MWSUG 2017 - Paper BB129-SAS

DATA Step in SAS® Viya™: Essential New Features

Jason Secosky, SAS Institute Inc., Cary, NC

ABSTRACT

The DATA step is the familiar and powerful data processing language in SAS® and now SAS Viya™. The DATA step's simple syntax provides row-at-a-time operations to edit, restructure, and combine data. New to the DATA step in SAS Viya are a varying-length character data type and parallel execution. Varying-length character data enables intuitive string operations that go beyond the 32KB limit of current DATA step operations. Parallel execution speeds the processing of big data by starting the DATA step on multiple machines and dividing data processing among threads on these machines. To avoid multi-threaded programming errors, the run-time environment for the DATA step is presented along with potential programming pitfalls. Come see how the DATA step in SAS Viya makes your data processing simpler and faster.

INTRODUCTION

The DATA step is a programming language to prepare tables for analysis. It excels at modifying existing values and computing new values in a row, as well as combining tables. In SAS, a DATA step runs in a single thread or core on your system.

This is where our problem lies. With big data, running in a single thread is slow. Some DATA steps can take hours to complete. With SAS Cloud Analytics Services (CAS) in SAS Viya, we have an environment where tens or hundreds of threads are available, across several machines. When you program your DATA steps to run in CAS, all of those threads become available for you to use to improve the performance with massive parallel processing.

For detailed information about CAS, please refer to *SAS® Cloud Analytic Services 3.1: Fundamentals.* A link to this document is in the Resources section of this paper.

RUNNING A DATA STEP IN PARALLEL IN CAS

How do you program your DATA steps to run in CAS? In many ways, you program just like you would in SAS, taking advantage of your experience as a SAS programmer. Where DATA steps run is driven by where your tables are stored. If your tables are stored in SAS, the DATA step runs in SAS. If your tables are stored in CAS, the DATA step runs in CAS. If the tables are in a mix of locations, then the DATA step runs in SAS. Running "in SAS" means the DATA step runs on the SAS client, not in CAS.

In this paper, we inspect several DATA steps, learn where they run (in SAS or in CAS), and understand how running in a single thread in SAS is different from running in many threads in CAS.

Below is a figure of what it looks like to run DATA steps in CAS. In this diagram, a table is stored in CAS and several threads of a DATA step operate on the table. The same DATA step program runs in each thread.

How does running the same program in each thread improve performance? CAS splits up the table so that each thread independently works on part of the table. Splitting the table among threads is where our parallelism and speedup come from. Instead of having one thread work on the entire table, you have many threads, each working on part of the table.

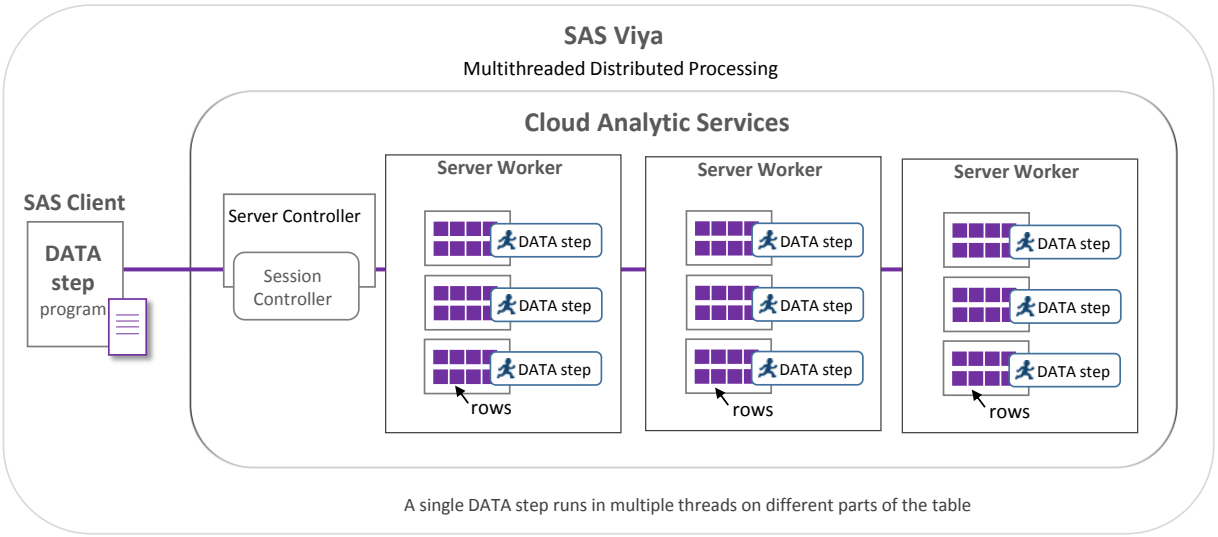

Figure 1. DATA Step Running in SAS Cloud Analytic Services (CAS)

In the next sections, we look at common DATA steps for preparing data sets for analytics. For most of these examples, we run a DATA step in SAS then modify the DATA step to run in CAS. The DATA steps we run perform these actions:

- Print "hello world" from each thread
- Load a table from SAS into CAS
- Compute a new variable value for each row
- Sum a value across an entire table
- Sum a value for BY groups within the data
- Operate on varying-length character data

PRINT "HELLO WORLD" FROM EACH THREAD

In this example, we run a program that prints a "hello" message from SAS and then CAS.

```
/* "Hello World" in SAS */
data _null_;
   put 'Hello world from ' _threadid_=;
run;

/* Create a CAS session to run DATA step in */
cas casauto;

/* "Hello World" in CAS */
data _null_ / sessref=casauto;
   put 'Hello world from ' _threadid_=;
run;
```

```
 61         /* "Hello World" in SAS */
 62         data _null_;
 63            put 'Hello world from ' _threadid_=;
 64         run;
```

```
Hello world from _THREADID_=1

65
66          /* Create a CAS session to run DATA step in */
67          cas casauto;
68
69          /* "Hello World" in CAS */
70          data _null_ / sessref=casauto;
71             put 'Hello world from ' _threadid_=;
72          run;
NOTE: Running DATA step in Cloud Analytic Services.
Hello world from _THREADID_=9
Hello world from _THREADID_=2
Hello world from _THREADID_=4
Hello world from _THREADID_=5
Hello world from _THREADID_=8
Hello world from _THREADID_=6
Hello world from _THREADID_=3
Hello world from _THREADID_=10
Hello world from _THREADID_=1
Hello world from _THREADID_=11
Hello world from _THREADID_=15
Hello world from _THREADID_=12
Hello world from _THREADID_=14
Hello world from _THREADID_=16
Hello world from _THREADID_=7
Hello world from _THREADID_=13
```

Output 1. "Hello World" Running in Several Threads

Notice the new automatic variable, _THREADID_, in the DATA step in SAS Viya. This variable is filled with the thread number of the thread running the DATA step. Thread numbers start at 1 and go up to N, where N is the maximum number of threads available.

In the program that runs in SAS, there is one-line output because only one thread is used.

In the program that runs in CAS, the note indicates the program ran in CAS. The next set of lines shows the program running in all available threads. There are 16 threads available, each runs the same program, so each outputs a line to the log. If you run the CAS program multiple times, you will see the output is in a different order. Why would the output be in a different order?

All of the threads operate independently. This means log messages are generated by each thread at slightly different times, resulting in slightly different ordering for each run. If the DATA step were to somehow synchronize the output of log messages, we would lose the performance gain that threads give.

Remember we said the DATA step runs in CAS if the tables that it operates on are stored in CAS? This program does not operate on any tables, so we have to use the SESSREF= option in the DATA statement to force the DATA step to run in CAS. Without the SESSREF= option, the DATA step runs in SAS.

In this first example, we have seen a DATA step running in SAS in a single thread and the same DATA step running in CAS in multiple threads. Each thread operates independently, so there is no ordering of the thread output.

LOAD A TABLE FROM SAS INTO CAS

In this example, we load a SAS data set into a CAS table using two methods. The first is a DATA step using the CAS engine. The second uses the LOAD statement of the CASUTIL Procedure.

```
/* Setup libraries we are going to use */
```

```
/* One for SAS data, one for CAS data */
libname sas '<path-to-data-sets>';
libname cas cas;

/* Load table into CAS with DATA step and the CAS engine */
data cas.purchase;
   set sas.purchase;
run;

/* Load table into CAS with PROC CASUTIL */
proc casutil;
  load data=sas.purchase casout="purchase" replace;
quit;
```

```
 62         /* Load table into CAS with DATA step and the CAS engine */
 63         data cas.purchase;
 64            set sas.purchase;
 65         run;

 NOTE: There were 192735 observations read from the data set SAS.PURCHASE.
 NOTE: The data set CAS.PURCHASE has 192735 observations and 19 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.17 seconds
       cpu time            0.08 seconds

 66
 67         /* Load table into CAS with PROC CASUTIL */
 68         proc casutil;
 NOTE: The UUID '369001c7-acce-4644-bee7-bd24cfd743a1' is connected
       using session CASAUTO.
 69           load data=sas.purchase casout="purchase" replace;
 NOTE: SAS.PURCHASE was successfully added to the "CASUSER(<userid>)"
       caslib as "purchase".
 70         quit;

 NOTE: PROCEDURE CASUTIL used (Total process time):
       real time           0.21 seconds
       cpu time            0.09 seconds
```

Output 2. Two Ways to Load Data into CAS

In this example, the DATA step and PROC CASUTIL perform the same operation. When would you select one over the other? Both DATA step and PROC CASUTIL support selecting rows and columns to add to the new table. If more data processing needs to occur while loading, the DATA step can be used to load and transform with one step. If additional processing is not necessary, the intent of loading data is clearer when using the LOAD statement with PROC CASUTIL.

Also notice when the DATA step completes, there is no message about running the DATA step in CAS. Because the input table is stored in a SAS library and not in CAS, the DATA step runs in SAS.

COMPUTING A NEW VARIABLE

With a table loaded into CAS, we can perform some operations on it. In this example, we create a new variable, CUST_ACTIVITY, whose values are *High*, *Medium*, *Low*, or *Unknown*, based on one of the categorical variables, CAT_INPUT1. The program first runs in SAS, then we change the DATA step to run in CAS by changing the librefs to indicate the data are stored in CAS.

The program also writes to the log the number of rows each thread operates on. This diagnostic message helps us to understand how each thread operates on the part of data assigned to it.

```
/* Use DATA step to transform a value in a row */
/* Run in both SAS and CAS to see how many rows each thread processes */

/* SAS */
data sas.purchase_activity;
   length cust_activity $ 8;

   set sas.purchase end=done;

  /* Transform value */
   select (cat_input1);
   when ('X') cust_activity = 'High';
   when ('Y') cust_activity = 'Medium';
   when ('Z') cust_activity = 'Low';
   otherwise  cust_activity = 'Unknown';
   end;

  /* Output number rows this thread processed */
   if done then
      put _threadid_=z2. _N_=;
run;

/* CAS */
/* To run in CAS, SAS librefs changed to CAS librefs. */
data cas.purchase_activity;
   length cust_activity $ 8;

   set cas.purchase end=done;

  /* Transform value */
   select (cat_input1);
   when ('X') cust_activity = 'High';
   when ('Y') cust_activity = 'Medium';
   when ('Z') cust_activity = 'Low';
   otherwise  cust_activity = 'Unknown';
   end;

  /* Output number rows this thread processed */
   if done then
      put _threadid_=z2. _N_=;
run;
```

```
 60         /* SAS */
 61         data sas.purchase_activity;
 62            length cust_activity $ 8;
 63
 64            set sas.purchase end=done;
 65
 66           /* Transform value */
 67            select (cat_input1);
 68            when ('X') cust_activity = 'High';
 69            when ('Y') cust_activity = 'Medium';
 70            when ('Z') cust_activity = 'Low';
 71            otherwise  cust_activity = 'Unknown';
```

```
72            end;
73
74           /* Output number rows this thread processed */
75            if done then
76               put _threadid_=z2. _N_=;
77         run;

_THREADID_=01 _N_=192735
NOTE: There were 192735 observations read from the data set SAS.PURCHASE.
NOTE: The data set SAS.PURCHASE_ACTIVITY has 192735 observations
      and 20 variables.
NOTE: DATA statement used (Total process time):
      real time           0.25 seconds
      cpu time            0.10 seconds


78
79         /* CAS */
80         /* To run in CAS, SAS librefs changed to CAS librefs. */
81         data cas.purchase_activity;
82            length cust_activity $ 8;
83
84            set cas.purchase end=done;
85
86           /* Transform value */
87            select (cat_input1);
88            when ('X') cust_activity = 'High';
89            when ('Y') cust_activity = 'Medium';
90            when ('Z') cust_activity = 'Low';
91            otherwise  cust_activity = 'Unknown';
92            end;
93
94           /* Output number rows this thread processed */
95            if done then
96               put _threadid_=z2. _N_=;
97         run;

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
_THREADID_=07 _N_=12000
_THREADID_=11 _N_=12000
_THREADID_=09 _N_=12000
_THREADID_=08 _N_=12000
_THREADID_=12 _N_=12000
_THREADID_=03 _N_=12000
_THREADID_=16 _N_=11735
_THREADID_=04 _N_=12000
_THREADID_=05 _N_=12000
_THREADID_=02 _N_=12000
_THREADID_=13 _N_=12000
_THREADID_=01 _N_=13000
_THREADID_=14 _N_=12000
_THREADID_=15 _N_=12000
_THREADID_=06 _N_=12000
_THREADID_=10 _N_=12000
NOTE: There were 192735 observations read from the table PURCHASE
      in caslib CASUSER(<userid>).
```

```
NOTE: The table purchase_activity in caslib CASUSER(<userid>) has
      192735 observations and 20 variables.
NOTE: DATA statement used (Total process time):
      real time           0.08 seconds
      cpu time            0.02 seconds
```

Output 3. Adding a New Variable

To run the DATA step in CAS, the data was loaded into CAS and the only modification made to the DATA step is to change the librefs used to CAS librefs. Because the bulk of the program remains intact, you can reuse your SAS programming experience to write DATA steps that run in parallel in CAS.

When the DATA step runs in CAS, notice how some threads are assigned a few more rows to operate on and some threads a few less. Although CAS does a good job trying to evenly distribute the input rows among threads, some threads might get a few more and some might get a few less than others. With small data, some threads might process no rows at all.

Also, notice how even this small data program sees an improvement in performance because the work was split up among several threads.

In these examples, we see by changing the librefs used in the program to be CAS librefs, the program runs in multiple threads in CAS. Otherwise, the program runs in SAS. We also see how the PUT statement can be used to give debugging feedback on where the program runs and the number of rows a thread processes.

TWO-STEP OPERATIONS

So far, transforming a single-threaded DATA step to run in multiple threads has been a simple transformation. Unfortunately, you cannot take any DATA step, change the librefs used, and have it run correctly in parallel. You still have to know what your program is doing to make sure you know what it does when it runs in parallel.

In this example, we sum a variable across all the rows of a table. The RETAIN statement tells the DATA step to hold a value from one row to the next without resetting it to missing. When running a DATA step in CAS, our first attempt is to only change the librefs to CAS librefs and we discover that while this gets us close to a solution, it isn't a complete solution. We have to be careful with retained variables because each thread retains the values that it sees. When you use a retained variable to hold a sum, the variable will contain the sum of the values that that thread sees.

```
/* Sum a variable across an entire table */

/* SAS */
/* Retain a variable to sum the number of homeowners over all rows */
data sas.sums;
   retain homeowners_sum;
   keep homeowners_sum;

   set sas.purchase end=done;

   if demog_ho = 1 then
      homeowners_sum + 1;

   if done then
      output;
run;

title 'DATA Step in SAS';
proc print data=sas.sums; run;

/* CAS */
```

```
/* Partial sums computed by threads */
data cas.sums;
   retain homeowners_sum;
   keep homeowners_sum;

   set cas.purchase end=done;

   if demog_ho = 1 then
      homeowners_sum + 1;

   if done then
      output;
run;

title 'DATA Step in CAS';
proc print data=cas.sums; run;
```

DATA Step in SAS

Obs	homeowners_sum
1	106054

DATA Step in CAS

Obs	homeowners_sum
1	7500
2	6563
3	7238
4	7391
5	6966
6	7278
7	7125
8	6822
9	6709
10	7276
11	6760
12	6262
13	4774
14	6787
15	5999
16	4604

Output 4. Multiple Threads Output Multiple Rows in CAS

When the DATA step runs in CAS, we see 16 rows of output instead of 1 as we do in SAS. This happens because each thread sums the values that it reads and generates the partial sum that it computed. In this case, 16 threads read data, so we see 16 rows of output. In some sense, this is what we want, to keep all the threads busy computing the sum of the rows they are assigned. To get one final sum, we need something new.

What we need is a way to gather all the partial sum rows and bring them to a single thread. This single thread can sum the partial sums to give a final sum. To do this, a new option is used in the DATA statement, SINGLE=.

When SINGLE=YES is specified, the DATA step runs in a single thread instead of all available threads. When SINGLE=NO is specified, the DATA step runs in all available threads. The default if SINGLE= is not specified is SINGLE=NO, which causes the DATA step runs in all available threads. Here is the program using SINGLE=YES to produce a final sum.

```
/* Use the SINGLE=YES option to run DATA step in a single thread */
data cas.sums_all / single=yes;
   retain homeowners_sum_all;
   keep homeowners_sum_all;

   set cas.sums end=done;

   homeowners_sum_all + homeowners_sum;

   if done then
      output;
run;

title 'Final Sum';
proc print data=cas.sums_all; run;
```

Final Sum

Obs	homeowners_sum_all
1	106054

Output 5. Using SINGLE=YES to Run DATA Step in One Thread

Using SINGLE=YES is convenient when needing a short serial section in a multi-threaded program. If a DATA step with SINGLE=YES processes large tables, not only does the program run slowly by operating in a single thread, the machine executing the single thread can run out of disk or memory resources. Use SINGLE=YES with caution.

While this program uses RETAIN to hold a value from row to row, there are additional features in the DATA step that create dependencies between rows. These are the LAG and DIF functions and temporary arrays. If your program uses these, then a two-step solution might be needed.

This example highlights that each thread has its own variables and computes based on the values that it reads. Because there is no sharing between the threads, we had to break the single DATA step that ran in SAS into two DATA steps that run in CAS. One DATA step runs in parallel on the big data and one runs in a single thread on smaller data. This solution speeds processing by running as much as possible in multiple threads.

BY-GROUP PROCESSING

So far, we have seen row-at-a-time processing: one row in, modify some variables, and one row out. Another useful form of programming is BY-group processing, where rows with the same BY value are grouped together and processed by a single thread.

To process BY groups in a DATA step, 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 statement.

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 DATA step in CAS, 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.

In this example, we sum a variable for a BY group. The result is one sum per BY group. This example is not as complex as summing across an entire table because one thread processes an entire BY group. Therefore, one thread can do the summing within a BY group. There is no need for two DATA steps.

```
/* Sum a variable for each BY group */

/* SAS */
/* Sort table and then we can use BY statement in DATA step */
proc sort data=sas.purchase;
   by cat_input2;
run;

data sas.sum_by_group;
   keep cat_input2 purch_sum;

   set sas.purchase;
   by cat_input2;

   if first.cat_input2 then
      purch_sum = 0;

  /* Sum purchase average for each profitability category */
  purch_sum + purchavgall;

   if last.cat_input2 then
      output;
run;

title 'DATA Step in SAS';
proc print data=sas.sum_by_group; run;

/* CAS */
data cas.sum_by_group;
   keep cat_input2 purch_sum;

   set cas.purchase;
   by cat_input2;

   if first.cat_input2 then
      purch_sum = 0;

   /* Sum purchase average for each profitability category */
   purch_sum + purchavgall;

   if last.cat_input2 then
      output;
run;

title 'DATA Step in CAS';
proc print data=cas.sum_by_group; run;
```

DATA Step in SAS

Obs	cat_input2	purch_sum
1	A	500058.68

Obs	cat_input2	purch_sum
2	B	433806.88
3	C	384411.25
4	D	382550.10
5	E	872475.33

DATA Step in CAS

Obs	cat_input2	purch_sum
1	A	500058.68
2	C	384411.25
3	B	433806.88
4	E	872475.33
5	D	382550.10

Output 6. BY-Group Processing in DATA Step

When the DATA step runs in SAS, one thread performs the BY-group sum, and rows are in the order in which the single thread encounters them in the input.

When the DATA step runs in CAS, the BY statement tells the DATA step to request that the data are grouped on the first BY variable from CAS. CAS groups the data and entire groups are given to a thread. With multiple threads DATA step processes multiple BY groups at the same time.

Performance is affected by the cardinality of the first BY variable. Too few BY values and threads sit idle. Too many BY values and the overhead of grouping becomes a factor.

Grouping rows in CAS does take time. If you are expecting to do a lot of BY processing on a table, you can use the CAS procedure to invoke the Partition action to pre-partition a table before using it. The LOAD statement in the CASUTIL procedure also has options to partition a table when loading the data.

Another popular use of the DATA step is to merge two or more tables with a MERGE statement. Merging BY groups also occurs in parallel. I encourage you to try it when you have access to SAS Viya.

We have seen BY-group processing in a DATA step in CAS. There is no need to sort the data before using a BY statement and multiple BY groups are operated on in parallel by different threads.

VARYING-LENGTH CHARACTER VARIABLES

Character variables in SAS are fixed width. This means if a variable is declared to have a length of 1024, 1024 bytes of memory are used and 1024 bytes of disk space per row are used. If the value does not use all 1024 bytes, the value is blank-padded.

Operations on fixed-width variables can be fast because you are guaranteed to always have a certain number of bytes available. Unfortunately, they can use more space than is needed and are not intuitive to program with for some programmers.

To save memory and have more intuitive string operations, we have added a varying-length character type, or varchar for short, to both SAS and CAS.

Varchar variables are given a maximum width, and space is allocated as needed up to the maximum. Shorter values take less space than longer values and values are not blank padded. Also, the length of a varchar value can go beyond the current 32KB limit for character variables in SAS. In a DATA step, varchar values can be up to 2GB in size.

This might seem like a win-win-win combination. Unfortunately, to manage varying-length values, more instructions are needed when operating with varchar variables. The hope is that additional time is made up by reduced I/O time and memory use.

In this set of examples, we see how varchar variables are more intuitive to work with than character variables. We see how string concatenation is simpler without blank padding. Then, we see how character variables are measured in units of bytes, whereas varchar variables are measured in units of characters. Using units of characters is more intuitive with non-English characters or symbols that take multiple bytes.

```
/* Use varying-length character variables (varchar) in a DATA step */
/* More intuitive operations -- no longer need to remove trailing blanks */
/* Before varchar, had to trim trailing blanks to get needed result */

data _null_;
   length first last $ 32 fullname_notrim fullname $ 64;

   first = "Jane";
   last = "Jones";
   fullname_notrim = first || ' ' || last;
   fullname = trim(first) || ' ' || last;

   put fullname_notrim=;
   put fullname=;
run;

/* No trailing blanks with varchar, more intuitive operations */
data _null_;
   length first last fullname varchar(*);

   first = "Jane";
   last = "Jones";
   fullname = first || ' ' || last;

   put fullname=;
run;

/* Find length of a string */
/* Character values are measured in units of bytes */
/* Varchar are measured in units of characters */
data _null_;
   length c $ 64;
   length vc varchar(*);

   c  = "谢谢为使用SAS!";
   vc = "谢谢为使用SAS!";

   c_len = length(c);
   vc_len = length(vc);

   put c_len= vc_len=;
run;
```

```
 61         data _null_;
 62            length first last $ 32 fullname_notrim fullname $ 64;
 63
 64            first = "Jane";
 65            last = "Jones";
 66            fullname_notrim = first || ' ' || last;
 67            fullname = trim(first) || ' ' || last;
```

```
68
69            put fullname_notrim=;
70            put fullname=;
71         run;

fullname_notrim=Jane                                Jones
fullname=Jane Jones
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


72
73         /* No trailing blanks with varchar, more intuitive operations */
74         data _null_;
75            length first last fullname varchar(*);
76
77            first = "Jane";
78            last = "Jones";
79            fullname = first || ' ' || last;
80
81            put fullname=;
82         run;

fullname=Jane Jones
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


83
84         /* Find length of a string */
85         /* Character values are measured in units of bytes */
86         /* Varchar are measured in units of characters */
87         data _null_;
88            length c $ 64;
89            length vc varchar(*);
90
91            c  = "谢谢为使用SAS!";
92            vc = "谢谢为使用SAS!";
93
94            c_len = length(c);
95            vc_len = length(vc);
96
97            put c_len= vc_len=;
98         run;

c_len=19 vc_len=9
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

Output 7. Varchar Use in DATA Step

In the first program, the blank padding of fixed-width character values is not so intuitive. To remove the blank padding, we have to use the TRIM function or call one of the CAT functions to perform the concatenation.

In the second program, varchar variables are used. There is no blank padding, and the concatenation operator produces the value we might expect without having to use additional functions.

In the third program, five Chinese characters are used. Each Chinese character uses three bytes. Although the character variable holds four Chinese characters and five Latin characters, the LENGTH function reports 19 bytes are used. When using a varchar variable to hold the same value, the LENGTH function reports nine characters are used, even though the value uses 19 bytes.

Other functions, like the SUBSTR and INDEX functions, also use numbers in units of characters instead of bytes when passed varchar variables. Using a unit of characters is more intuitive when processing multiple types of languages.

CONCLUSION

We have explored running the DATA step in CAS. Understanding when the DATA step runs in SAS and when it runs in CAS is important in knowing how the DATA step operates. While changing SAS librefs to CAS librefs can get many programs to run faster in multiple threads in CAS, other programs require more thought and might need to be broken into two steps instead of one. In exchange for this effort, running DATA step in multiple threads can greatly reduce the time to process large tables.

ACKNOWLEDGMENTS

Developing software is a team effort. I would like to thank those involved in the development, testing, documentation, and support of DATA step in CAS: David Bultman, Melissa Corn, Hua Rui Dong, Jerry Pendergrass, Denise Poll, Mike Jones, Lisa Davenport, Jane Eslinger, Kevin Russell, Al Kulik, Rick Langston, Kanthi Yedavalli, Joe Slater, Robert Ray, Mark Gass, and Oliver Schabenberger. Thanks to David Bultman, Lisa Davenport, Jane Eslinger, Kevin Russell, and Julia Schelly for reviewing this paper.

RESOURCES

- SAS Institute Inc. 2017. *SAS® Cloud Analytic Services 3.1: Fundamentals.* Available at http://go.documentation.sas.com/?docsetId=casfun&docsetVersion=3.1&docsetTarget=titlepage.htm.
- SAS Institute Inc. 2017. *SAS® Visual Data Mining and Machine Learning 8.1, Accessing and Manipulating Data, Data Step Programming*. Available at http://go.documentation.sas.com/?cdcId=vdmmlcdc&cdcVersion=8.1&docsetId=casdataam&docsetTarget=p0qlpqt6uhyqhen17qyj6mc9shtp.htm.
- *Accelerating SAS DATA Step Performance with SAS Viya*. Available at http://support.sas.com/training/tutorial/viya/viyavsp01.html.

CONTACT INFORMATION

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

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