw number of records. Some sources have suggested that indexed access is the preferred technique for up to 20% of the large dataset. But in our situation, it became prohibitively expensive at the low end of our first range (5% of the large dataset). Even at that "small" size, we were looking up over 200,000 records. Other techniques performed much better at that level than key indexing.

You also have to remember to consider the time to create and maintain the index on the large dataset. It would not be worthwhile to create an index to do one lookup, no matter how small. But if you frequently need to find small numbers of records, you can justify the overhead of the index.

Although not by any means exhaustive, these figures are included to make several points:

- 1) Indexing a large dataset can make repeated small lookups fly. Proc SQL or **set** with **key=** will take advantage of the index.
- 2) There are noticeable differences in the performance of the techniques. You can work more effectively by understanding and exploiting all these techniques.

3) With the **%logparse** macro (publicly available, see References section of this paper for location) and the macro included in this paper, it is easy to do benchmarking on your own system to produce performance guidelines tailored to your configuration of hardware and operating system.

OTHER GUIDELINES

In addition to performance, there are other considerations to keep in mind in choosing the right technique for a given situation:

- 1) What kind of join do you need? Only sort-and-merge and SQL can produce a full outer join. If you have a many-to-many situation, SQL is the only approach that will give you a Cartesian product or a “fuzzy join.” Fuzzy joins are those where the match between datasets is not precisely defined by matching variable values. For example, you may want to join all the claims that occurred between admit and discharge date. If you need a one-sided join, keeping all the records in the large dataset, indexed lookup will not work. If you need one-sided join, keeping all the records in the small dataset, formats and hash tables are not the best choices.
- 2) Are the datasets already sorted/indexed on the join keys? If so, use the benchmarks without the cost of the sort/index included to determine relative performance.
- 3) How often will this program be run? Intricate code such as hash tables pays back dividends over multiple uses; for a single use you may buy down programming time with run time by choosing a technique that is easier to code even if it takes little longer to run. SQL, because it has the flexibility to decide at run time which approach it will use to achieve the desired result set, can be unpredictable. If you need tight control or expect that appropriate information (such as current sort sequence) will not be available to SQL to help the optimizer make the right decision, you might want to choose another technique for a production job. For situations that call for multiple small lookups against a very large data source, indexed lookup is the hands down winner.
- 4) How many variables are needed from the data sources? Formats work very well for one variable lookups. More than that can be handled, but it takes some manipulation.
- 5) How many variables make up the match key? Benchmarks on one- and two- variable lookups show different relative performance.
- 6) How large are the datasets? Formats and hash tables have to fit into available memory. When joining two huge datasets, disk-based techniques (sort-and-merge, SQL and indexed lookup) might be more successful.
- 7) What system constraints do you have? If you are low on memory, your system can’t handle large formats or hash tables.
- 8) Who will support/maintain the code once it is written? If you do not have good SAS programmers to support your program once it leaves your hands, you may want to use SQL (since it is more commonly understood) or sort-and-merge (since it is more straightforward).

CONCLUSIONS

When faced with the task of joining data from different sources, a SAS programmer has to choose the approach that makes the most sense in a particular situation. The data, the hardware available, the skill level of the programmer, and the context of the requirement all play a part in determining the best decision. This paper contains information intended to expand the programmer’s skill set, to remove that limitation, and provide guidelines about which solution to choose in various combinations of other parameters.

Direct access lookup, using SET with KEY= on an indexed dataset, is the clear choice when there will be many small lookups from a large data source. Formats are excellent ways to accomplish a selection or one-field lookup. Hash objects offer huge savings when combining multiple fields from several datasets on different match keys when the datasets are not sorted by the join keys. Proc SQL is a great solution for novice SAS programmers who have experience with SQL in other databases. Judicious use of sort and merge provide the programmer the ability to execute a specific approach when needed, especially on extremely large, previously sorted datasets.

Since no one tool serves all purposes equally well, SAS programmers are well-advised to expand their toolkits. Gain familiarity with the techniques described here to increase both your job satisfaction and your value in the workplace.

REFERENCES:

Clifford, Billy. 2005. "Frequently Asked Questions about SAS Indexes." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available <http://www2.sas.com/proceedings/sugi30/008-30.pdf>

Dorfman, Paul and Vyverman, Koen. 2005. "Data Step Hash Objects as Programming Tools." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available <http://www2.sas.com/proceedings/sugi30/236-30.pdf>

Eason, Jenine. "Proc Format, a Speedy Alternative to Sort/Merge." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available <http://www2.sas.com/proceedings/sugi30/054-30.pdf>

Lavery, Russ, 2005. "The SQL Optimizer Project: _Method and _Tree in SAS 9.1." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available <http://www2.sas.com/proceedings/sugi30/101-30.pdf>

Lavery, Russ, 2004. "An Animated Guide: Speed Merges: Resource use by common non-parallel procedures." *Proceedings of the Seventeenth Annual NorthEast SAS Users Group Conference*. Available <http://www.nesug.org/html/Proceedings/nesug04/as/as01.pdf>

Levin, Lois, 2004. "PROC FORMAT – Not Just Another Pretty Face." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available <http://www2.sas.com/proceedings/sugi30/001-30.pdf>

Moorman, Denise J. and Warner, Deanna. "Indexing in the SAS® System, Version 6" Technical Report TS-580, SAS Institute, Inc. Available <http://support.sas.com/techsup/technote/ts580.pdf>

Raithel, Michael A. "Programmatically Measure SAS Application Performance on Any Computer Platform with the New LOGPARSE SAS Macro." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available <http://www2.sas.com/proceedings/sugi30/219-30.pdf>

Snell, Greg P. *post to SAS-L*. Subject line: Re: how to clear up log automatically. Tuesday, May 11, 2004.

Here is where to find and download the %**logparse** macro:

<http://support.sas.com/rnd/scalability/tools/fullstim/fullstim.html>

ACKNOWLEDGMENTS

SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina.

Intel® and Intel®Xeon™ are trademarks, service marks or registered trademarks of Intel Corporation.

Microsoft® and Windows® are registered trademarks of Microsoft Corporation in the United States and/or other countries.

The authors wish to express their thanks to Kimball Lewis for his ideas and leadership.

DISCLAIMER

All code provided in this paper is provided on an "AS IS" basis, without warranty. Neither the authors nor Health Dialog make any representation, or warranty, either express or implied, with respect to the programs, their quality, accuracy, or fitness for a specific purpose. Therefore, neither the authors nor Health Dialog shall have any liability to you or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by the programs provided in this paper. This includes, but is not limited to, interruption of service, loss of data, loss of profits, or consequential damages from the use of these programs.

CONTACT INFORMATION

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

Judy Loren or Sandeep Gaudana
Health Dialog
39 Forest Avenue
Portland, ME 04101
JLoren@HealthDialog.com or SGaudana@HealthDialog.com

APPENDIX A – MACRO TO GATHER BENCHMARKS

```

/*****
*
* The %Performance_Macro(), can be used to produce benchmarking results for any
* joining technique. The macro uses %LOGPARSE(), an experimental macro for SAS 9.1
* to parse the log file for performance statistics. The Macro can also plot a
* Graph for the measured performance statistics.
*
* Note that this macro was developed for the Windows environment. It can be
* adapted for other operating systems.
*
* Macro Parameters:
*
* @ Detail          : Description pertaining to a technique
*
* @ Dataset_to_sample : User supplied Source data used to generate different
*                        sizes of Small Datasets
*
* @ Program_Path     : Path of the included Program containing the technique
*                        whose Performance Characteristics will be measured
*
* @ Macro_Path       : %LOGPARSE() and %PASSINFO() should be stored in this
*                        location
*
* @ Output_Path      : "Performance_Data" Dataset used to capture the
*                        Performance Statistics will be created in this location
*
* @ Log_Path         : Logs generated from this macro will be stored at this
*                        location
*
* @ Low_limit        : Minimum Size of Small_Dataset (Default Value = 0)
*
* @ High_Limit       : Maximum Size of Small_Dataset (Default Value = 95)
*
* @ Range            : Range of Sizes for Small_Dataset between Low_Limit and
*                        High_limit
*
*                        Note : The following options require SAS/Graph
*
* @ Plot_Graph (0 or 1) : Value of 1 produces SAS Graph at the end of
*                        Performance Run
*
* @ Replace(Y or N)   : Based on Description for "Detail" Parameter
*                        "Replace = Y" replaces the Performance Statistics in
*****/
```

```

*                                     "Performance_Data"                                     *
/*****/;

%macro performance_macro(detail=,
                        dataset_to_sample=,
                        program_path=, macro_path=,
                        output_path=,
                        log_path=,
                        low_limit=0,
                        high_limit=95,
                        range=5,
                        plot_graph= 0,
                        replace=Y);

/*Options fullstimer is required for Logparse macro to capture performace*/
options mprint symbolgen fullstimer;

/*Performance_data is the dataset that will be used to store information
   for performance for any technique*/

%if %sysfunc(fileexist("&output_path.\performance_data.sas7bdat")) = 1 %then %do;
    data "&output_path.\performance_data.sas7bdat";
        set "&output_path.\performance_data.sas7bdat";
        if detail = "&detail." and "&replace." = 'Y' then delete;
    run;
%end;

/*****
*       The following DO Loop generates small dataset of different sizes       *
*****/;
%if %index(&low_limit,.) %then %do;
    %let decnum_low=%eval(%length(&low_limit)-%index(&low_limit,.));
%end;
%else %do;
    %let decnum_low=0;
%end;
%if %index(&high_limit,.) %then %do;
    %let decnum_high=%eval(%length(&high_limit)-%index(&high_limit,.));
%end;
%else %do;
    %let decnum_high=0;
%end;
%if %index(&range,.) %then %do;
    %let decnum_range=%eval(%length(&range.)-%index(&range,.));
%end;
%else %do;
    %let decnum_range = 0;
%end;

%let decnum = %sysfunc(max(&decnum_low,&decnum_high,&decnum_range));
%let ints=%sysevalf(&low_limit.*10**&decnum.);
%let inte=%sysevalf(&high_limit.*10**&decnum.);
%let inti=%sysevalf(&range.*10**&decnum.);

%do i = &ints %to &inte %by &inti;

```

```

data one;
  set &dataset_to_sample.;
  if ranuni(0) le ((&i*0.01)/(10**&decnum.));
  random=ranuni(0);
run;

/* The resulting small_dataset should be the standard name to be used in the code
containing the technique*/
proc sort data=one out=small_dataset;
  by random;
run;

/*The records are counted and are recorded in Performance_data dataset*/
proc sql noprint;
  select count(*)
    into :count
    from small_dataset;
quit;

/*****
/*The following proc printto saves the log file and it is followed */
/*by the inclusion of a program whose Performance Statistic is      */
/*required to be measured                                           */
/*NOTE: Logparse Macro only captures performance from saved log file*/
*****/
      proc printto log = "&log_path.\&detail._&i..log" new;
        run;

        %include "&macro_path.\passinfo.sas";
        %passinfo;

        %include "&program_path.";

        proc printto;
          run;

/*-----*/

/*The following datastep captures the errors if present in the above saved log
file*/
data errors(keep=logline);
  infile "&log_path.\&detail._&i..log" missover;
  length logline $256 code $20;
  retain code ;
  input;
  if index(_infile_, '0D'x) then logline=scan(_infile_,1,'0D'x);
  else logline=_infile_;
  logline = translate(logline, ' ', '%');
  if index(logline, ':') then code=scan(logline,1,':');
  else if substr(logline,1,5) ne ' ' then code=scan(logline,1,' ');
  if index(code, 'ERROR') =1 and logline ne ' ' then output errors;
run;

/*Prints the errors in Output Window*/
proc print data = errors;
  title "Errors in Log File : &detail.&i..log";
run;

/*Including Logparse Macro to capture performance from saved Log File*/
%include "&macro_path.\logparse.sas";
%logparse(&log_path.\&detail._&i..log,time);

```

```

/*The following steps are to append the information for performance into
performance_data Dataset*/
proc sql noprint;
    create table two as
        select "&detail." as Detail, &count. as records, ((&i.)/(10**&decnum.)) as
            small_data_size, sum(realtime) as real_time, sum(systime) as
            sys_time, sum(memused) as memory
        from time;
quit;

%if %sysfunc(fileexist("&output_path.\performance_data.sas7bdat")) = 0 %then
%do;
    data "&output_path.\performance_data.sas7bdat" ;
        length detail $ 50 ;
        records = .;
        small_data_size = .;
        real_time = .;
        sys_time = .;
        memory = .;
        if _n_ = 1 then delete;
    run;
%end;

proc append base = "&output_path.\performance_data.sas7bdat" data = two force;
run;
%end;

/*****
* The following steps are to plot a graph after the logparse macro measures
* performance
* Note: SAS/Graph is required
*****/;

%if &plot_graph. ne 0 %then %do;

    data graph_plot;
        set "&output_path.\performance_data.sas7bdat";
        where detail = "&detail.";
    run;

    symbol1 i=j v=dot c=blue w=3;

    /*Plots Graph for Real Time v/s Size of Dataset*/
    proc gplot data = graph_plot;
        plot real_time * small_data_size / autovref;
        label sys_time = 'Real Time' small_data_size = 'Percentage of Big Dataset';
        title "Real Time Plot : &Detail";
    run;
quit;

    /*Plots Graph for System Time v/s Size of Dataset*/
    proc gplot data = graph_plot;
        plot sys_time * small_data_size / autovref;
        label sys_time = 'System Time' small_data_size = 'Percentage of Big Dataset';
        title "System Time Plot : &Detail";
    run;
quit;

    /*Plots Graph for Memory v/s Size of Dataset*/

```

```

proc gplot data = graph_plot;
  plot memory * small_data_size / /*haxis = axis1*/ vaxis=axis2 autovref;
  label memory = 'Memory Used' small_data_size = 'Percentage of Big Dataset';
  title h = 1 "Memory Usage Plot : &Detail.";
run;
quit;

%end;

%mend performance_macro;

```

APPENDIX B – SAMPLE USAGE OF MACRO

User sets up the following code and stores it as

\\srv6\A\nesug_2005\programs\SQL_Index_Match_One_Var.sas

```

/*Proc SQL technique*/
proc sql _tree _method;
  create table sql_out as
    select a.*
      from wh.members_indexed a
        join
          small_dataset b
        on a.contno = b.contno;
quit;

```

Macro call:

```

%performance_macro(
  detail = SQL_Index_Match_One_Var,
  dataset_to_sample = wh.members_indexed,
  program_path = \\srv6\A\nesug_2005\programs\SQL_Index_Match_One_Var.sas,
  macro_path = \\srv6\A\nesug\_2005\macros,
  output_path = \\srv6\A\nesug_2005\outputs\new_output1,
  log_path = \\srv6\A\nesug_2005\logs,
  low_limit = 0.01,
  high_limit = 0.07,
  range = 0.05,
  plot_graph = 1,
  replace=Y);

```