Paper 1717-2014

Collaborative Problem Solving in the SAS® Community

Tom Kari, Tom Kari Consulting

ABSTRACT

When a SAS® user asked for help scanning words in textual data and then matching them to pre-scored keywords, it struck a chord with SAS programmers! They contributed code that solved the problem using hash structures, SQL, informats, arrays, and PRX routines. Of course, the next question was which program is fastest? This paper compares the different approaches and evaluates the performance of the programs on varying amounts of data. The code for each program is provided to show how SAS has a variety of tools available to solve common problems. While this won't make you an expert on any of these programming techniques, you'll see each of them in action on a common problem.

INTRODUCTION: A HISTORY OF SAS EXPERTS HELPING THEIR PEERS

One of the best things about being a SAS user is that you're not alone. In addition to the renowned SAS Technical Support organization, several venues exist to help with questions and problems, among them:

- SAS-L, the e-mail list that discusses SAS;
- The SAS Support Communities, at communities.sas.com;
- www.sascommunity.org, the collaborative online community for SAS users worldwide.

Last summer, a user whose handle is *sucksatsas* posted the following:

> *Hello- I am trying to count the number of negative and positive words within each "Comment" observation.*
>
> *Variables: Comment: A list of a website comments*
>
> *Negative: A list of negative associated words, each observation is one negative word*
>
> *Positive: A list of positive associated words, each observation is one positive word*
>
> *I have gathered the information and combined them into one data set, now I am confused how to scan/index each comment observation when I'm looking to see if it contains any word in an entire list and eventually count them to something like this:*
>
> *Does anyone have any insight?*

Well, did people have insight! Over the next few days, a number of responses displayed the flexibility and power of SAS.

During the discussion, the topic of performance came up, and by the time that the different approaches had been assessed, it seemed worthwhile to present the results to a wider audience.

And *sucksatsas* clearly doesn't; he knows possibly the most important thing about SAS, that there's a community of users ready and willing to help.

ACKNOWLEDGMENTS

Ordinarily this section comes near the end, but in this case the project was such a collaborative effort that the participants need to be acknowledge at the start. And so, I present your *dramatis personæ*! (In order of appearance)

sucksatsas: a humble SAS user, looking to get a job done;

Hai.kuo: Esteemed *master* of the SAS communities, overseer of data of the meta variety;

Linlin: Another *master*, plying her trade at an institution of healing;

PGStats: From the frozen North, this *master* applied mathematician has few peers;

Ksharp: An exotic figure from the far Orient, a *master* skilled in the arts of the DATA step;

Vince28@Statcan: A mysterious *apprentice*, about whom little is known;

*data_null_*: A great *master*, whose rare pronouncements shake the Earth like thunder;

Fugue: SAS *apprentice*, again cloaked in mystery;

and your lowly scribe, an *apprentice* to those with higher wisdom.

And I convey my sincere thanks to these colleagues, who teach me new things about SAS every day.

SAMPLE DATA

To initially validate the different programs, I used the following steps based on the original post. You can use this code as well if you want to try any of the programs.

```
data GoodList;
    input GoodWord$;
    cards;
wow
great
ok
good
better
best
run;

data BadList;
    input BadWord$;
    cards;
bad
meh
boring
never
run;

data Have;
    input Comment $50.;
    cards;
wow so great
it's ok
good but boring
meh
good good good better best never let it rest
run;
```

APPROACH 1: HASH TABLES

Hai.kuo was first of the mark, and as he frequently does he demonstrated his mastery of the SAS hash object. This is a relatively new addition to SAS. As stated in the ***SAS 9.3 Language Reference: Concepts, Second Edition***:

> *The hash object provides an efficient, convenient mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on lookup keys.*

and Hai.kuo's code looks like this[1]:

[1] In fact, these programs also contained functionality to deal with upper/lower case, word delimiters other than space, etc. To keep this paper focused on the different matching techniques used, the problem has been simplified to one of all lower-case words, separated by spaces.

In addition, each contributor's code has been changed so that the different samples provided in this paper will have similar structure and terminology; responsibility for making them worse is mine alone.

```
data Want;
        if _n_ = 1 then                                                          ①
                do;
                        if 0 then                                                ②
                                set GoodList BadList;
                        declare hash G(dataset:'GoodList', multidata:'y');       ③
                        G.definekey('GoodWord');                                 ④
                        G.definedone();                                          ⑤
                        declare hash B(dataset:'BadList', multidata:'y');
                        B.definekey('BadWord');
                        B.definedone();
                end;

        set Have;                                                                ⑥
        length TestString $ 50;

        do _i = 1 by 1 until (missing(TestString));
                TestString = scan(Comment,_i,' ');

                do rc_good = g.find(key:TestString) by 0 while (rc_good = 0);    ⑦
                        GS + 1;                                                  ⑧
                        rc_good = g.find_next(key:TestString);
                end;

                do rc_bad = b.find(key:TestString) by 0 while (rc_bad = 0);
                        BS + 1;
                        rc_bad = b.find_next(key:TestString);
                end;
        end;

        Rating = sum(GS,-BS);
        call missing(GS, BS);
        keep Comment Rating;
run;
```

There isn't room in this paper to cover all of the functionality of the program, but I will describe what the statements relating to the hash object do. (As we have a set of "good" and "bad" files and variables, I'll use x and xxx to represent either.)

1. The code within the "if _n_ = 1" block runs once at the start of the program, and initializes the hash objects. This only needs to be done once, and the objects are retained for the program duration.
2. The "if 0 then set ..." is one of the odder constructs used in SAS. If you try running the code without it, you'll receive an error to the effect that xxxWord is an undeclared key symbol. By including it, the definitions of the variables in xxxList are accessible in the data step, even though no data is actually read by the SET statement. While a detailed discussion is beyond the scope of this paper, one can be found at http://www.lexjansen.com/nesug/nesug88/sas_supervisor.pdf.
3. The DECLARE statements tells the compiler that the object references G and B are of type hash. The dataset:'xxxList' argument in the DECLARE hash statement causes SAS to load the work.xxxlist dataset into the hash object. The multidata argument causes SAS to allow duplicate keys in the hash object.
4. The x.definekey('xxxWord'); statement specifies that the hash object will be searched using the values of the xxxWord variable. Note that there is no DEFINEDATA statement; we are only interested in finding whether or not the key exists in the hash object, not in retrieving any data associated with the key.
5. And finally, the two definedone statements cause SAS to complete the initialization of the hash objects.

So at this point we have the two hash objects loaded into memory from the datasets, one with all of the good words as keys, the other with the bad words.

6. The program then reads through the Have dataset. For every record, the "do _i = 1 by 1" loop will extract the words from the comment, and then the two inner loops will find first all of the good word matches, and then all of the bad word matches.
7. The rc_xxx = x.find_next(key:TestString) assignment positions the hash to the next match using TestString as the key. Ordinarily, the data associated with the key would then be retrieved, but in our case we simply want to know if the key exists, which is indicated with a zero return code.
8. Counts are incremented, and when all of the words in Comment have been processed, the difference between the good and bad scores is calculated and the comment and associated rating are output to SAS dataset Want.

APPROACH 2: SQL

Personally, I thought Hai.kuo hit it out of the park with his response, but processes that require matching and compiling data records are amenable to a SQL approach, so I coded one and submitted it to the thread. (On the unlikely chance that somebody reading this paper isn't familiar with SQL, it is the most common, possibly the only remaining syntax for communicating with databases. Now go and get familiar with it!)

My code follows:

```
data Have_Words(keep=Comment CandidateWord);                                    ①
      set Have;
      length CandidateWord $ 50;

      do _i = 1 by 1 until (missing(CandidateWord));
            CandidateWord=scan(Comment, _i, ' ');

            if ~missing(CandidateWord) then
                  output;
      end;
run;

proc sql;                                                                       ②
      create table Want_Details as                                              ③
            select Comment, 1 as Rating                                         ④
                  from Have_Words h inner join GoodList g                       ⑤⑥
on(h.CandidateWord = g.GoodWord)
                        outer union corresponding                               ⑧
                              select Comment, -1 as Rating                      ⑦
                                    from Have_Words h inner join BadList
b on(h.CandidateWord = b.BadWord)
      ;
quit;

proc means noprint nonobs nway;                                                 ⑨
      class Comment;
      var Rating;
      output out=Want(drop=_type_ _freq_) sum=;
run;
```

1. The first program reads in the comments, and creates a dataset containing one record per word, with the original comment as well. This is because SQL is set-oriented, and since the things we're looking for are words, we have to use a target file that contains words as well.[2]

[2] Not completely true. There are actually SQL operators that will let you scan text for words, but since the SAS community exists to help people who are learning SAS, this would have made the post very complicated.

The next program uses SQL to compare the words in the word lists to the words associated with the comments, and assign scores.

2. "proc sql" … "quit;" wraps the SQL code. If you're using SAS interactively, you can type SQL, get the results, type some more SQL … This is why PROC SQL requires a "quit;", not a "run;".
3. "create table Want_Details as" causes SAS to create a SAS dataset from the results of the SQL program. If this wasn't there, the results would be sent to the print destination. SQL can be used for both reporting and data manipulation.

The remainder of the SQL code consists of two select statements, joined by an outer union statement. The first select statement does the following:

4. "select" creates a result set, which will consist of the variables following it, namely Comment and a variable named Rating which will always contain the literal 1.
5. The "from" clause causes SAS to get the data from two datasets, Have_Words and GoodList, which are referred to in the rest of the SQL code with abbreviations h and g. Default processing in SQL is to create an output dataset containing every possible combination of the records in the first dataset and the second dataset. Known as a "Cartesian join", there are very few cases where this is what we want. Therefore we limit it…
6. …with an "inner join … on" clause, which specifies that out of all of these possible combinations of output records, we only want the ones that have matching values of CandidateWord from file h and GoodWord from file g.
7. The second select statement does the same thing, except returning the records that matched BadWord with a Rating of -1.
8. And finally, the "outer union" clause concatenates the two sets of results, from the comparisons with GoodList and BadList. The result is a dataset with two variables, Comment and Rating, where Rating is always 1 or -1.
9. The proc means step then summarizes the results by Comment, producing the total of the Rating values for each Comment.

APPROACH 3 INFORMAT:

Next through the door was PGStats, with an approach based on informats. Informats are a SAS facility that are usually used to transform character strings into numbers; for example they can be nicely used to convert "YES" and "NO" to 1 and 0. Leave it to the creative mind of my fellow Canadian to use them in a transformation like this!

```
data Emotion;
       length type $1 label $4 fmtname $7;
       set GoodList(in=pos rename=(GoodWord=start)) BadList(in=neg                    ①
rename=(BadWord=start));
       fmtname = "Emotion";                                                           ②
       type = "I";
       label = put(pos - neg, best4.);                                                ③
run;

proc format cntlin=Emotion;                                                           ④
run;

data Want(drop=pos len _i);
       set have;
       Rating = 0;
       call scan(Comment, 1, pos, len, " ");                                          ⑤

       do _i = 2 by 1 while (pos>0);                                                  ⑥
              Rating = sum(Rating, input(substr(Comment, pos, len), ??                ⑦
Emotion.));
              call scan(Comment, _i, pos, len, " ");
       end;
run;
```

Most of the code in this program is involved with creating a user-defined format. For those of you who have only used PROC FORMAT to create formats and informats with lists of values and labels, it can be a real time saver to use the procedure with pre-defined datasets containing the information that the procedure needs.

1. In the set statement, SAS first reads GoodList, and then BadList. The "in=" options set the variables to 1 when the dataset contributes a record, otherwise 0. The xxxWord is renamed to "start", because proc format requires that the value have this name when using an input dataset.
2. The informat name will be "Emotion", and it will be a numeric informat.
3. This ingenious little trick results in good words having a label of 1, and bad having -1.

The resulting dataset looks like this…

type	label	fmtname	start
I	1	Emotion	wow
		…	
I	1	Emotion	best
I	-1	Emotion	bad
		…	
I	-1	Emotion	never

4. …and in the next step it is the input to proc format, via the "cntlin" option. This results in PROC FORMAT creating an informat the equivalent of specifying:

```
proc format;
    invalue Emotion
           'wow' =    1
           ...
           'best' =    1
           'bad' =   -1
           ...
           'never' =   -1
    ;
run;
```

This is a technique that you definitely want to keep in your back pocket…it works for formats also!

So we've run all this code, and created an informat; big deal.

Now the fun starts. In the next step, the comment records are read in the usual way.

5. The "call scan" statement returns the start and length of the nth word in the comment.
6. The do loop cycles through all of the words in the comment.
7. PGStats didn't make my job easier with this terse statement!

   First, "Rating = sum(Rating, [other stuff])" will add 1, -1, or nothing to Rating depending on what happens in [other stuff].

   In "input(substr(Comment, pos, len), ?? Emotion.)", the substr function pulls the word out of the comment, as found by "call scan".

   And "input(xxx, ?? Emotion.)" will use the Emotion informat to transform whatever character value is in xxx into the numeric value specified by the informat (in this case, 1 for "good" words, and -1 for "bad" ones).

   Just as in the input statement, the ?? modifier will prevent a failure to find something from causing an error. So if the word isn't found, a missing value will be returned, which will be disregarded by the sum function.

APPROACH 4: ARRAY

Not to be outdone, Hai.kuo revisited with yet another approach, this one using an array:

```
proc sql noprint;                                                              ①
      select nobs into :bad_obs from dictionary.tables where LIBNAME='WORK'
AND MEMNAME='BADLIST';
      select nobs into :good_obs from dictionary.tables where LIBNAME='WORK'
AND MEMNAME='GOODLIST';
quit;

data Want;
      if _n_=1 then                                                            ②
            do;
                  do _i=1 to bobs;
                        set BadList nobs=bobs point=_i;
                        array B(&bad_obs) $10. _temporary_;                    ③
                        B(_i)=BadWord;
                  end;

                  do _i=1 to gobs;
                        set GoodList nobs=gobs point=_i;
                        array G(&good_obs) $10. _temporary_;
                        G(_i)=GoodWord;
                  end;
            end;

      set Have;
      length TestString $ 50;

      do _i = 1 by 1 until(missing(TestString));                               ④
            TestString=scan(Comment, _i, ' ');
            GS+ (TestString in G);
            BS+ (TestString in B);
      end;

      Rating=sum(GS,-BS);
      call missing(GS, BS);
      keep Comment Rating;
run;
```

1. Another good technique to keep in your back pocket. Hai.kuo is using proc sql and the SAS dictionary tables to find the number of observations in the good and bad word tables, and to write these numbers to macro variables for later use.
2. Very similar to the hash table example, except in this case it's a little more work because the tables actually have to be read into the arrays, instead of being loaded with one statement.
3. Because the _temporary_ option is used, there's no need to name the array elements.
4. In this case, there's less work needed, because the entire array can be examined without a loop.

VARIATIONS ON A THEME: TWO APPROACHES WITH PRX ROUTINES, AND A DIFFERENT USE OF SCAN

Now you've seen the four major approaches discussed in this paper. But "that's not all", as they say in the commercials.

PRX 1: USING PRX TO PARSE THE WORDS, INSTEAD OF THE SCAN FUNCTION

Up till now, everyone has used the scan function to parse the comments into individual words. Another way to do this is with the SAS prx (Perl regular expression) functions, which implement regular expressions in SAS.

PGStats contributed one example of using prx functions. In the "if _n_ = 1" section of the code, add these lines:

```
retain prxId;
```

```
prxId = prxparse("/\w+/");
```

The prxparse string instructs the facility to look for one or more “word” characters (alphanumeric and _).

and then in the code, instead of using scan, first set Start to 1:

```
Start = 1;
```

and then loop through the comment with the following statement:

```
call prxnext(prxId, Start, -1, Comment, pos, len);
```

SAS will automatically advance “Start” to the position after a string is identified, and the word can be retrieved simply using:

```
TestString = substr(comment, pos, len);
```

PRX 2: USING PRX TO FIND THE GOOD AND BAD WORDS

Vince28@Statcan had an even wilder idea.

He used the following code

```
proc sql noprint;
     select GoodWord
     into :good seperated by '|'
     from GoodList;

     select BadWord
     into :bad seperated by '|'
     from BadList;
quit;
```

to create macro variables &good and &bad, which contained all of the good and bad words separated with | symbols, like this:

```
wow|great|ok
```

and then his prxparse statement looks like this:

```
prxId = prxparse("/\b(?:&good.)\b/i");
```

which instructs the facility to look for a word boundary, followed by any of the words (because they’re separated by the | (or) symbol), followed by a word boundary.

And then his search looks like this:

```
do until (pos=0);
call prxnext(prxId, Start, -1, Comment, pos, len);
if pos > 0 then GS=GS + 1;
end;
```

using the prxnext function to walk through the comment.

TWO ADDITIONAL SCANNING TOOLS

Ksharp pointed out that the countw function can be used to control the loop:

```
do _i = 1 to countw(Comment, ,'ka');
…
end;
```

where the modifiers to countw are “a”, which instructs sas to use alphabetic characters as word dividers, which makes little sense until you find out that “k” reverses it, using all characters that AREN’T alphabetic characters.

And similarly, he uses

```
TestString = scan(Comment, i, ,'ka');
```

to extract the words from the comments.

PERFORMANCE TESTING

So, how do our different approaches compare?

To test these programs under somewhat realistic circumstances, I pulled a bunch of Toronto restaurant reviews off the web. The “Comment” file contained 26,434 records and 12,561,666 words, consuming 25 MB. For lists of good and bad words, I extracted some appropriate words from the comments, and ended up with 133 bad and 187 good words.

These tests were run on my PC, which has the following characteristics:

- Intel Core i7, 2.40 GHz
- 16 GB RAM
- Windows 8 64-bit

On this machine these tests ran too quickly to distinguish between the contenders, so I upped the “Comments” file to 264,340 records and 25,616,660 words, consuming 255 MB. And since these tests didn’t take too long, I went up yet another order of magnitude, to 2,643,400 records and 256,166,600 words, consuming 2.54 GB.

With the SQL approach, the large test never finished, and locked up my computer. It had to be cancelled.

The results are shown below:

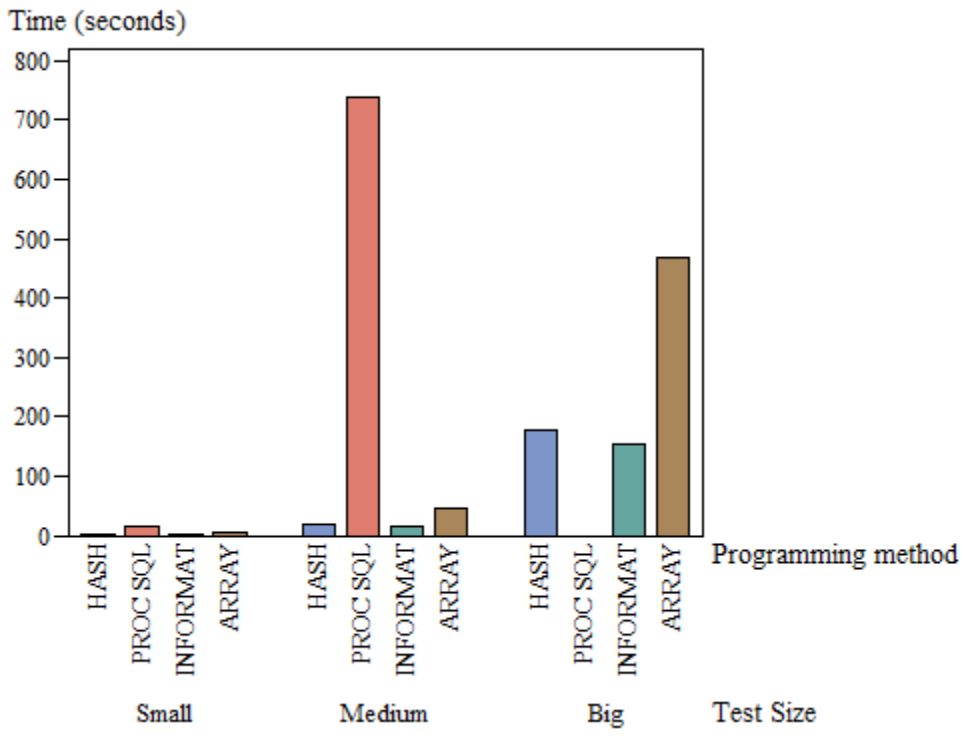

Figure 1: Comparing SAS Programming Methods: Elapsed Time

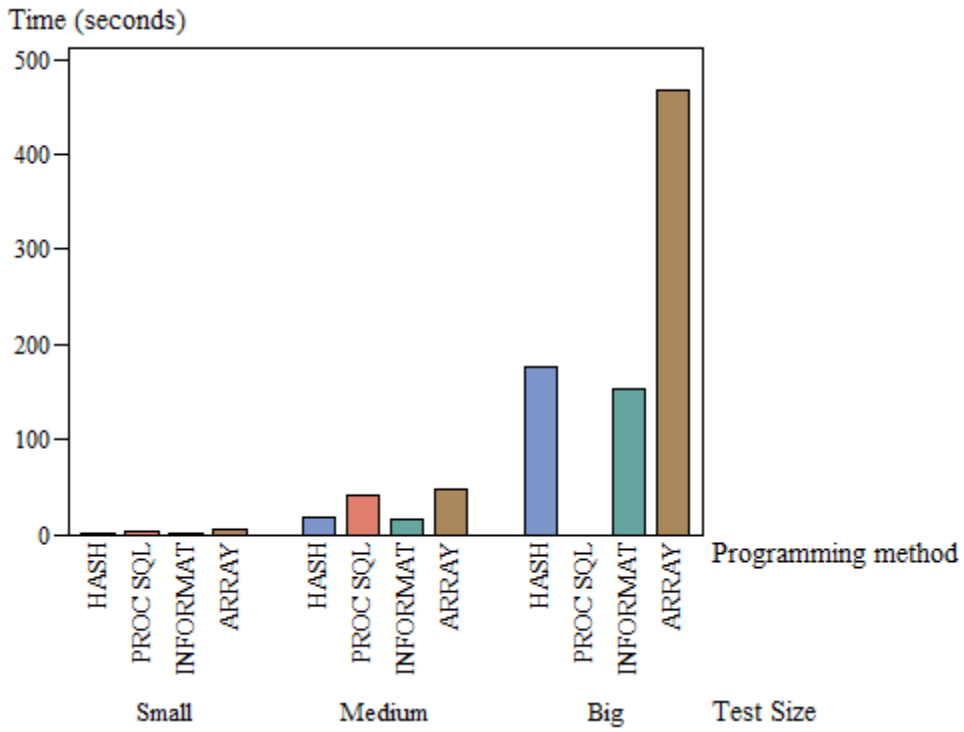

Figure 2: Comparing SAS Programming Methods: User CPU Time

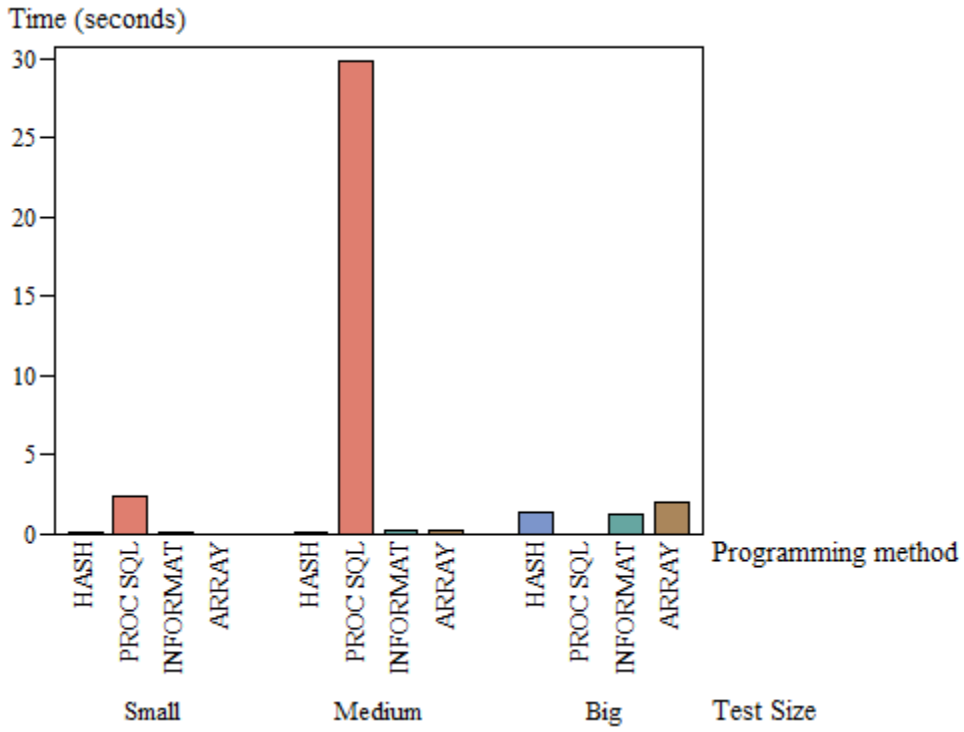

Figure 3: Comparing SAS Programming Methods: System CPU Time

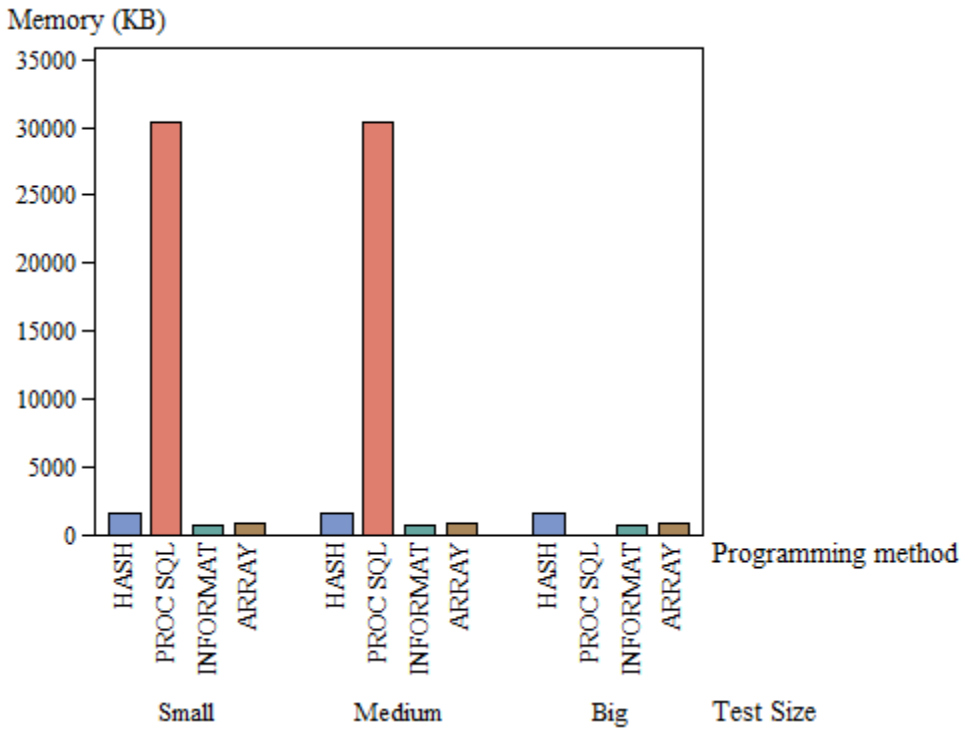

Figure 4: Comparing SAS Programming Methods: Memory usage

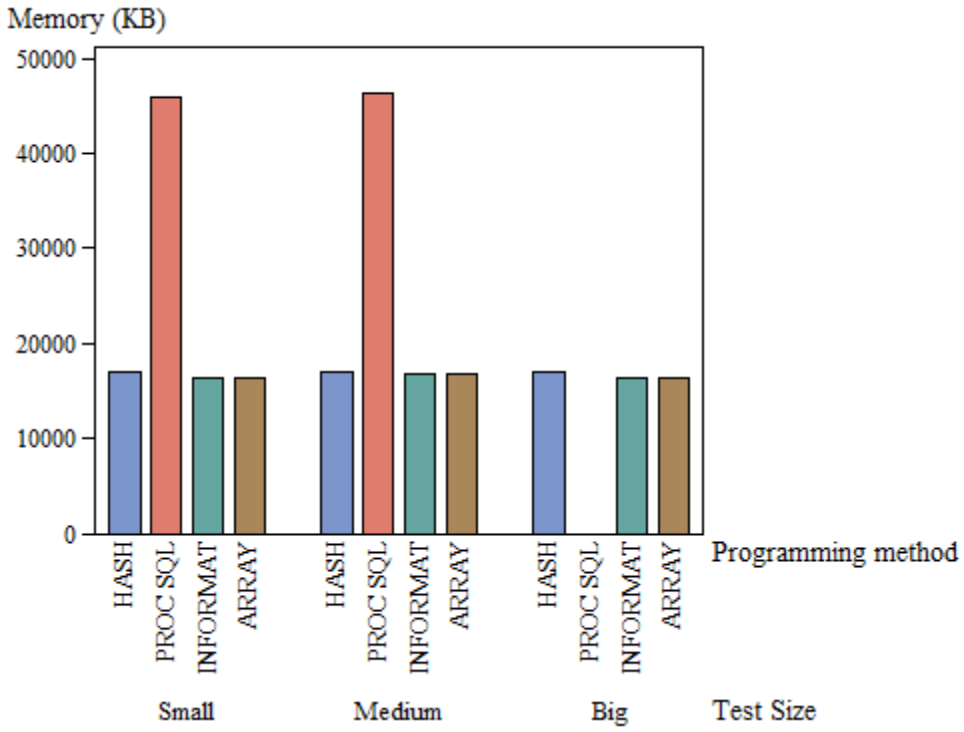

Figure 5: Comparing SAS Programming Methods: OS Memory Usage

IMPACT OF USING MORE KEYWORDS

The other factor that could have a bearing on the speed is the number of keywords to be processed. To see what effect this had on system resources and time, I reran the large volume tests, but with 7,210 bad keywords and 10,136 good keywords.

In these tests, the Array software never finished and had to be cancelled. Rerunning it with 10% of the phrases, it ran in around 27 minutes, so it could be expected to take in the area of five hours to complete.

The following chart shows how elapsed and CPU time vary between the small and large keyword tests:

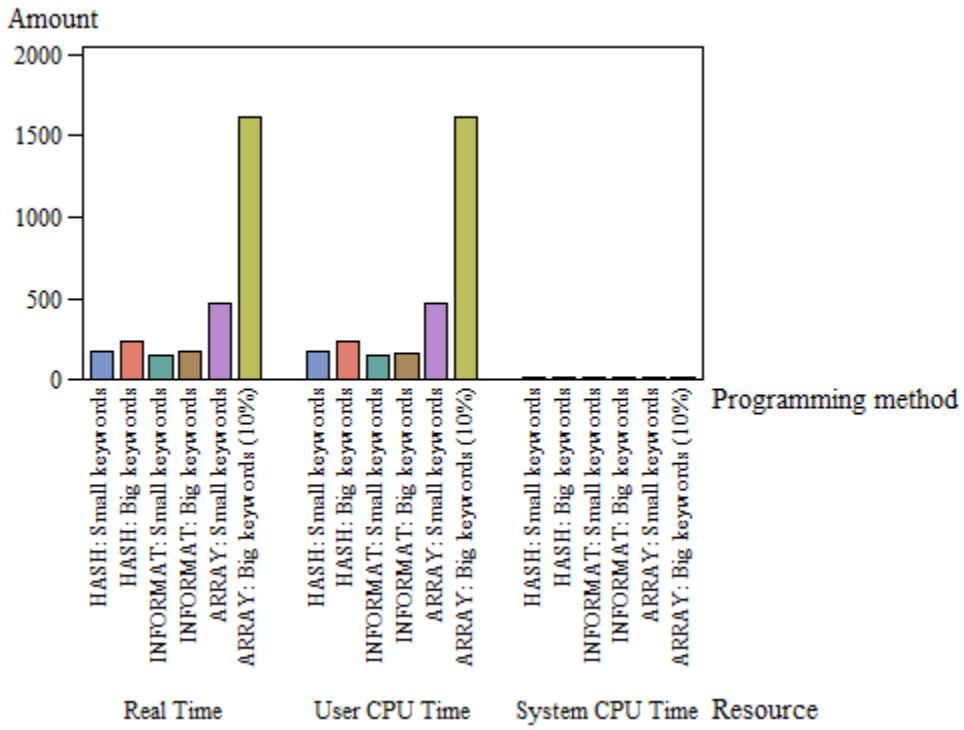

Figure 6: Impact on time of large keyword count

And this one shows the same for memory:

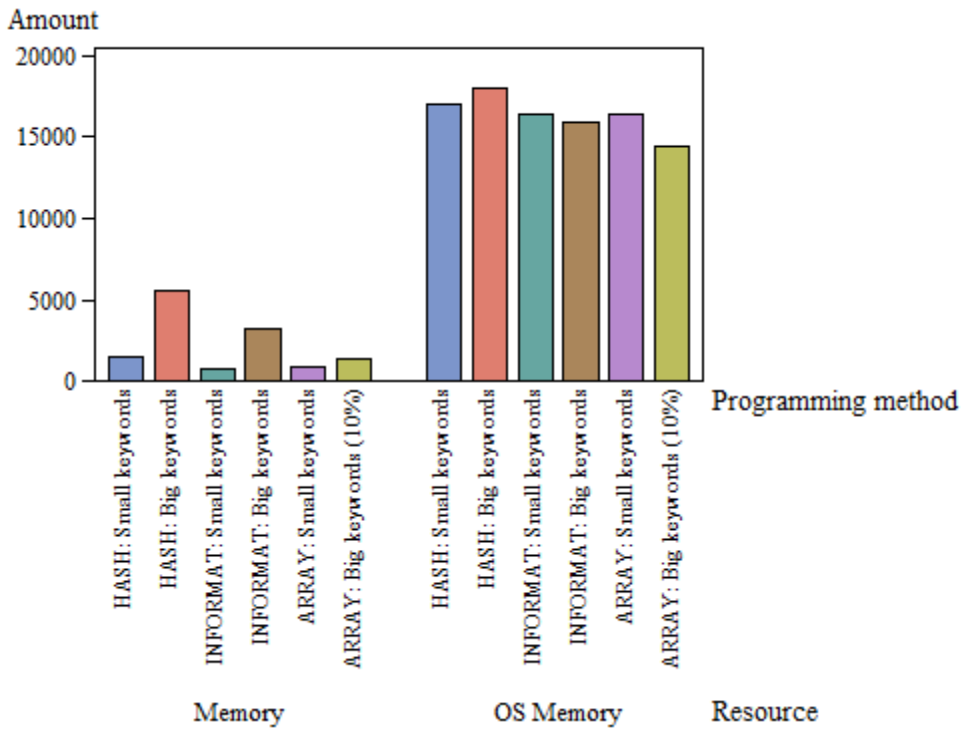

Figure 7: Impact on memory of large keyword count

VARIATIONS ON A THEME

And finally the largest tests were rerun using the prx scanning techniques and the countw loop control. As expected, the performance statistics were very similar to the hash and informat tests, as long as the 'o' parameter is added to the countw and scan functions, to allow them to only parse the input once.

CONCLUSION

SAS is clearly rich in functionality to perform this kind of processing. If volume and time weren't constraints, any of these alternatives would be fully adequate to the requirement. However, with these results I feel that some options were better than others.

1. The SQL approach is obviously not appropriate for this problem. The need to expand the Comment file to one record for every word in the file, and to then match against this expanded file, imposes a performance penalty that makes this approach unrealistic.

   This is very much in line with my previous experience with SAS for data manipulation, where either SQL or a DATA step is a naturally efficient way to process the data. In this case, it's clearly the DATA step that is the way to go.

2. I also wasn't expecting the approach of concatenating all of the keywords into a macro variable, and feeding it into a prx lookup, to be very realistic. This turned out to be the case, with slow performance and problems once the list becomes large. But in another context, it might perfectly reasonable.

3. I was surprised that the hash approach was so much faster than the array option. My original thought was that since they are both memory-based lookups, performance would be similar, but this wasn't the case, particularly with a large number of keywords.

   I'm wondering if the keys in the hash object are organized with a btree or some other kind of index that makes the search so quick.

4. I would have to recommend against using arrays for this operation in the future, as the hash and informat approaches are so superior.

5. There appears to be very little to recommend the hash or informat approach against the other. In my case, it would probably come down to the fact that I haven't used hash objects previously, so if I was in a rush I would lean towards the informat.

   On the other hand, if I truly had a key that was the index into additional data, as the hash object is designed for, it would be an absolute no-brainer to use it. The code in this simple example is enough to give me confidence that I would be able to use it if needed.

6. The various character functions like countw, scan, and prx… all appear to deliver similar performance. If I had a simple requirement like this I would tend to use countw and scan, but it's nice to know that prx is there if I need to do something more challenging.

I hope you have found this exploration of SAS coding approaches interesting and informative. Remember, there is a wealth of knowledge available on the SAS online support sites.

CONTACT INFORMATION

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

Name: Tom Kari
Organization: Tom Kari Consulting
Address: 302-317 Metcalfe St
City, State ZIP: Ottawa, Ontario, K2P 1S3
Work Phone: 613-899-6359
Email: tom.kari.consulting@bell.net