Paper CT08

Scaling up for the SQL payoff, the most efficient way to code

Matt Metherell, PHASTAR, London, UK

ABSTRACT

Many programmers have a love-hate relationship with SQL, partly through lack of familiarity, and partly because many of the savings it offers are in the writing, rather than the performance, of code. There are of course notable exceptions to this such as dictionary.columns running significantly faster than sashelp.vcolumn, but there are also cases when proc SQL outperforms proc means, and even the humble merge statement. In this paper I offer some examples and look at why SQL is more efficient in these scenarios.

INTRODUCTION

We will look at situations where SQL runs more efficiently than data steps and other SAS procedures, particularly on larger datasets where the time saving can be quite significant. A lot of what is covered here may only be useful for datasets > 100,000 observations, although this will depend on individual servers' performance, the number and type of by variables, and the width of the dataset, to name but a few factors.

METHODOLOGY

In all of the cases described, tests were performed ten times with the average duration reported and graphed. Each instance involved all methods being tested on the same randomly-generated dataset.

PRELIMINARIES

One may assume *proc means* would be more efficient than *proc sql*, and while that is true for simple cases, there are many instances when *proc sql* gives the upper hand. First we shall look at some baseline comparisons of the abstract cases, as building blocks for the practical uses we investigate later in this paper. We shall look in depth at *proc means*, but broadly similar results can be replicated with *proc summary* and *proc univariate*.

AVERAGING

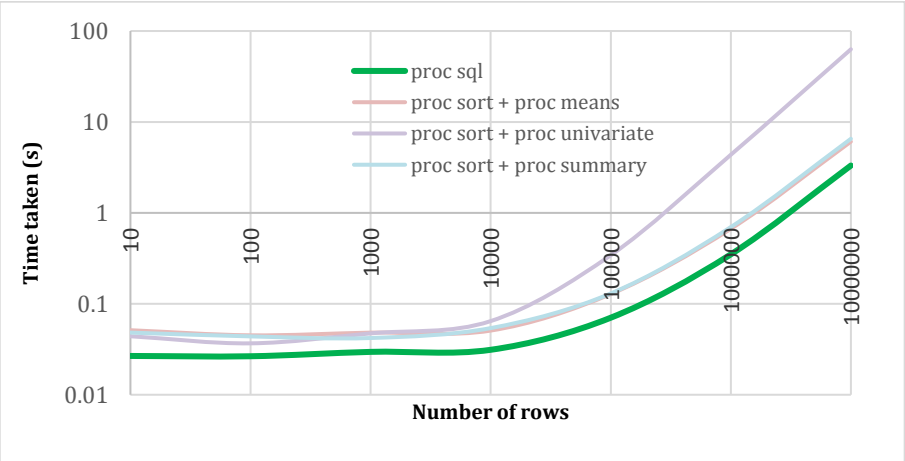
Proc sql is no match for *proc means* when finding the mean of a string of values in a sorted dataset, but when working with unsorted datasets, a single *proc sql* outperforms *proc means* on large datasets when a *proc sort* is used in addition.

```
proc sql;
   create table dsout_sql as
   select studyid, mean(age) as age
   from dsin_unsorted
   group by studyid;
quit;
```

vs.

```
proc sort data=dsin_unsorted out=dsin_ms;
   by studyid;
run;

proc means data=dsin_ms noprint;
   by studyid;
   var age;
   output out=dsout_means_s mean=mean;
run;
```

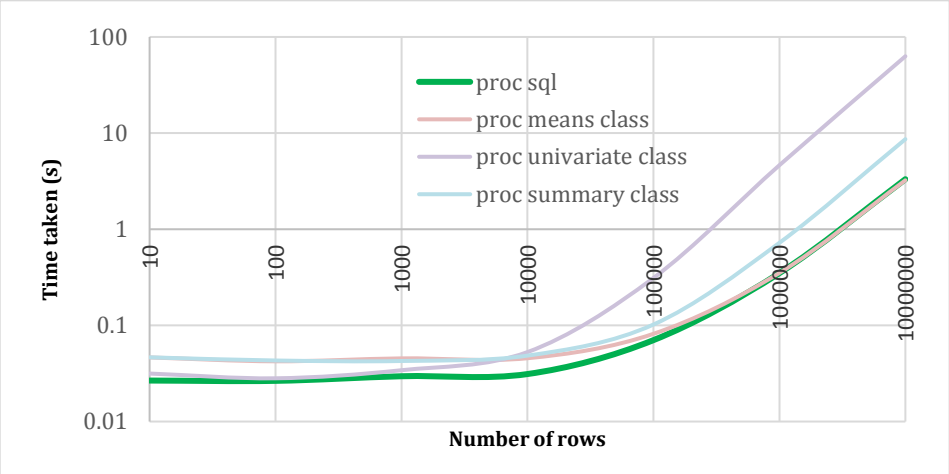


Alternatively, when the class statement is used, things are a lot closer.

```
proc sql;
   create table dsout_sql as
   select studyid, mean(age) as age
   from dsin_unsorted
   group by studyid;
quit;
```

vs.

```
proc means data=dsin_unsorted noprint;
   class studyid;
   var age;
   output out=dsout_means_c mean=mean;
run;
```



While *proc sql* is not faster than *proc means* in a straight shoot-out, we shall see that it does outperform it when other factors come into play. This also demonstrates what an advantage it is when sorting can be performed without the need for a separate procedure.

SUMMARY STATISTICS

As of SAS 9.4, *proc sql* can calculate the median as an aggregate function, meaning it uses values down a column, rather than across a row. When adding *min*, *max*, *median*, *stddev*, and *n* into the mix, *proc means* shows its advantage, as it is faster than *proc sql* for finding the collection of all of these statistics in one go, even when the data is unsorted.

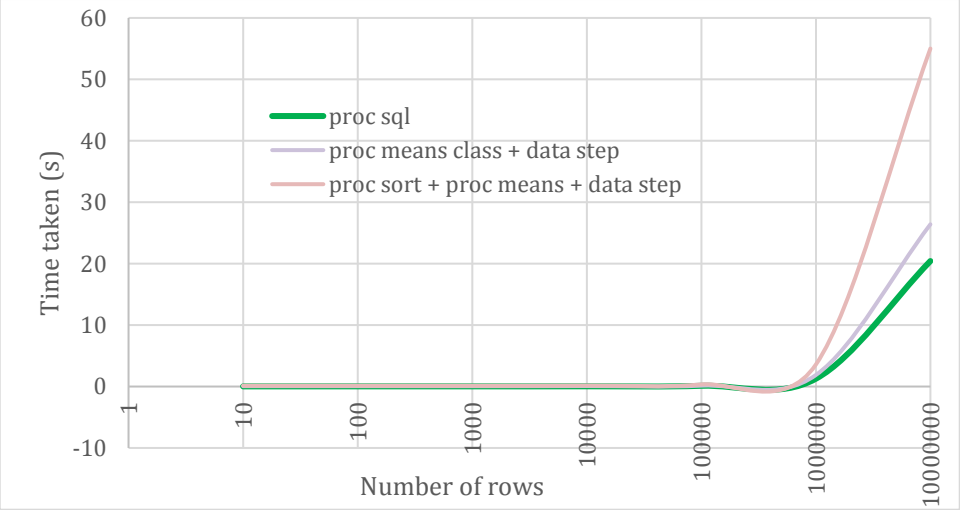
However, when formatting is needed, *proc sql* does not need an extra data step, and therefore becomes the quicker option. In the following code, we round values, set the number of decimal places and combine variables.

```
proc sql;
   create table dsout as
   select distinct studyid,
          put(count(age),8.) as n,
          put(round(mean(age),0.01),5.2)||" ("||put(round(std(age),0.001),5.3)||")" as mean_std,
          put(round(median(age),0.01),5.2) as median,
          put(round(min(age),0.1),4.1)||" to "||put(round(max(age),0.1),4.1) as min_max
   from dsin_unsorted
   group by studyid;
quit;
```

vs.

```
proc means data=dsin_unsorted noprint;
   class studyid;
   var age;
   output out=dsout_means_c n=n1; mean=mean1; std=std1; median=median1; min=min1 max=max1;
run;

data dsout_means_c_2 (keep = studyid n mean median min_max);
   set dsout_means_c;
   length n mean median min_max $20;
   n       = put(n1,8.);
   mean    = put(round(mean1, 0.01), 5.2)||" ("||put(round(std1,0.001),5.3)||")";
   median  = put(round(median1, 0.01), 5.2);
   min_max = put(round(min1, 0.1), 4.1)||" to "||put(round(max1,0.1),4.1);
run;
```



In the next few sections, we can see some practical applications of these differences, and will only consider the class case of *proc means*.

MAXIMUMS AND MINIMUMS

Even on sorted data, *proc sql* outperforms *proc means* when deriving a maximum or minimum row, because we want to keep variables associated with the maximum row, which is not something *proc means* is designed to do.

When creating something like the following table, we don't just want to find the maximum AVAL, we also want to find the ADY on which that maximum occurred (coloured in the table below):

USUBJID	AVISIT	ADY	AVAL
ABC001	Week 1	7	20
ABC001	Week 2	14	35
ABC001	Week 3	21	25
ABC001	Week 4	28	30
ABC001	***Maximum***	***14***	***35***
ABC002	Week 1	7	30
ABC002	Week 2	14	35
ABC002	Week 3	21	40
ABC002	Week 4	28	45
ABC002	***Maximum***	***28***	***45***

Proc sql can keep variables not included in its *group by* or *having* statements, for example the date on which the maximum value occurred, by simply adding them to the *select* statement. *Proc means* cannot, and will need to be merged on to the original data to populate these.

```
proc sql;
   create table dsin_sql as
   select usubjid, ady, aval,
          "Maximum" as avisit length=50
   from dsin_sorted
   group by usubjid
   having aval=max(aval)
   order by usubjid, avisit, ady;
quit;

proc sort data=dsin_sql out=dsout_sql nodupkey;
   by subjid avisit;
run;
```

vs.

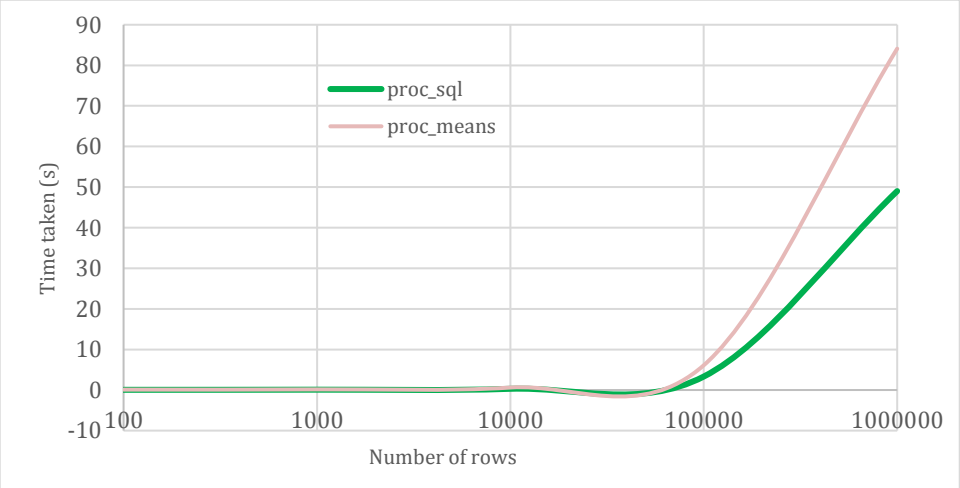
```
proc means data=dsin_sorted noprint;
   class usubjid;
   var aval;
   output out=dsin_means max=max;
run;

data dsin_means_2;
   merge dsin_sorted
         dsin_means (in=b rename=(max=aval));
   by usubjid aval;
   if b;
   avisit = "Maximum";
run;

proc sort data=dsin_means_2 out=dsin_means_2s;
   by subjid avisit ady;
run;

proc sort data=dsin_means_2s out=dsout_means nodupkey;
   by subjid avisit;
run;
```

In the case of ties, both methods will need to de-duplicate the final data, and there's not a simple way to add a *nodupkey* to SQL sorting. In the above example, the earliest occurrence of the maximum value is kept.



ADDING AN AVERAGE ROW

Sometimes we'll want to find an average only if there are multiple observations on the same day or visit.

USUBJID	ADY	AVAL	DTYPE
ABC001	1	20	
ABC001	2	35	
ABC001	3	25	
ABC001	3	30	
ABC001	***3***	***27.5***	***AVERAGE***
ABC001	4	30	
ABC001	5	35	
ABC001	6	40	
ABC001	6	45	
ABC001	***6***	***42.5***	***AVERAGE***

Here *proc sql* beats *proc means*, even on sorted data, because of when the data is filtered down to that which needs to be averaged.

In order to keep only results with multiple observations, *proc means* must find all results and then filter the final dataset, unless we use another procedure to find when we need an average. *Proc sql* can filter earlier, so only produces results when there are multiple observations.

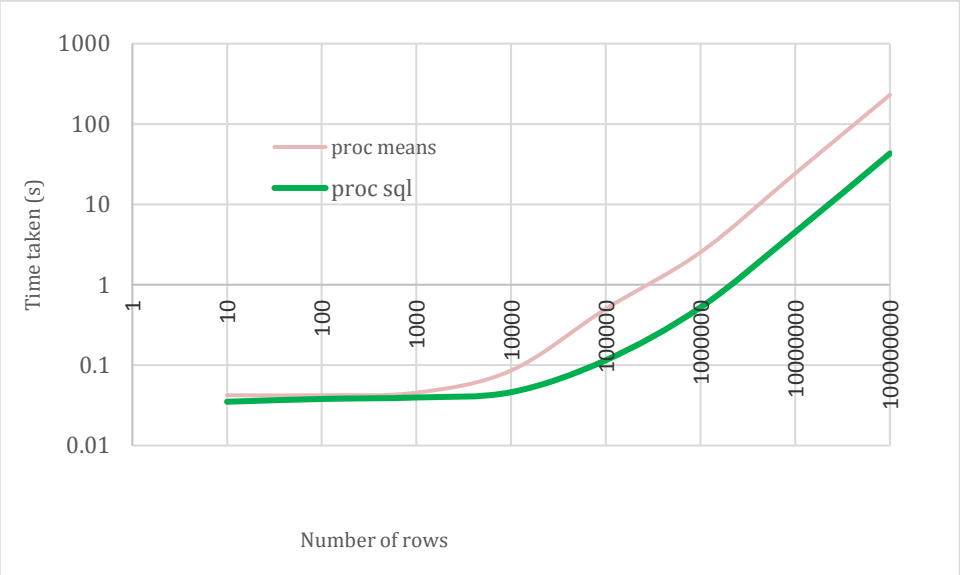
In the following example, whenever any combination of USUBJID and ADY appears more than once, we create an observation including the mean as AVAL, thanks to the *group by* and *having* statements highlighted below. Plus SQL has the added advantage of adding new variables and values, such as DTYPE="AVERAGE".

```
proc sql;
   create table dsout_sql as
   select usubjid, ady,
          mean(aval) as aval,
          count(aval) as count,
          "AVERAGE" as dtype length=8
   from dsin_unsorted
   group by usubjid, ady
   having count>1;
quit;
```

vs.

```
proc means data=dsin_unsorted noprint;
   class usubjid ady;
   var aval;
   output out=dsout_means_c (where=(n>1)) mean=mean n=n;
run;
```

As the differences are considerable (~45s vs ~230s for 100 million observations), the results are displayed on a log scale:



CONSOLIDATION OF FLAGS

If we want to find out if a subject ever has a variable flagged as Y, there's a really simple way with SQL, using the *max* function in a way that highlights a difference between *proc sql* and *proc means*. Suppose we have the following table:

USUBJID	ADY	VAR1FL	VAR2FL
ABC001	1		Y
ABC001	2	Y	
ABC001	3	Y	
ABC001	4		
ABC002	1		
ABC002	2		Y
ABC002	3		
ABC002	4		
ABC003	1	Y	
ABC003	2		
ABC003	3		
ABC003	4		

And we want to summarise the flags as:

USUBJID	VAR1FL	VAR2FL
ABC001	Y	Y
ABC002		Y
ABC003	Y	

With *proc sql* we can find the maximum value of VARxFL per USUBJID, but *proc means* can only find the maximum of a numeric variable. For *proc sql* the maximum value is the latest alphabetically, so we need to know the possible values, such as null < N < Y.

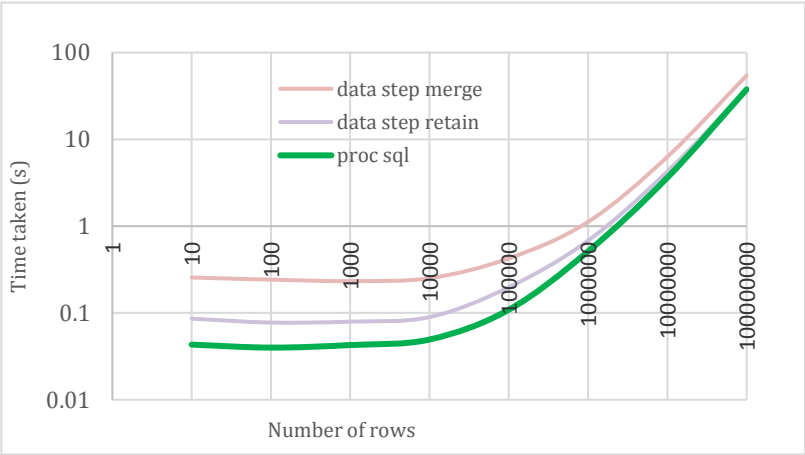
```
proc sql;
   create table dsout_sql as
   select distinct usubjid,
        max(var1fl) as var1fl,
        max(var2fl) as var2fl
   from dsin_unsorted
   group by usubjid;
quit;
```

vs.

```
proc sort data=dsin_unsorted out=dsin_retain;
   by usubjid;
run;

data dsout_retain;
   set dsin_retain;
   by usubjid;
   retain temp1 temp2 " ";
   if first.usubjid then call missing(temp1, temp2);
   if var1fl="Y" then temp1="Y";
   if var2fl="Y" then temp2="Y";
   if last.usubjid then do;
      var1fl = strip(temp1);
      var2fl = strip(temp2);
      output;
   end;
run;
```

Using a data step, we would need to merge together a dataset for each flag sorted with a *nodupkey*, or use a retained variable that only keeps the last row per subject. When the data is unsorted, both of these options take longer that the SQL code above, although when performed on a dataset of 100 million observations the difference became less than a second, so the timings of SQL and the retain method appear to converge for large datasets.



MERGING

For some people, SQL merges are reserved only for many-to-many merges which cannot be done in a *data* step, but we shall see that it can also be quicker for inner joins.

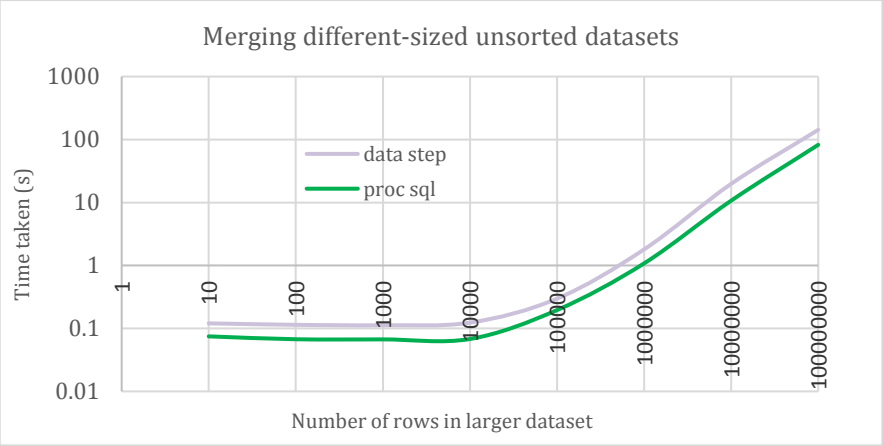
SQL will use a different method for merging depending on the datasets involved, and the memory constraints of the system being used. *Proc sql* inner joins are quicker than a *data* step merge up to a point. If both datasets are the same size, a *data* step becomes more efficient once the datasets are too big to be stored in memory.

However, when the smaller of the two datasets fits in memory, *proc sql* uses the hash method, which is faster than a standard dataset merge. If we use the *_method* option, it will print to our log the method that was used, with *sqxjhsh* denoting the hash merge.

```
proc sql _method;
   create table dsout as
   select a.*, b.var2
   from ds1 as a, ds2 as b
   where a.ind=b.ind;
quit;
```

```
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxjhsh
              sqxsrc( WORK.DS1(alias = A) )
              sqxsrc( WORK.DS2(alias = B) )
```

So if the datasets are different sizes, only the smaller of the two needs to fit in memory for *proc sql* to be more efficient, no matter how big the other may get. In the below graph, 100 million observations are the difference between ~80s and ~140s.



CONCLUSION

I hope this paper has clearly demonstrated some of the ways in which *proc sql* can speed up the runtime of one's code, and while this is far for a complete list, I hope it gives a sense of intuition for other cases where it can be beneficial.

REFERENCES

How data step and proc sql works, Deepanshu Bhalla
https://www.listendata.com/2016/04/how-data-step-and-proc-sql-works.html
Efficiency Techniques for Beginning PROC SQL Users, Kirk Paul Lafler
https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/127-29.pdf
Improving Query Performance
https://documentation.sas.com/?docsetId=sqlproc&docsetTarget=n1wue1kaft1rlsn1v0y5m569besp.htm&docsetVersion=9.4&locale=en
A Hash Alternative to the PROC SQL Left Join, Kenneth W. Borowiak, Howard M. Proskin et al
https://www.lexjansen.com/nesug/nesug06/dm/da07.pdf

RECOMMENDED READING

DATA Step vs. PROC SQL: What's a neophyte to do? Craig Dickstein, Ray Pass
https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/269-29.pdf
SAS Coding: PROC SQL Versus DATA Step, Jamie D'Agord
https://www.zencos.com/blog/sas-data-step-vs-proc-sql-superbowl/
Powerful and Hard-to-find PROC SQL Features, Kirk Paul Lafler
https://support.sas.com/resources/papers/proceedings14/1240-2014.pdf

CONTACT INFORMATION

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

Matt Metherell
PHASTAR
Unit 2, 2A Bollo Lane,
London W4 5LE
Email: matthew.metherell@phastar.com