Paper SAS2184-2018

Parallel Programming with the DATA Step: Next Steps

David Bultman and Jason Secosky, SAS Institute Inc., Cary, NC

ABSTRACT

The DATA step has been at the core of SAS® applications and solutions since the inception of SAS. The DATA step continues its legacy, offering its capabilities in SAS® Cloud Analytic Services (CAS) in SAS® Viya®. CAS provides parallel processing using multiple threads across multiple machines. The DATA step leverages this parallel processing by dividing data between threads and executing simultaneously on the data assigned to each thread. The LAG function, RETAIN statement, and the automatic variable _N_ require special attention when running DATA step programs in CAS. This paper describes each of these features, discusses how you can continue to use them with CAS, and compares performance with that in SAS® 9.4.

INTRODUCTION

The DATA step was written at a time when programs ran sequentially with one process. The notion of a thread was irrelevant because there was one thread and it was the process. Memory was tight and CPU speed was much slower than it is now. Today, threads are the norm and CAS runs on multiple machines with multiple threads simultaneously. The DATA step runs in CAS. What does it mean to run a DATA step in CAS?

The DATA step uses the Single Program Multiple Data paradigm (SPMD). When you are running the DATA step in CAS, each thread operates on part of the data. With a 1M row table, 10 workers, and 10 threads on each worker, each thread of the DATA step operates on approximately 10,000 rows. The benefit of running the DATA step in CAS, compared to SAS®, is that multiple cores are used and there is more I/O bandwidth when using multiple machines. In other words, "*it's a lot faster.*"

Despite some limitations, there are many DATA step programs that plug and play into CAS. For example, when using BY GROUPS in CAS, often the program needs little to no modification. There are other cases where using a two-step process allows you to use your existing code and take advantage of the power of CAS.

This paper focuses on SPMD concepts when using _N_, RETAIN, and LAG with the DATA step in CAS. It discusses when you can reuse your code without change and when you need to customize your code for the SPMD paradigm.

THE DATA STEP'S ACTION SET

A typical CAS server hosts many hundreds of actions. CAS organizes actions into groups called *action sets*. You can run the DATA step in CAS using two of the DATA step's actions: runCode and runCodeTable. You can also submit DATA step code to run a DATA step on CAS from SAS.

USING THE SESSREF= OPTION

You can run DATA step code in CAS from SAS by using the SESSREF= option in the DATA statement. The SESSREF= option specifies the CAS session to run the DATA step in:

```
cas casauto;
data _null_ / sessref=casauto;
  put 'Hello world from ' _threadid_=;
run;
```

Under the hood, when using SESSREF=, SAS compiles the DATA step code on the client and stores the compiled code in a stored program. The stored program is sent to the controller and distributed to all workers to run in all available threads.

If the DATA step includes a SET statement, each thread reads the data assigned to the thread. If the thread doesn't receive data, the thread completes.

The DATA step can automatically run in CAS even if SESSREF= is not specified. The DATA step runs in CAS if all the following are true:

1. All librefs in the step are CAS engine librefs to the same CAS session.
2. All statements in the step are supported by the CAS DATA step.
3. All functions, CALL routines, formats, and informats in the step are available in CAS.

You know your DATA step runs in CAS if you see the following in the log:

```
NOTE: Running DATA step in Cloud Analytic Services.
```

If you do not see the above message, your program did not run in CAS. Rather, it ran directly in SAS.

RUNCODE ACTION

The runCode action takes as input a string of DATA step code. The DATA step code is sent to the controller and the controller sends the code to each worker. The code is compiled on every thread for each node. Each thread runs the compiled program reading the data assigned to the thread. If you're running in MPP mode, only worker nodes are assigned data.

This paper uses the CAS procedure and the CASL language to submit DATA step programs using the runCode action. Mark Gass gives a brief tour of the programming interfaces for SAS, Java, Python, R, and REST (2018). You can call the DATA step from any of these programming languages via the runCode action. The programming language used in this paper is CASL. CASL is available via PROC CAS.

Here is what a simple DATA step program looks like using CASL and PROC CAS.

```
cas casauto;
proc cas;
  dataStep.runCode /
    code = "
      data _null_;
        put 'Hello world from ' _threadid_=;
      run;
    ";
    run;
quit;
```

RUNCODETABLE ACTION

In SAS® Viya® 3.3, the DATA step introduces another action to its *action set*, runCodeTable. The runCodeTable action runs a DATA step program stored in a CAS table. The CAS table can have a small footprint, including just the essential columns for the DATA step, or the CAS table can be compatible with the SAS Model Publishing action set. The use of runCodeTable is beyond the scope of this paper. See the Resources section for runCodeTable documentation.

USING RETAIN WITH THE DATA STEP IN CAS

The RETAIN statement tells the DATA step to hold a value from one row to the next without resetting it to missing. Traditionally, RETAIN is used to summarize a variable across all the rows of a table. For example, suppose you create the following data set:

```
data retain_example;
  do i = 1 to 100000;
    j=1;
    output;
  end;
run;
```

Then suppose you run the following code in SAS:

```
data retain_summary(drop=i j);
  retain sum_i sum_j;
  set retain_example end=done;
  sum_i + i;
  sum_j + j;
  if done then
    output;
run;
```

You see the following results:

Obs	sum_i	sum_j
1	5000050000	100000

If you run this DATA step in CAS, you discover that while this gets you close to a solution, it isn't a complete solution. Caution with retained variables is necessary because each thread retains the values that it sees. When you use a retained variable to hold a sum, the variable contains the sum of the values that that thread sees.

The following code is an example:

```
proc cas;
  upload path="/one/two/three/sgf2018/retain_example.sas7bdat";
  run;

  dataStep.runCode /
    code = "
      data retain_summary(drop=i j);
        retain sum_i sum_j;
        set retain_example end=done;
        sum_i + i;
        sum_j + j;
        if done then
          output;
        run;
    ";
    run;
quit;
```

It produces the following results:

```
Obs        sum_i       sum_j
 1       536854528     32767
 2      1610530817     32767
 3      2684207106     32767
 4       168457549      1699
```

In the example above, the DATA step in CAS divides the data between all available threads. Each thread summarizes the data it is assigned. Therefore, each thread produces an intermediate result that needs to be summarized. How do you sum the intermediate results? You invoke the fetch action to sum the intermediate results in this CASL code. You must tell the fetch action to fetch all the rows. Therefore, first call tableinfo to obtain the number of rows, and then call fetch to fetch, specifying the total number of rows in the intermediate results table.

```
proc cas;
  upload path="/one/two/three/sgf2018/retain_example.sas7bdat";

  dataStep.runCode /
    code = "
      data retain_summary(drop=i j);
        retain sum_i sum_j;
        set retain_example end=done;
        sum_i + i;
        sum_j + j;
        if done then
          output;
        run;
    ";
    run;

  tableinfo result=m / table="retain_summary"; run;
  nobs = m.tableinfo[1,"rows"];

  fetch result=local / fetchVars={{name="sum_i"}, {name="sum_j"}},
    table= {CASLib="CASUSER", name="retain_summary"}
    to=nobs maxrows=nobs;
  run;

 /* only needed if you want to view the results from fetch */
 localt=findtable(local);
 print localt;

quit;
```

The DATA step in CAS divides the data between the threads using the SPMD paradigm. In this example, the data was divided into four threads. Each thread received 32,767 rows except for thread four. Thread four received 1,699 rows.

```
localt: Results from table.fetch
Selected Rows from Table RETAIN_SUMMARY
_Index_       sum_i           sum_j
   1       536854528         32767
   2      1610530817         32767
```

```
    3       2684207106          32767
    4        168457549           1699
```

With just a little bit more CASL code, you can summarize the intermediate results and presto, produce the same end results as SAS.

```
  i=0;
  j=0;

  do x over localt;
    i = x.sum_i + i;
    j = x.sum_j + j;
  end;
  print "sum_i=" i;
  print "sum_j=" j;
quit;
```

```
sum_i=5000050000
sum_j=100000
```

In the cloud world with big data, the expectation is faster processing. The grid we use has six nodes: a controller, a backup controller, and four worker nodes. Each worker has 16 CPUs with hyperthreading turned on. Therefore, the DATA step runs with a maximum of 128 worker threads. Each worker node has over 180 gigabytes of available memory. The CPUs are *Intel(R) Xeon(R) CPU E5-2450 v2 @ 2.50GHz*. Using this grid, running with five billion observations, SAS took almost four minutes to perform the summarization, while CAS took 13 seconds.

Secosky provides another discussion on how to do this from a SAS Client (2017).

There is an easy way to get the same results in CAS: use SINGLE=YES:

```
libname mycas cas sessref=mysess;
data mycas.retain_summary(drop=i j) / single=yes;
  retain sum_i sum_j;
  set retain_example end=done;
  sum_i + i;
  sum_j + j;
  if done then
    output;
run;
```

This method is a reasonable approach for small to medium sized data sets. When you are using SINGLE=YES, all rows, both input and output, are moved to a single thread. For big data, you do not want to move the data to a single thread. This is discussed in more detail in the next section.

GENERATING A UNIQUE ID WITH THE DATA STEP IN CAS

Often assigning a unique ID to a row in a CAS table is required. The requirements of the unique ID can vary. Perhaps just a unique ID is needed without any order or starting point. Sometimes the IDs need to be consecutive. On occasion you might want consecutive IDs, starting from one for each row in your CAS table.

USING _N_ TO ASSIGN A SEQUENTIAL UNIQUE ID IN CAS

A traditional use for _N_ is to assign a unique, consecutive identifier to each observation in a table. Here's how you do this with SAS:

```
data new_dataset;
  set existing_dataset;
  uniqueid=_N_;
run;
```

As seen in the prior section, you can use SINGLE=YES with the DATA step in CAS to assign unique IDs to each row. However, as when using SINGLE=YES with RETAIN, this method is not recommended because all the rows are moved to a single thread. This pertains to both input data and output data. For substantial amounts of data, all the data might not fit on one worker. It also doesn't use the power of the DATA step in CAS.

```
libname mycas cas sessref=mysess;
data mycas.new_dataset / single=yes;
  set mycas.existing_dataset;
  uniqueid=_N_;
run;
```

If you decide SINGLE=YES performs well enough for your use case, you can use the PARTITION= data set option to partition the output on no variables. Partitioning on no variables redistributes the rows across all workers, as is done here:

```
data mycas.new_dataset(partition=());
```

This might be good enough for a one-off. However, if you decide SINGLE=YES is too slow, consider using all available threads in CAS instead of a single thread.

The default is SINGLE=NO. When you are using SINGLE=NO, each thread of the DATA step running in CAS operates on part of the data. With a 1M row table, 10 workers, and 10 threads on each worker, each thread of the DATA step operates on about 10,000 rows. You use 100 threads and each thread's uniqueid is 1 to 10,000.

There is a way to use all threads in CAS and assign a consecutive unique ID just like SAS. To assign a consecutive unique ID to each row in a table, each thread needs to know how many rows or observations were assigned to the other threads. Once you know how many rows are assigned to each thread, you can generate a DATA step and run the generated program using PROC CAS and the DATA step's runCode action. The steps to do this are as follows:

1. Upload your data to CAS if it's not already there.
2. Determine how many rows were assigned to each thread and put results in a table (metadata).
3. Generate DATA step code with CASL using the table (metadata) generated in step 2.
4. Run the code generated in step 3.

The metadata is used along with some CASL code to generate some DATA step code. The generated DATA step code processes the data in parallel and produces identical results to SAS.

```
proc cas;
```

```
  /* 1. Upload your data to CAS */
  upload path="/one/two/three/sgf2018/adds.sas7bdat";

  /* 2. Determine how many rows were assigned to each thread
        and put results in a table */
  dataStep.runCode /
    code = "
      data metadata(keep=n t);
        set adds end=done;
        if done then do;
          t=_THREADID_;
          n=_N_;
          output;
        end;
      run;
    ";
  run;

  /* 3. Generate DATA step code using CASL and
        the table generated in step 2 */
  tableinfo result=m / table="metadata"; run;
  nobs = m.tableinfo[1,"rows"];

  fetch result=local / fetchVars={{name="n"},{name="t"}},
    table={CASLib="CASUSER", name="metadata"}
    to=nobs maxrows=nobs;
  run;

  localt=findtable(local);

  codeExpr = 'data addN; set adds; select (_THREADID_);';

  prevcount=0;
  newcount=0;

  do x over localt;
    newcount=newcount+prevcount;
    codeExpr=
      codeExpr ||
      " when ( " || x.t || ") unique= " || newcount || "+ _N_;";
    prevcount=x.n;
  end;

  codeExpr = codeExpr || " otherwise put 'PROBLEM ' _THREADID_=; end; ";

  /* Debug code to checkout generated code */
  print codeExpr;

  /* 4. Run the code generated in step 3 */
  dataStep.runCode / code=codeExpr; run;

  /* Debug code to verify distinct values equals number of rows in table */
  distinct result=output table={name='addN'};
  print output.distinct[,1:2];
quit;
```

Step 3 might require a little more explanation. On the grid when using 100,000 rows, the metadata table contains the following:

```
Selected Rows from Table METADATA
_Index_          n               t
   1          32766             1
   2          32766            33
   3          18085            65
   4          16383            97
```

There is a row for each thread receiving data. In this case there are four threads that receive data. The thread number is represented by the column "t". The column "n" is the number of rows assigned to thread "t".

A SELECT statement is generated and each thread is given a formula to calculate their respective values for _N_. The formula is to add the sum of the previous thread's row count to the value of _N_ on the thread.

In other words, for thread 1, use the value of _N_, by adding zero to _N_. For thread 33, add the total number of rows assigned to thread 1, plus the value of _N_. For thread 65, add the sum of the total number of rows assigned to thread 1 and 33 to the value of _N_. Finally, for thread 97, add the sum of the total number of rows assigned to thread 1, 33, and 65 to the value of _N_.

The following code is generated:

```
data addN;
  set adds;
  select (_THREADID_);
  when ( 1)  unique= 0+ _N_;
  when ( 33) unique= 32766+ _N_;
  when ( 65) unique= 65532+ _N_;
  when ( 97) unique= 83617+ _N_;
  otherwise put 'PROBLEM ' _THREADID_=;
  end;
```

If you scale the above example up, from 100,000 rows to .5 billion, on the grid with four workers, CAS uses 128 threads. Each thread receives about 390,000 rows. The total time for adding the unique ID to .5 billion rows took an average of 19 seconds. This is five times faster than doing the operation with SAS on a sas7bdat file.

GENERATE A NONSEQUENTIAL UNIQUE ID IN CAS

If you want to generate a unique ID for each row in your CAS table and do not care if the unique IDs are consecutive, contiguous, or related to _N_, you can do the following:

```
data mycas.new_dataset;
  if _N_ = 1 then do;
    _mult = 10 ** (int(log10(_NTHREADS_)) + 1);
    retain _mult;
    drop _mult;
 end;
 set mycas.existing_dataset;
 rowid = _THREADID_ + (_N_ * _mult);
run;
```

This method places the thread number in the lower digits of ROWID, and places the row number in the upper digits. The IDs are unique. However, they are not sequential, nor do they start from 1.

If you need a unique ID, however, you do not need the IDs to be consecutive or start from a particular number. This technique has the advantage of being one-step. In other words, one and done.

You have three methods available to you, if you need a consecutive unique ID:

- Pull the data out of CAS, use SAS, and then reload into CAS. Loading the sas7bdat file with .5 billion rows into CAS, on the example grid, took approximately 3 minutes and 15 seconds.
- Use the SINGLE=YES option and then redistribute your data with the PARTITION= data set option.
- Leave your data in CAS and use the two-step process. If your data is already in CAS, it is very efficient to use this technique.

USING LAG WITH THE DATA STEP IN SAS VIYA

WITH BY GROUPS

The LAG function allows access to a variable's value from the previous observation and enables computations across observations. The following SAS example is like the example used in Yunchao Tian's paper (2009). The example contains four columns:

```
Alphabetic List of Variables and Attributes
Variable  Type  Len   Format
Price     Num   8     DOLLAR10.2
Product   Num   8     Z8.
Sold      Num   8
Year      Num   8
```

The years range from 1950-2017 for each product. Each product has a price and number of items sold for each year. Here's the first 10 observations for a query where sold is equal to 40:

Obs	product	price	year	sold
1	77002304	$220.69	1990	40
2	30307200	$1.99	1950	40
3	30307200	$8.39	1990	40
4	30307200	$8.99	1992	40
5	55023136	$16.59	1960	40
6	55023136	$127.49	2015	40
7	03597040	$86.49	1958	40
8	54984976	$162.29	2003	40
9	54984976	$186.39	2010	40
10	54984976	$202.59	2015	40

In the paper, Tian asserts a common use of LAG is to determine the increase of price from one time period to another and offers code like this for SAS:

```
proc sort data=example_data;
  by product year;
run;

data lag_sas_results;
```

```
  set example_data;
  retain product year sold price lag_price dif_price per_increase;
  format lag_price dif_price dollar10.2 per_increase percent10.2;
  by product year;
  lag_price = lag(price);
  dif_price = price - lag_price;
  if first.product then do;
    lag_price =.;
    dif_price =.;
    per_increase =.;
  end;
  per_increase = dif_price/lag_price;
run;
```

The first 10 observations for product 00037928 are shown here:

Obs	product	price	year	Sold	lag_price	dif_price	per_increase
1	00037928	$51.99	1950	82	.	.	.
2	00037928	$56.09	1951	134	$51.99	$4.10	7.89%
3	00037928	$57.39	1952	106	$56.09	$1.30	2.32%
4	00037928	$57.89	1953	176	$57.39	$0.50	0.87%
5	00037928	$58.09	1954	146	$57.89	$0.20	0.35%
6	00037928	$56.49	1955	150	$58.09	$-1.60	(2.75%)
7	00037928	$57.29	1956	104	$56.49	$0.80	1.42%
8	00037928	$59.19	1957	92	$57.29	$1.90	3.32%
9	00037928	$60.79	1958	102	$59.19	$1.60	2.70%
10	00037928	$61.39	1959	138	$60.79	$0.60	0.99%

How does this translate into CAS? First, let's consider SAS. To process BY groups in SAS, you use the SORT procedure to group rows by the BY variables. After the rows are grouped, the BY statement in the DATA step creates special FIRST. and LAST. variables to detect the start and end of each BY group. The BY statement also allows rows to be interleaved or combined with the SET or MERGE statements.

When running a DATA step in CAS, everything is the same, except you do not sort the data before running the DATA step. When using a BY statement in a CAS DATA step, CAS groups and orders the data on the fly, and complete BY groups are given to a thread for processing. With multiple threads, multiple BY groups are processed at the same time.

Because a thread of DATA step processes an entire BY group, the SAS code can run in CAS unmodified:

```
proc cas;
   upload path="/one/two/three/sgf2018/lag.sas7bdat";
   dataStep.runCode /
    code = "data lag_cas_results;
              set lag;
              retain product year sold price
                     lag_price dif_price per_increase;
              format lag_price dollar10.2
                     dif_price dollar10.2
                     per_increase percent10.2;
              by product year;
              lag_price = lag(price);
              dif_price = price - lag_price;
              if first.product then do;
                lag_price = .;
                dif_price = .;
                per_increase = .;
```

```
              end;
              per_increase = dif_price/lag_price;
            run;";
   run;
quit;
```

The SPMD paradigm fits nicely into this use case. The data is divided between threads where each thread gets a unique set of BY groups. Because a thread of DATA step processes an entire BY group, the SAS code can run in CAS unmodified and produces identical results.

The above example is written with runCode. However, you could submit the code directly from SAS using the DATA step. If you use the DATA step to submit your code to CAS, your program automatically runs in CAS if you are connected to a CAS session and all data sets specified in the DATA step program are in CAS. Whether you choose runCode or a DATA step, your existing DATA step code is reusable.

WITHOUT BY GROUPS

The LAG function without BY groups is more challenging. The reason is, SAS guarantees an order of observations in a SAS data set. In CAS, there isn't the notion of an observation number. If you run a program like the following in SAS, uniqueid equals the observation number. Therefore, there is one occurrence of uniqueid equal to 1.

```
data existing_dataset;
  do i = 1 to 100000;
    j=1;
    output;
  end;
run;

data new_dataset;
  set existing_dataset;
  uniqueid=_N_;
  thread=_THREADID_;
run;

data one;
  set new_dataset(where=(uniqueid=1));
run;

proc print data=one; run;
```

Obs	i	j	uniqueid	thread
1	1	1	1	1

In CAS, because _N_ is initialized to 1 for each thread, uniqueid equals 1 for as many threads running the program. In this program, on the previous described grid, CAS DATA step uses seven threads to create new_dataset.

```
cas mysess;
libname mycas cas caslib=casuser;

data mycas.existing_dataset / single=yes sessref=mysess;
  do i = 1 to 100000;
    j=1;
    output;
```

```
  end;
run;

data mycas.new_dataset;
  set mycas.existing_dataset;
  uniqueid=_N_;
  thread=_THREADID_;
run;

data one;
  set mycas.new_dataset(where=(uniqueid=1));
run;

proc print data=one; run;
```

Obs	i	j	uniqueid	thread
1	65537	1	1	69
2	1	1	1	65
3	49153	1	1	68
4	16385	1	1	66
5	32769	1	1	67
6	81921	1	1	70
7	98305	1	1	71

If you want to match CAS results with SAS, using LAG without BY groups, you need to assign a uniqueID in CAS that is equivalent to the observation number in SAS. Once you decide on the observation order, you can run a two-step process and successfully match results with LAG. To demonstrate this, let's generate this hypothetical data set with SAS:

```
data example;
  do uniqueid = 1 to 100000;
    target = round(rand("Uniform")*100000000, 8);
    output;
  end;
run;
```

The goal of this exercise is to match results of the following SAS code in CAS:

```
data new_dataset;
  set example;
  previous = lag(target);
  difference = target - previous;
run;
```

If you use this DATA step code with the runCode action, you find the results do not match:

```
proc cas;
   upload path="/one/two/three/sgf2018/example.sas7bdat";
   dataStep.runCode /
    code = "data new_dataset;
              set example;
              previous = lag(target);
              difference = target - previous;
            run;
           ";
   run;
```

```
quit;
```

The reason is, the input data set is divided between threads and each thread's initial value for the LAG function is missing. When you run in SAS, only the first observation has a missing value. When you run in CAS, the following observations contain missing values because the data is assigned to four threads:

Obs	uniqueid	target	previous	difference
1	1	36878336	.	.
32768	32768	86488408	.	.
65535	65535	99096880	.	.
98302	98302	11575312	.	.

To make things work like SAS, you need to know the value of the variable "target" for each thread's last observation. You can do this with the following code:

```
dataStep.runCode /
    code = "data metadata;
              set example end=done;
              if done then
                output;
             run;
            ";
    run;
```

The metadata table contains the last value of target for each thread:

Obs	uniqueid	target
1	32767	65323496
2	65534	97134312
3	98301	48450440
4	100000	99330112

Once you know the value of the last observation for each thread for the target variable, you can then generate code that checks the overall observation number, uniqueid, and, if uniqueid+1 is found, substitute the generated value in place of the missing value returned by the lag function. In the end, you generate the following code:

```
data match_sas;
  set example;
  previous = lag(target);
  if _N_ = 1 then do;
    select (uniqueid);
      when (32768)  previous = 53040360;
      when (65535)  previous = 87070688;
      when (98302)  previous = 507280;
      when (100001) previous = 17913816;
      otherwise;
    end;
  end;
  difference = target-previous;
run;
```

The entire test example follows:

```
/* 1. SAS code */
libname mylocal '/one/two/three/sgf2018';
```

```
data mylocal.example;
  do uniqueid = 1 to 1000000;
    target = round(rand("Uniform")*100000000, 8);
    output;
  end;
run;

data mylocal.example_sas;
  set mylocal.example;
  previous = lag(target);
  difference = target - previous;
run;

proc sort data=mylocal.example_sas; by uniqueid; run;

/* 2. CAS code */
cas mysess sessopts=(caslib=caslib);
libname mycas cas sessref=mysess;
libname mylocal '/one/two/three/sgf2018';

proc cas;
  upload path="/one/two/three/sgf2018/example.sas7bdat";

  dataStep.runCode /
    code = "data metadata;
              set example end=done;
              if done then
                output;
            run;
           ";
  run;

  tableinfo result=m / table="metadata"; run;
  nobs = m.tableinfo[1,"rows"];

  fetch result=local /
    fetchVars={{name="target"},{name="uniqueid"}},
    table={caslib="casuser", name="metadata"}
    to=nobs maxrows=nobs;
  run;

  localt=findtable(local);

  codeExpr = "data match_sas; set example; ";
  codeExpr = codeExpr || "previous = lag(target);";
  codeExpr = codeExpr || "if _N_ = 1 then do;";
  codeExpr = codeExpr || "select (uniqueid);";

 do x over localt;
   codeExpr =
     codeExpr ||
     " when ( " || $(x.uniqueid+1) || ") previous= " || x.target || ";";
 end;

  codeExpr = codeExpr || " otherwise; end; end;";
  codeExpr = codeExpr || " t=_THREADID_; ";
```

```
  codeExpr = codeExpr || " difference = target-previous; run;";

  print codeExpr;
  dataStep.runCode / code=codeExpr;
quit;

/* 3. TEST code (SAS-side) */
data mylocal.example_meta;
  set mycas.metadata;
run;

data mylocal.example_cas;
  set mycas.match_sas;
run;

proc sort data=mylocal.example_cas; by uniqueid; run;
proc compare base=mylocal.example_sas compare=mylocal.example_cas; run;
```

```
                          Observation Summary

                     Observation      Base  Compare

                     First Obs           1        1
                     Last  Obs      100000   100000

Number of Observations in Common: 100000.
Total Number of Observations Read from MYLOCAL.EXAMPLE_SAS: 100000.
Total Number of Observations Read from WORK.EXAMPLE_CAS: 100000.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 100000.

NOTE: No unequal values were found. All values compared are exactly equal.
```

You can expand the LAG example for LAGn, where n=1 to 100, by adding more metadata and generating code dependent on your metadata and use case. Whenever you use a two-step technique, the advantage is that your data stays in CAS, you operate with multiple threads, and you use the power of CAS.

CONCLUSION

We have explored running the DATA step in CAS with special attention to RETAIN, LAG, and _N_. We have seen that using BY groups with distributed data in CAS often allows you to plug and play your SAS DATA step code. This allows you to use your existing DATA step code to operate on big data more efficiently.

We've also explored ways to use your existing DATA step code by customizing it for a two-step process. By generating code in the first step and running the code in the second step, you can retain your SAS code, increase your data size, and run faster than ever before.

ACKNOWLEDGMENTS

We would like to thank those involved in the development, testing, documentation, and support of the DATA step and CASL in CAS: Melissa Corn, Hua Rui Dong, Jerry Pendergrass, Lisa Davenport, Jane Eslinger, Kevin Russell, Mark Gass, Jerry Pendergrass, Steve Krueger, Caroline Brickley and Oliver Schabenberger.

REFERENCES

Gass, Mark. 2018. “Cloud Analytic Services Actions: A Holistic View.” *Proceedings of the SAS Global Forum 2018 Conference*. Cary, NC: SAS Institute Inc. Available http://support.sas.com/resources/papers/proceedings18/SAS1981-2018.pdf.

Secosky, Jason. 2017. “DATA Step in SAS Viya: Essential New Features.” *Proceedings of the SAS Global Forum 2017 Conference.* Cary, NC: SAS Institute Inc., Available http://support.sas.com/resources/papers/proceedings17/SAS0118-2017.pdf.

Tian, Yunchao (Susan). 2009. “LAG - the Very Powerful and Easily Misused SAS Function.” *Proceedings of the SAS Global Forum 2009 Conference.* Cary, NC: SAS Institute Inc. Available http://support.sas.com/resources/papers/proceedings09/055-2009.pdf.

RESOURCES

- SAS Institute Inc. 2018. *SAS Getting Started with CASL Programming 3.3.* Cary, NC: SAS Institute Inc. Available http://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.3&docsetId=casl&docsetTarget=titlepage.htm&locale=en.

- SAS Institute Inc. 2018. *SAS Cloud Analytic Services 3.3: CASL Reference*. Cary, NC: SAS Institute Inc. Available http://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.3&docsetId=proccas&docsetTarget=titlepage.htm&locale=en.

- SAS Institute Inc. 2018. *SAS DATA Step Action Set: Syntax, runCodeTable Action*. Cary, NC: SAS Institute Inc. Available http://go.documentation.sas.com/?docsetId=caspg&docsetTarget=cas-datastep-runcodetable.htm&docsetVersion=3.3&locale=en.

- SAS Institute Inc. 2018. *SAS DATA Step Action Set: Examples, runCodeTable Action*. Cary, NC: SAS Institute Inc. Available:
  - http://go.documentation.sas.com/?docsetId=caspg&docsetTarget=n0lfj0o5vz2702n1w3wi71xanknm.htm&docsetVersion=3.3&locale=en
  - http://go.documentation.sas.com/?docsetId=caspg&docsetTarget=n1rbaldyj1x2w9n1qqhimihqdk63.htm&docsetVersion=3.3&locale=en

CONTACT INFORMATION

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

David Bultman
100 SAS Campus Drive
Cary, NC 27513
SAS Institute Inc.
David.Bultman@sas.com
http://www.sas.com

Jason Secosky
100 SAS Campus Drive
Cary, NC 27513
SAS Institute Inc.
Jason.Secosky@sas.com
http://www.sas.com