Paper 3699-2019

Calculating Leads (and Lags) in SAS®: One Problem, Many Solutions

Andrew Gannon, The Financial Risk Group, Cary NC

ABSTRACT

This paper seeks to explain and demonstrate the many ways of calculating Leads, and to a lesser extent, Lags in SAS. Proc Expand, reverse sorting, data merge, Proc SQL, and application of data-set functions will be discussed. It will look at the practicality of each method and examples of the code structure of each. It will also analyze how much time and memory each method uses and present the findings to the readers.

INTRODUCTION

SAS has several built-in methods for calculating Lags. The most utilized is the **lag()** data-step function which can be specified with values. There is no corresponding lead function available to users at this time. The only known built-in lead calculation feature is via the PROC EXPAND procedure, which will be discussed in later in this paper. This paper will show examples that use **info** dataset - the creation of which can be found in Appendix A. Each section will contain an overview as well as the advantages and disadvantages of the given method. There are accompanying appendices that look at more in depth and the memory usage and timing of each method graphically.

REVERSE SORT METHOD

The first technique for calculating Leads is the reverse sort method. This method utilizes the lag functionality to generate leads. This is one of the simpler methods to utilize from a programming perspective. The design is to reverse sort the dataset on the key variables and then create lead variables using the lag() function. Then reverse sort again to get the original dataset, but now containing the lead variables. Note that to calculate lags in this method, the lag function can simply be used on the final data-step.

```
proc sort data = work.info1 out = work.reverse_info;
   by descending cnt;
run;

data work.leads_reverse;
   set work.reverse_info;
   value1_lead1 = lag(value1);
   value2_lead5 = lag5(value2);
run;

data work.leads;
   set work.info end = eof;
   if _n_ eq 1 then do;
      if 0 then set work.leads;
      declare hash lead(dataset:"work.leads_reverse");
      lead.definekey( "cnt");
      lead.definedata("value1_lead1 ","value2_lead5");
      lead.definedone();
   end;
   lead.find();
run;
```

ADVANTAGES AND DISADVANTAGES

This method is simple and scales nicely. The memory usage does not change very much when more leads or lags are added but does increase significantly with more observations. Timing goes up by ~10% for double the number of current leads being calculated. This method does use a lot of memory due to the

multiple steps, especially the sort and the lookup. This method also requires the most lines of code among the methods listed in this paper.

GROUPS WITH LEADS & LAGS

When dealing with groups with the reverse sort method, only the middle data step requires changes. These changes are to check if the current id matches the id associated with the lead observation. Memory usage goes up marginally when using groupings, but time does go up by roughly ~30% - though it's important to note that this is only for the middle step.

```
data work.leads_reverse;
   set work.reverse_info;
   value1_lead1 = lag(value1);
   value2_lead5 = lag5(value2);
   if lag1(id) ne id then value1_lead1 = .;
   if lag5(id) ne id then value2_lead5 = .;
run;
```

PROC EXPAND METHOD

This technique uses a built-in SAS procedure to easily calculate leads and lags on the fly. This procedure can be used for far more than just this - it can calculate moving averages, weighted moving averages, logit, and reciprocals just to name a few. Simply define the new lead variables and the procedure will handle the rest.

```
proc expand data = work.info1 out = work.leads method = none;
   id cnt;
   convert value1 = value1_lead1 / transformout = (lead 1);
   convert value2 = value2_lead5 / transformout = (lead 5);
   convert value1;
   convert value2;
run;
```

ADVANTAGES AND DISADVANTAGES

This method uses the only built-in lead abilities that SAS offers. It is easy to use and very valuable if other proc expand functionalities are being used. Memory grows at pace much slower and lower than the reverse-sort method and scales evenly - the same amount of memory is used regardless of how many transformations are being used. The biggest disadvantage to this method is the timing. Though it uses nearly half the memory of the reverse-sort method, it can take significantly more time. Each transformation adds a factor to the amount of time, for four transformations, it takes nearly twice as long as the reverse-sort method. The amount time also scales faster as data increases, rather than in a higher, more parallel fashion.

GROUPS WITH LEADS & LAGS

Groups are very easy using proc expand because it has a built-in by group option. The example below shows it in action. Using the by-group has two major implications with this method. First, it increases time significantly - upwards to three times as much for two leads with one by-group. Secondly, the memory usage can fluctuate wildly - going down and up depending on the groupings.

```
proc expand data = work.info2 out = work.leads method = none;
   id cnt;
   by id;
   convert value1 = value1_lead1 / transformout = (lead 1);
   convert value2 = value2_lead5 / transformout = (lead 5);
   convert value1;
   convert value2;
run;
```

DATA MERGE METHOD

This technique is another very simple method to calculate leads. It involves using the merge abilities within the data-step processing. This method merges the data with itself, as many times as necessary, to obtain leads utilizing the *firstobs*= functionality. For each future value that is needed, the user will merge a dataset with the *firstobs = 1 + lead_increment* in the merge. This will then begin adding the rows of the same dataset to itself but starting at the specified lead value. Note that this method cannot be used to calculate lags - it can only shift the observation back, not forward (the lag function would need to be used in this method).

```
data work.leads;
   merge work.info1
         work.info1 (firstobs = 2 rename=(value1 = value1_lead1) keep = value1)
         work.info1 (firstobs = 6 rename=(value2 = value2_lead5) keep = value2);
run;
```

ADVANTAGES AND DISADVANTAGES

This method is not as simple as the others due to its intricate use of the merge and its options. This method requires the user to increment by one more than the lead and to properly rename the variables. It is also important to keep only the value(s) that are being used to save time and memory. If user needed two leads at the same lead increment, the above code would need to be adjusted to rename for each variable and to keep both variables as well. The advantage to this method is the speed and memory usage. It uses significantly less memory than both the proc expand and the reverse-sort methods. It also scales very well, with no change to memory based on data-size or number of leads and lags. Timing goes up by ~25% for double the number of current leads being calculated.

GROUPS WITH LEADS & LAGS

Working with groups of variables when calculating using this method requires the creation of flags. These flags will be generated for each merged dataset (except for the first) and will be used in comparison operations to adjust for moving from one group to the next. To do so, the ID variable (group) will be read in and renamed from each dataset - then the original ID (from first dataset) will be compared with the merged dataset to determine if the same dataset is still being used. If they are not, we manually adjust the values to missing or zero. Timing goes up by roughly 40-50% and memory goes up marginally by less than 5%.

```
data work.leads;
  merge work.info2
    work.info2 (firstobs = 2 rename=(value1 = value1_lead1 id=id1) keep = id value1)
    work.info2 (firstobs = 6 rename=(value2 = value2_lead5 id=id5) keep = id value2);
  if id ne id1 then value1_lead1 = .;
  if id ne id5 then value2_lead5 = .;
run;
```

PROC SQL JOIN

The method is done using a simple sql left-join. It involves using a left join with specific join keys. The user will join the data with itself where the join is set to *key = lead - key* where the keys are the same between the two datasets.

```
proc sql;
   create table work.leads as
   select a.*, b.value1 as value1_lead1, c.value2 as value2_lead5
      from work.info1 as a
         left join
            work.info1 as b
               on a.cnt eq (b.cnt - 1)
         left join
            work.info1 as c
               on a.cnt eq (c.cnt - 5);
quit;
```

ADVANTAGES AND DISADVANTAGES

This method is only sql method that is discussed in this paper and can be very useful for SAS users who primarily use the sql features. This method uses less memory than the reverse-sort and expand methods, but more than the data-merge. The time is towards the high end of those discussed. Timing goes up by ~25% for double the number of current leads being calculated. This method can also be used for lags which is important because the lag() function is not available in sql steps.

GROUPS WITH LEADS & LAGS

Dealing with groups includes one additional step utilizing SQL. Simply add in a check with each join to verify that the current id matches the id of the joined variable. Time goes up by ~25% with two leads. Memory usage goes up, similarly, by roughly ~30%. Below is an example with two leads and a singular by-group.

```
proc sql;
   create table work.leads as
   select a.*, b.value1 as value1_lead1, c.value2 as value2_lead5
      from work.info2 as a
         left join
           work.info2 as b
         on a.cnt = (b.cnt - 1)
            and a.id = b.id
         left join
           work.info2 as c
         on a.cnt = (b.cnt - 5)
            and a.id = b.id;
quit;
```

DATA SET FETCHING

The last method discussed is somewhat different than the rest. It utilizes new data set functionality that allows the user to 'open' datasets without reading them into memory. This is similar to the Data Merge method, but rather than each dataset being read into memory, only the initial dataset is read and then that same dataset is opened (this only needs to be opened once regardless of the number of lags or leads being generated). The steps involve opening the dataset, pulling the location of the desired column, fetching the desired row, and writing out the data from that row. It uses the open(), varnum(), fetchobs(), and getvarn() functions.

```
data work.leads (keep = cnt value1 value2 value1_lead1 value2_lead5);
   set work.info1;
   retain loc_value1 loc_value2 dsid;
   if _n_ eq 1 then do;
      dsid = open('work.info1');
      loc_value1 = varnum(dsid,'value1');
      loc_value2 = varnum(dsid,'value2');
   end;
   rc1 = fetchobs(dsid,_n_ + 1);
   value1_lead1 = getvarn(dsid,loc_value1);
   rc2 = fetchobs(dsid,_n_ + 5);
   value2_lead5 = getvarn(dsid,loc_value2);
run;
```

ADVANTAGES AND DISADVANTAGES

This is one of the most efficient methods both in timing and by memory standards. This is because it only every reads the dataset into memory once, and the data step functions execute very quickly. Unlike the merge, memory goes up marginally for every new lag or lead (less than half a percent for each new lead). Similarly, the timing also goes up by less than 1% per variable. This is, by far, the most efficient method, though it is more complicated and requires three lines of code per new variable.

GROUPS WITH LEADS & LAGS

The upside to this method with groups is that it utilizes the same dataset for everything. The initialization of the dataset with the open() function can be used to pull in the id variable and then data-set if-then statements can be used to determine the groupings. Timing goes up by roughly ~80% and memory goes up by ~20% (little due to fixed amount regardless of size).

```
data work.leads (keep = cnt value1 value2 value1_lead1 value2_lead5);
   set work.info2;
   retain loc_value1 loc_value2 dsid loc_id;
   if _n_ eq 1 then do;
      dsid = open('work.info2');
      loc_value1 = varnum(dsid,'value1');
      loc_value2 = varnum(dsid,'value2');
      loc_id = varnum(dsid, 'id');
   end;
   rc1 = fetchobs(dsid,_n_ + 1);
   value1_lead1 = getvarn(dsid,loc_value1);
   value1_id = getvarn(dsid,loc_id);
   if value1_id ne id then value1_lead1 = .;
   rc2 = fetchobs(dsid,_n_ + 5);
   value2_lead5 = getvarn(dsid,loc_value2);
   value5_id = getvarn(dsid,loc_id);
   if value5_id ne id then value2_lead5 = .;
run;
```

CONCLUSION

There are many ways to generate lags and leads in SAS. There are surely more ways to do it than the five methods listed in this paper. Each method has its own advantages and disadvantages. The method that a user picks should be based on several factors including skill level, timing, and memory. There are accompanying appendices that look at the efficiencies of the five methods discussed in the paper.

REFERENCES

https://support.sas.com/documentation/onlinedoc/ets/132/expand.pdf

CONTACT INFORMATION

If you have any comments or concerns, please feel free to reach out at any time. Please contact at:

Andrew Gannon

The Financial Risk Group, Inc.

+1 (919) 439-3819

Andrew.Gannon@frgrisk.com

www.frgrisk.com

APPENDX A --- INFO DATASET

The **info** dataset that is used throughout this paper as the incoming data for all methods of calculating leads was created using the following code:

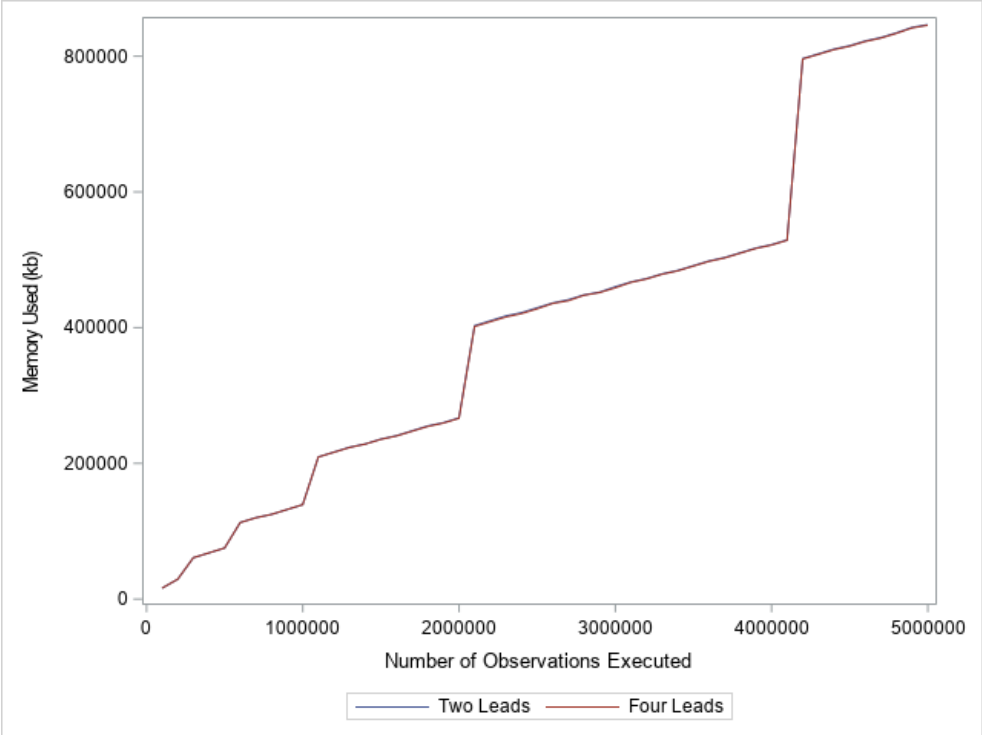
```
data work.info1;
   do cnt = 1 to 5000000;
      value1 = ranuni(1) * cnt;
      value2 = ranuni(2) * cnt;
      output;
   end;
run;
```

The **Info2** dataset that is used throughout the paper as incoming data for all methods with groupings was calculated using the following code:

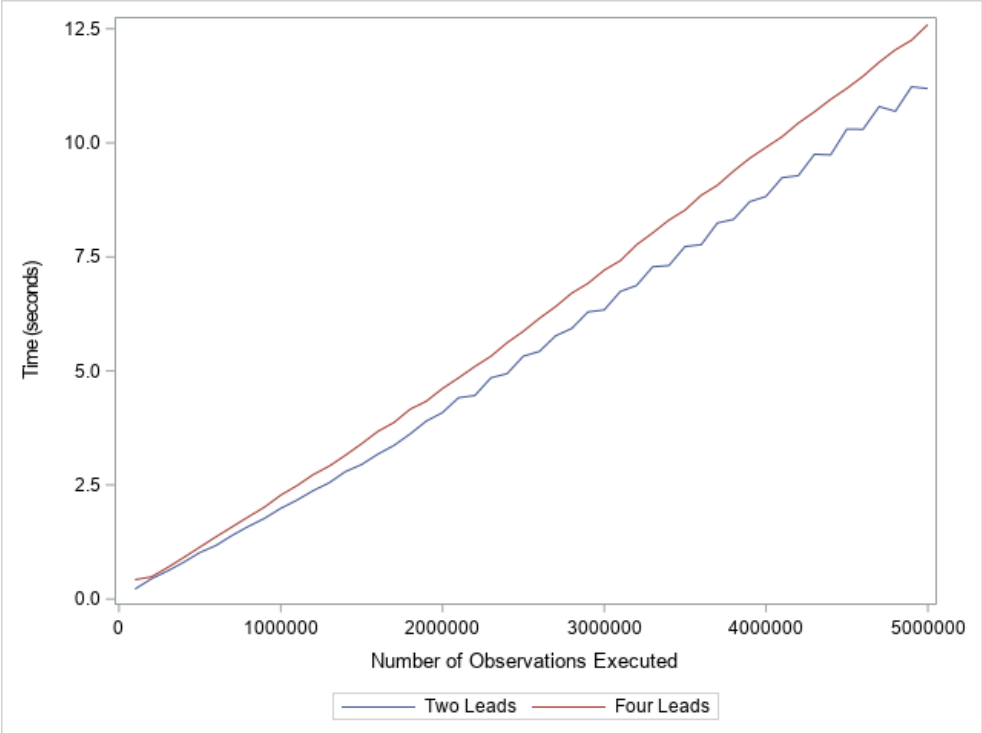
```
data work.info2;
   retain id 0;
   do cnt = 1 to 1000;
      if mod(cnt,20) = 0 then id = cnt;
      value1 = ranuni(1) * cnt;
      value2 = ranuni(2) * cnt;
      output;
   end;
run;
```

APPENDIX B --- REVERSE SORT

Memory usage between 2 and 4 leads generated:

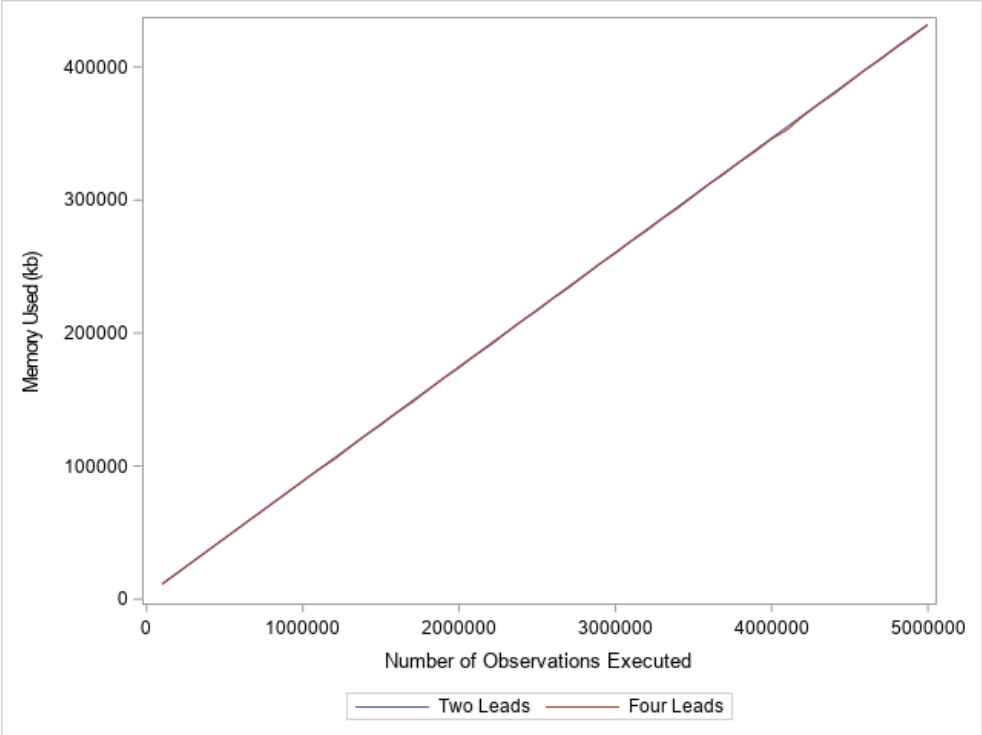


Timing between 2 and 4 leads generated:

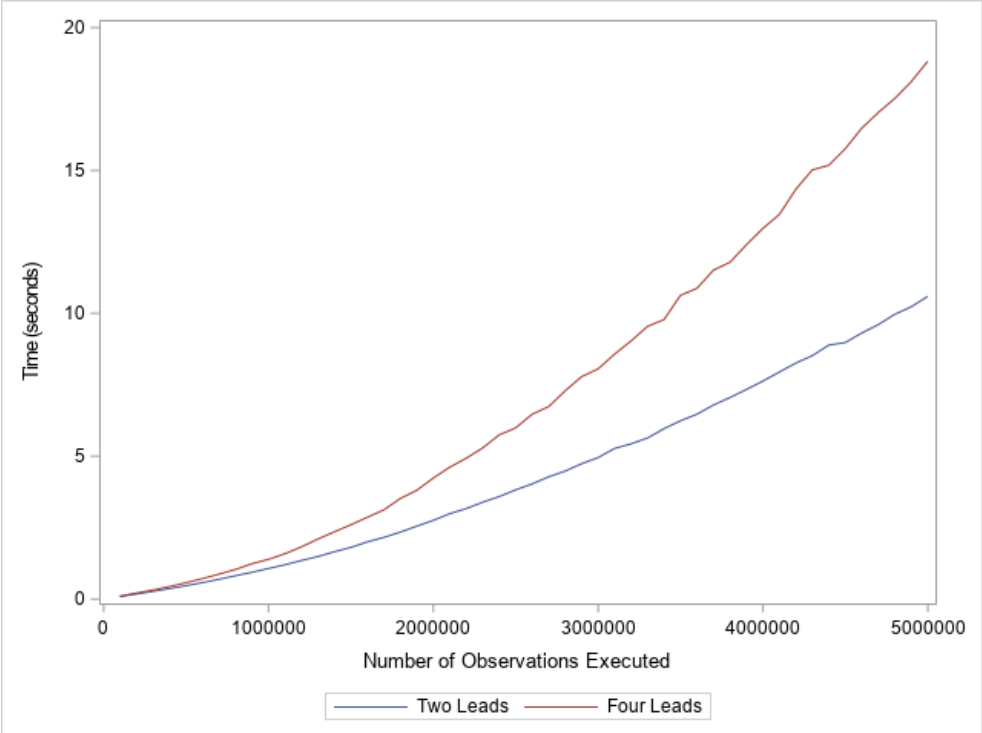


APPENDIX C --- PROC EXPAND

Memory usage between 2 and 4 leads generated:

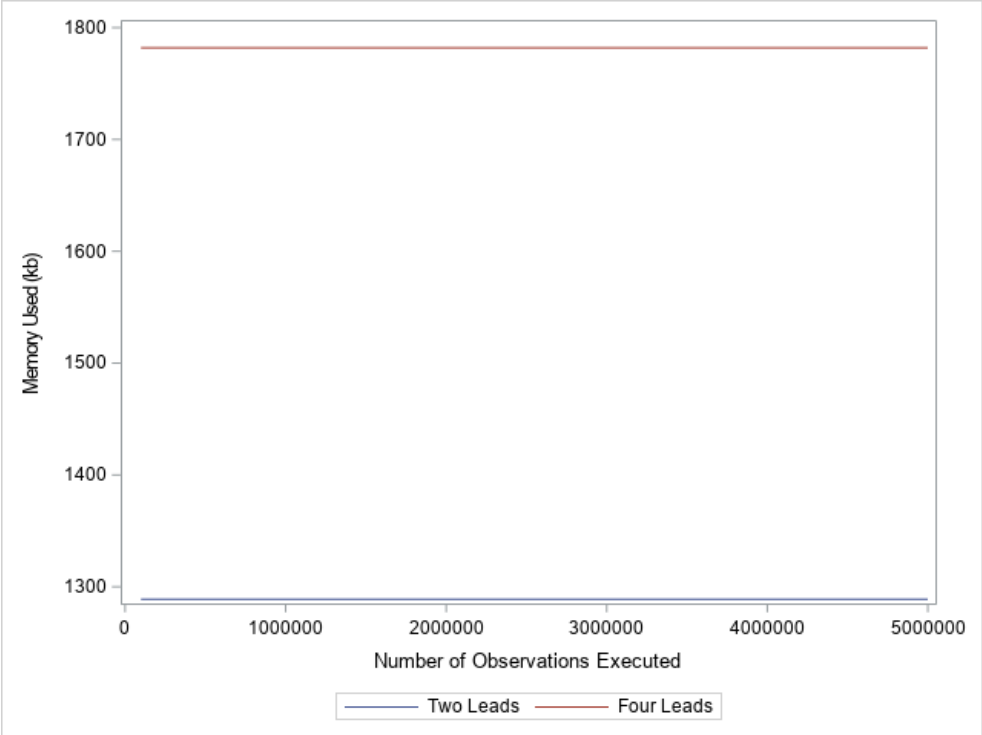


Timing between 2 and 4 leads generated:

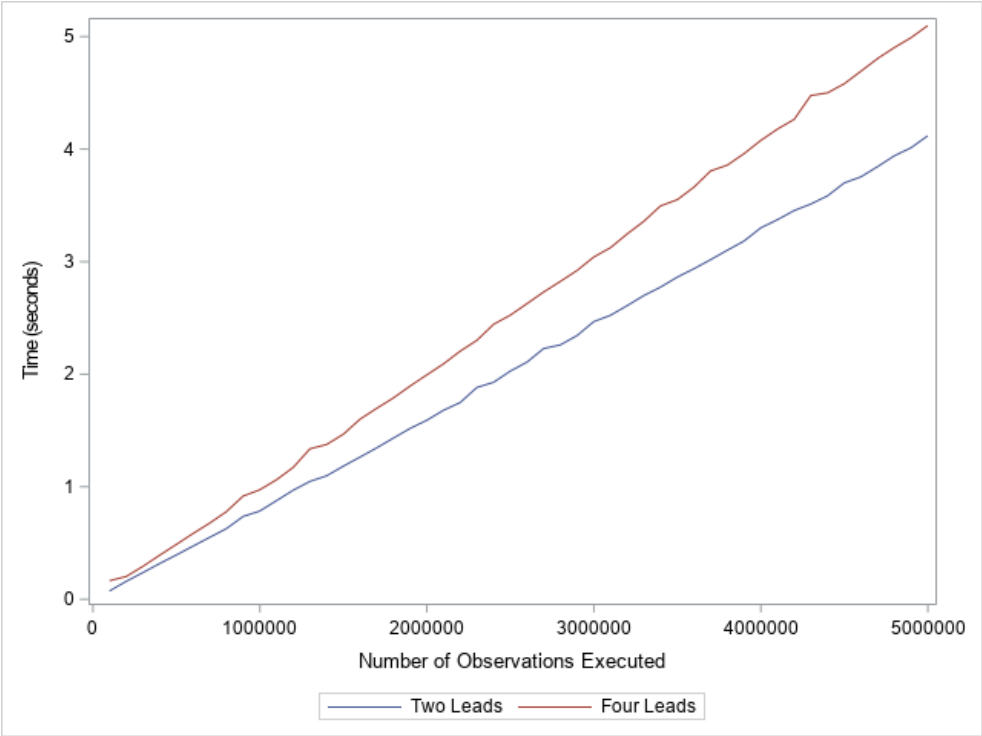


APPENDIX D --- DATA MERGE

Memory usage between 2 and 4 leads generated:

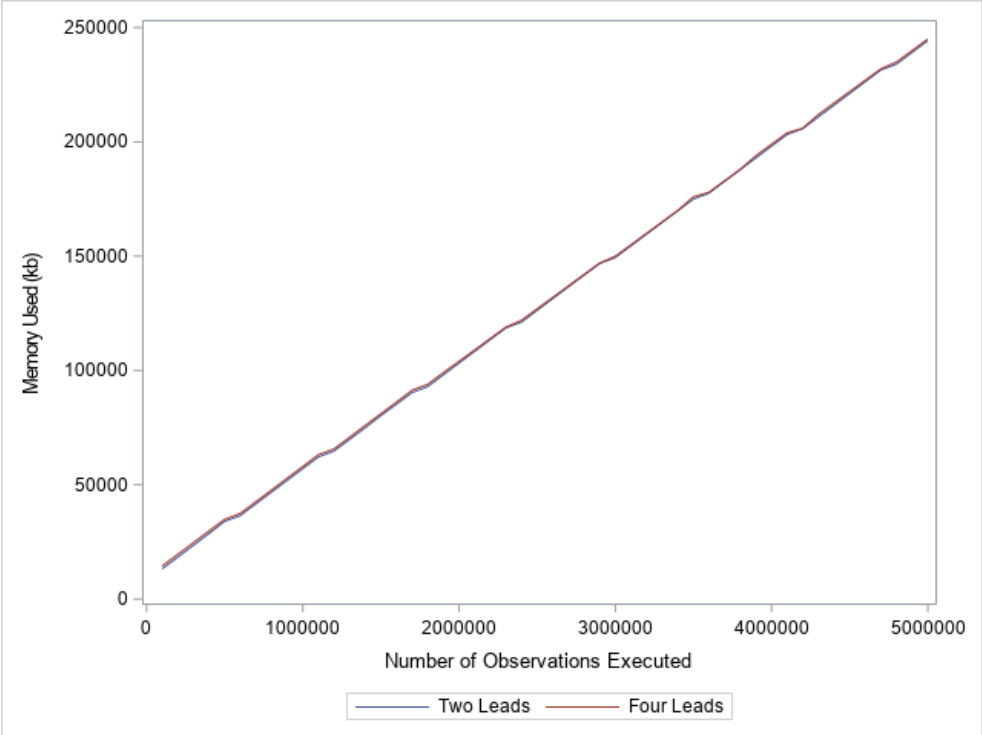


Timing between 2 and 4 leads generated:

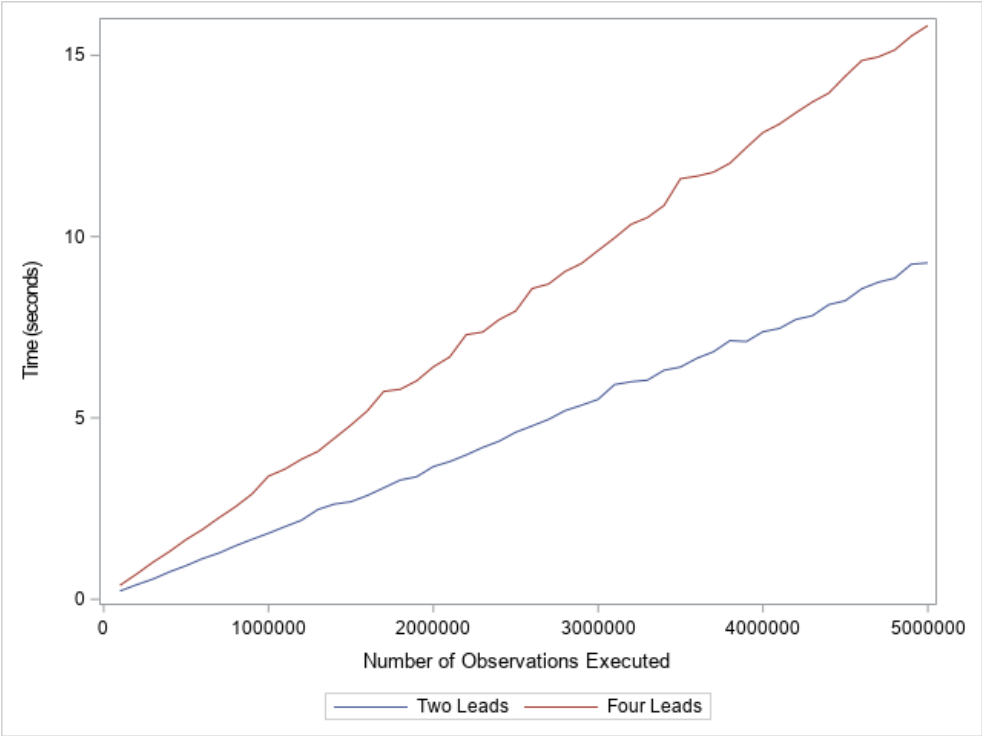


APPENDIX E --- PROC SQL JOIN

Memory usage between 2 and 4 leads generated:

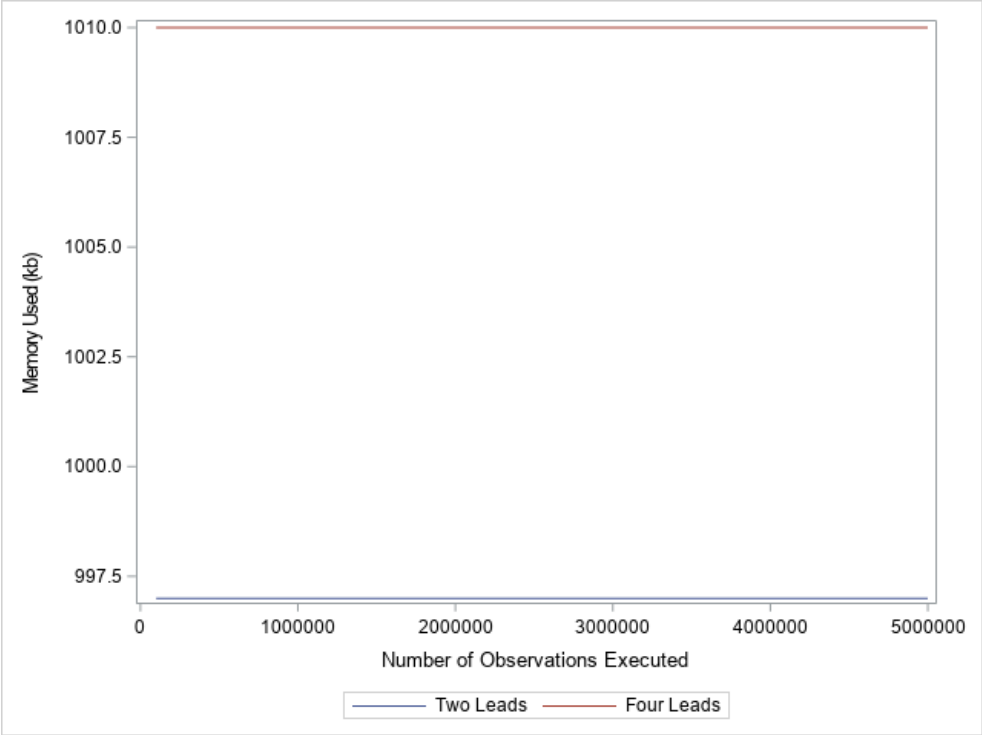


Timing between 2 and 4 leads generated:

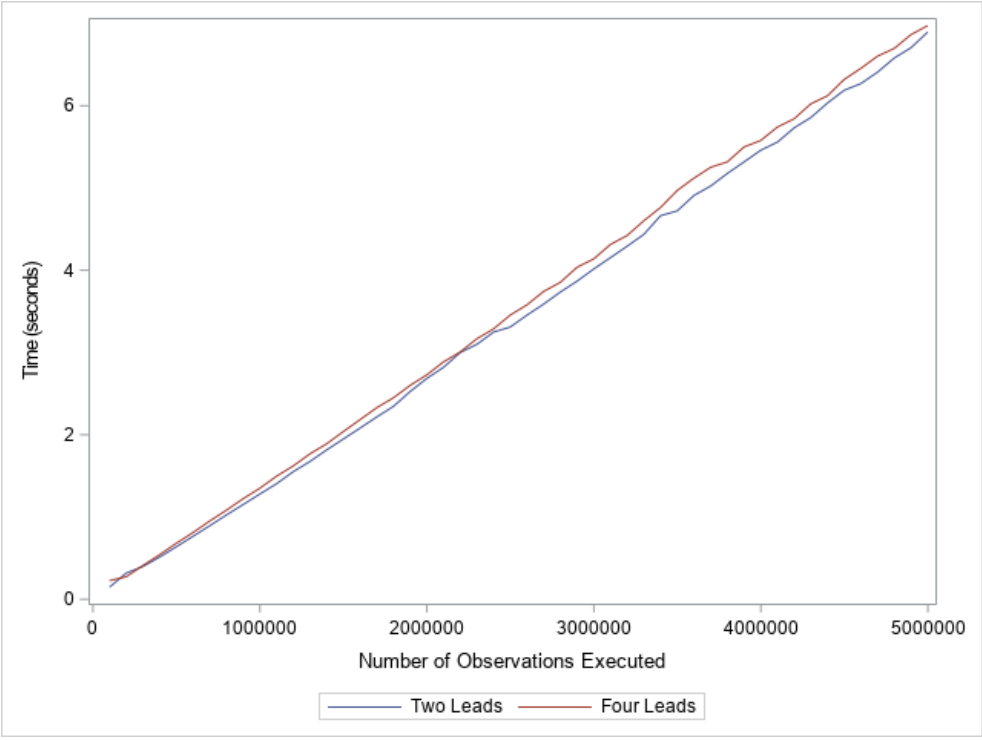


APPENDIX F --- DATA SET FETCING

Memory usage between 2 and 4 leads generated:

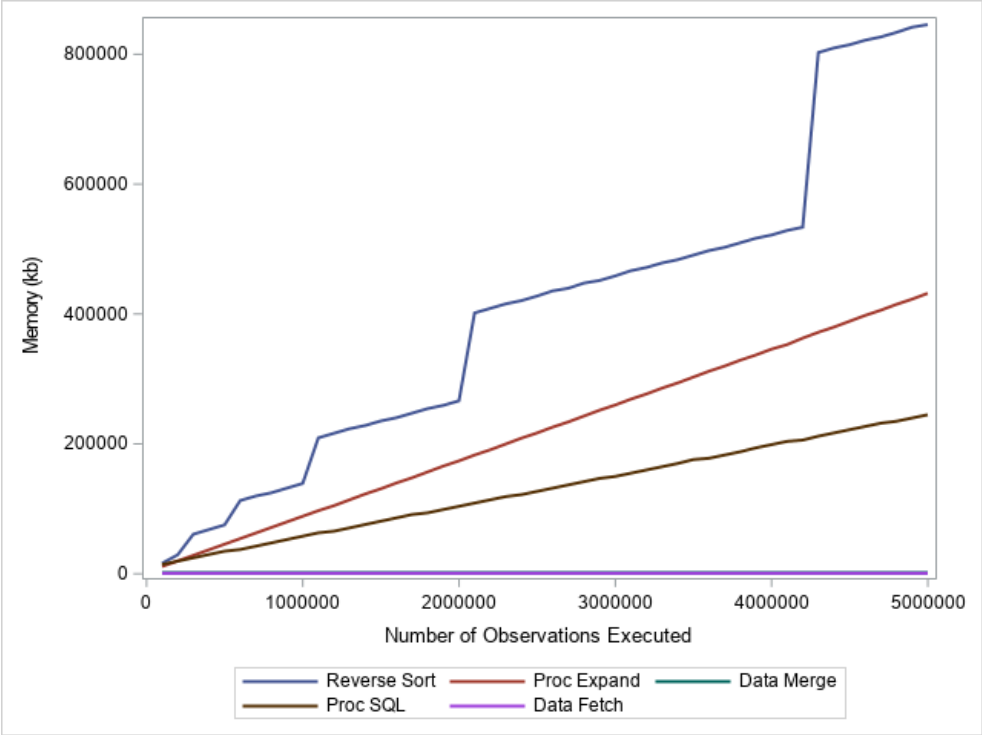


Timing between 2 and 4 leads generated:



APPENDIX G --- DIFFERENCES BETWEEN METHODS

Memory usage for 4 leads between different methods:



Timing for 4 leads between different methods: