Paper SAS3220-2019

Working with Big Data in SAS®

Mark L. Jordan, SAS Institute Inc.

ABSTRACT

This paper demonstrates the challenges you might face and their solutions when you use SAS® to process large data sets. First, we demonstrate how to use SAS system options to program query efficiency. Next, we tune SAS programs to improve big data performance, modify SQL queries to maximize implicit pass-through, and re-architect processes to improve performance. Along the way, we identify situations in which data precision might be degraded, and then leverage FedSQL and DS2 to conduct full-precision calculations on a wide variety of ANSI data types. Finally, we look at boosting speed by using parallel processing techniques. We begin by using DS2 in Base SAS® to speed up CPU bound processes, and then kick it up a notch with the SAS® In-Database Code Accelerator for in-database processing on massively parallel processing (MPP) platforms like Teradata and Apache Hadoop. And for the ultimate speed boost, we convert our programs to use distributed processing and process memory-resident data in SAS® Viya® and SAS® Cloud Analytic Services (CAS).

INTRODUCTION

SAS programmers often have to manipulate data from a wide variety of data sources. SAS provides tools for accessing that data, but the burgeoning size of today's data sets makes it imperative that we understand how SAS works with external data sources and how to detect processing bottlenecks. In this way, we can tune our SAS processes for better performance. While it is essential to conduct formal benchmarking to tune production programs, this process is tedious and time consuming. We'll develop a few programming rules of thumb that should help us build better ad hoc programs in day-to-day programming tasks and that demonstrate how to detect common problems if our code runs for longer than you expect.

This paper provides a brief overview of the following items:

- data flow using SAS/ACCESS Interface technology in traditional SAS and SAS Viya architectures
- data manipulation techniques using DATA step, SQL, and DS2

This paper also provides a discussion of potential bottlenecks during processing, including these details:

- where bottlenecks might occur
- using SAS system options and the SAS log to detect processing bottlenecks
- SAS programming rules of thumb to avoid or minimize the effects of bottlenecks

DATA FLOW IN SAS PROGRAMS

DATA ACCESS IN SAS PROGRAMS

When we attempt to tune our code for more efficient processing, we try to optimize the system's I/O processes, memory utilization, and CPU utilization. Because I/O is generally the slowest operation in data processing, it's generally the first bottleneck we tackle. Let's take a conceptual look at how data is retrieved for processing by SAS and how results are stored back to disk.

All data in SAS is accessed via a SAS engine. Native SAS data sets are accessed via the default (V9) engine, which is shipped with all SAS installations. The V9 engine coordinates with the operating system (OS) to read and write data to disk, and all data manipulation occurs in SAS. Figure 1 illustrates the conceptual data flow from disk storage to memory, processing, and storing the processed data back to disk:

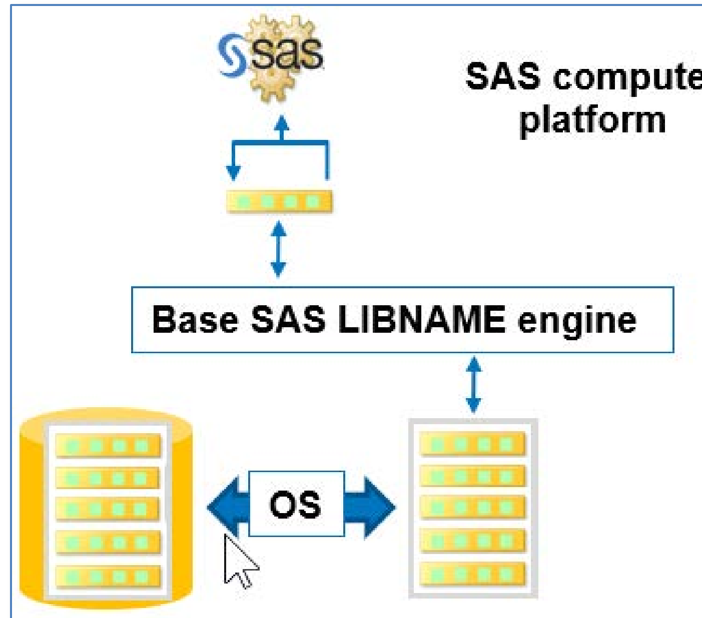

Figure 1 - Data Flow for Base SAS DATA Step Processing

The OS reads the data into memory in blocks from the disk drive and then passes data to SAS for processing. SAS processes data and writes results to memory one observation at a time, passing the result back to the OS for writing to disk. If the data is very large, processing delays are often primarily attributable to the time it takes to read and write the data to disk. Generally, this results in a process that is Input/Output (I/O) bound – that is, each row can be processed more quickly than the next row can be delivered. However, in today's world of sophisticated algorithms the processing for each row might be so complex that the computations cannot be completed on a single row before another row could be made available. This situation results in a process that is CPU bound. Base SAS 9.4 includes the DS2 programming language, which can easily conduct threaded processing of your data, speeding up these CPU bound processes, as illustrated in Figure 2.

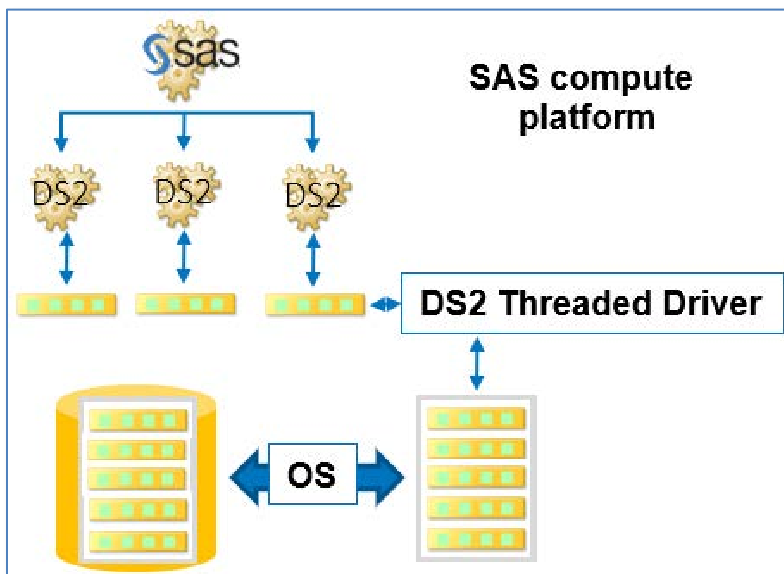

Figure 2 – Parallel Processing with DS2 on the SAS Platform

In addition to the V9 engine, SAS/ACCESS engines can be licensed, providing transparent access to data from a wide variety of database management systems (DBMS). SAS/ACCESS engines provide access to data via the SAS LIBNAME statement, and DBMS data accessed this way can be consumed directly by any SAS processes. The programmer writes traditional SAS programs, relying on the LIBNAME engine to generate native code (usually database-specific SQL) to retrieve data from the DBMS. Figure 3 illustrates this process

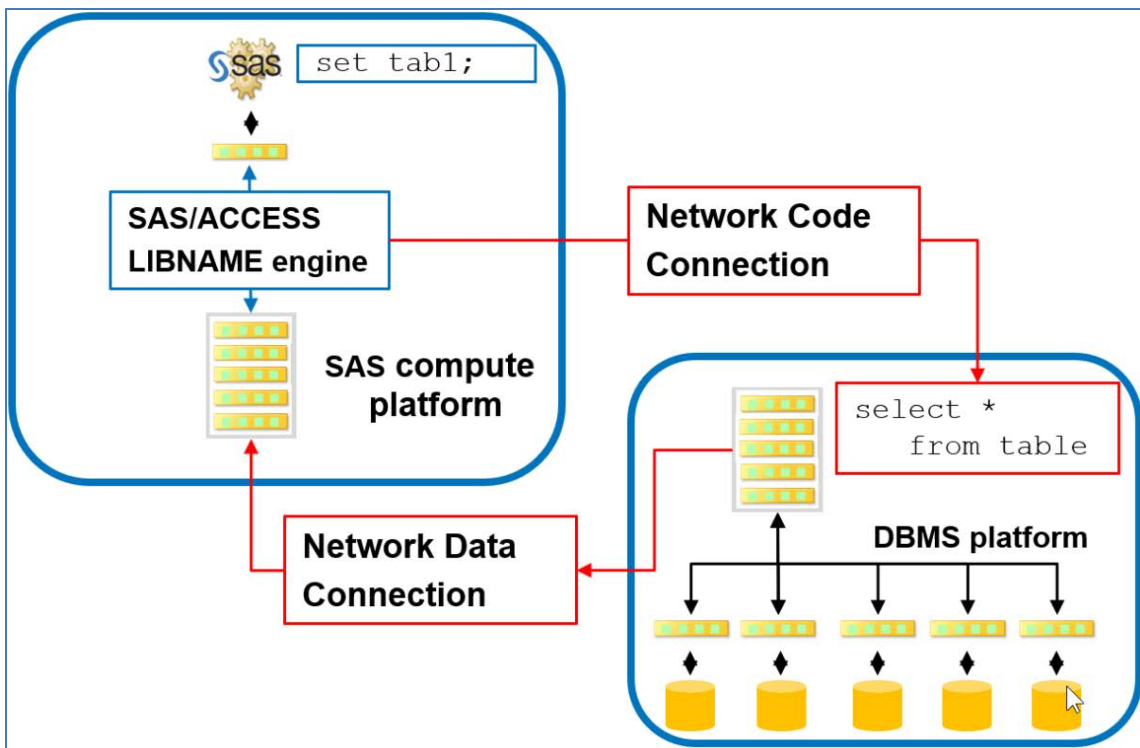

Figure 3 - Processing DBMS Data in SAS

Accessing DBMS data adds network lag time to our process, but because most DBMS systems provide parallel processing and ideally are co-located and connected to the SAS system by very fast networks, the actual time required to read and write data can be significantly reduced. Additional speed in throughput could easily be achieved if we reduced the size of the data before moving it to the SAS platform.

PERFORMANCE CONSIDERATIONS WHEN SUBSETTING DATA

Many DATA steps processes require only a subset of the input data. Row subsetting can be accomplished using either a WHERE statement or an IF statement. When processing large data sets, it is important to be aware of the difference between the two. WHERE subsetting occurs before the O/S or data source passes the data to SAS for processing. IF subsetting occurs during SAS processing. When reading native SAS data sets from disk, WHERE processing does not reduce the number of data blocks read from the disk, but it *does* reduce the amount of data passed to SAS, minimizing data movement in memory as well as reducing the amount of processing required, improving process throughput. Here is Rule of Thumb 1 for writing efficient DATA step code:

1. When WHERE and IF statements produce the same result, use WHERE for efficiency's sake.

Figure 4 illustrates the difference in processing by showing that fewer rows are presented to SAS for processing when the WHERE statement is used:

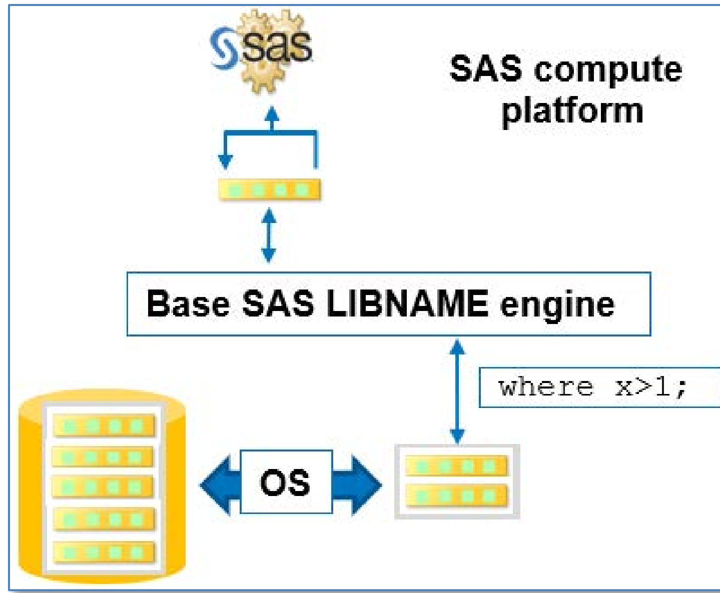

Figure 4 - Subsetting Rows in the DATA Step

Data sets used in SAS processing often have a large number of columns. For example, when we start building predictive models, the best input data sets have thousands of variables, which can influence the outcome. When the model is finished, usually a small number of influential variables have been identified and only those are required when scoring the data. In the SAS DATA step, there are two methods of reducing the number of columns in the output data set: the KEEP or DROP statement, or the KEEP= or DROP= data set option. The KEEP and DROP statements prevent unwanted columns from being written out to the result set, but using the KEEP= or DROP= data set option on the data sets being loaded prevents the unwanted columns from being passed to SAS by the data source. Once again, while KEEP= and DROP= processing does not reduce the number of data blocks read from the disk, it *does* reduce the amount of data passed to SAS, minimizing data movement in memory and reduces the amount of processing required, improving process throughput. It also reduces the amount of memory required for the Program Data Vector (PDV). Therefore, here is Rule of Thumb 2 for writing efficient DATA step code:

2. If KEEPor DROP statements and KEEP= or DROP= data set options produce the same result, use the KEEP= or DROP= data set options for efficiency's sake.

Figure 5 compares and contrasts processing with the KEEP statement versus the KEEP= data set option:

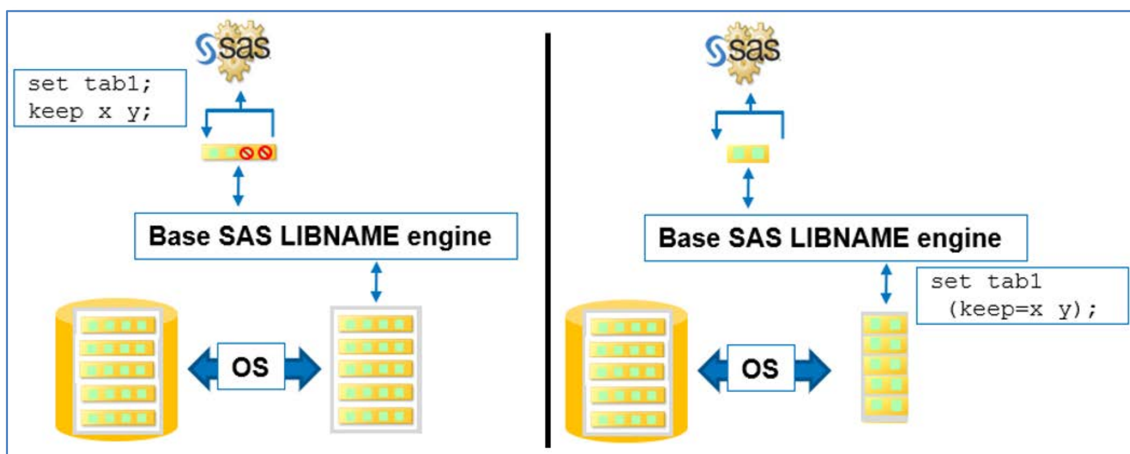

Figure 5 - KEEP Statement versus KEEP= Data Set Option Processing

The SAS/ACCESS Interface can generate quite sophisticated native DBMS code when retrieving data from the DBMS for processing in SAS. The engines are designed to pass-through as much of the work as possible to the DBMS before returning the results into SAS for final processing. Subsetting, summarization, and sorting are all best performed in the DBMS, which is generally well provisioned and optimized for this type of processing. Subsetting columns and rows, or pre-summarizing the data on the DBMS side also means less data movement over the network to the SAS platform. As network latency is usually a significant bottleneck, subsetting or summarizing on the DBMS should significantly improve throughput. Figure 6 illustrates how WHERE subsetting of rows and KEEP= or DROP= subsetting of columns is converted to native SQL by the SAS/ACCESS engine for in-database processing, minimizing data movement between the DBMS and the SAS platform:

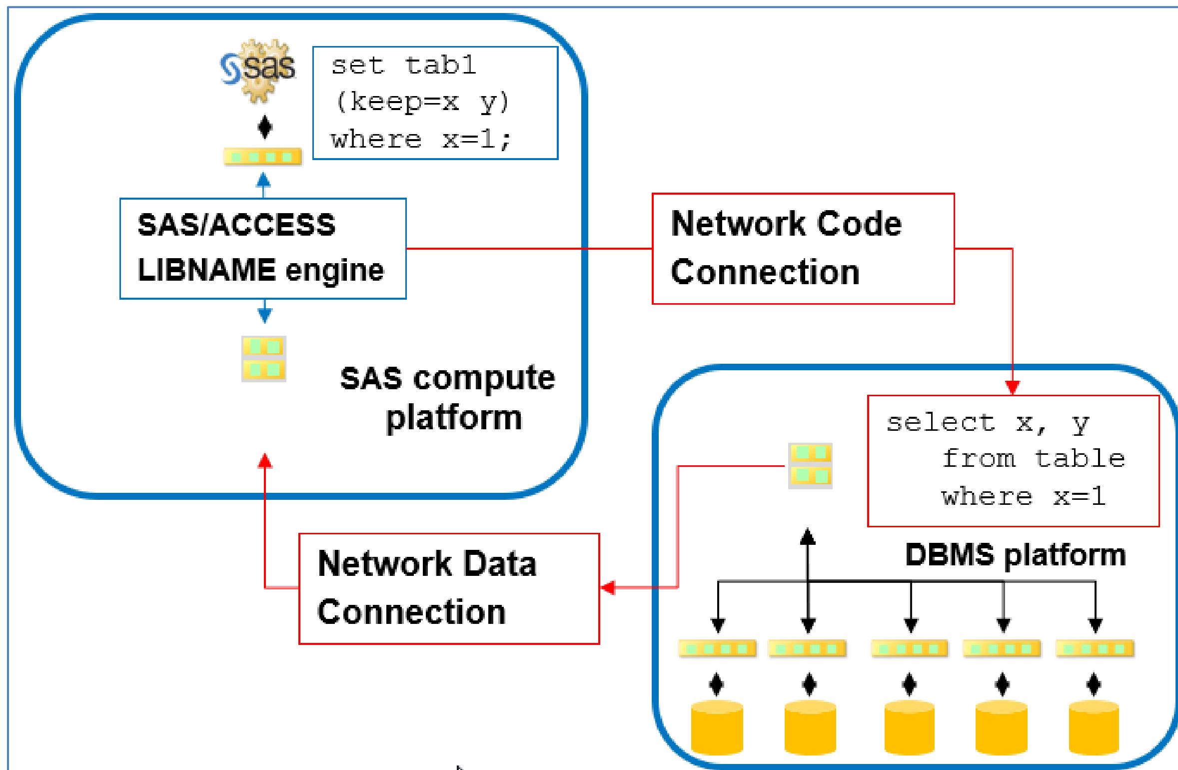

Figure 6 - In-database Subsetting by the SAS/ACCESS LIBNAME Engine

The dilemma here is that not all SAS syntax can be converted into native SQL by the LIBNAME engine. Sometimes the SQL is just too complex for a code generator to create, but many times the SAS DATA step has syntax and control features that are not available in ANSI SQL. SAS ensures that all processing is completed, so whatever processing can't be accomplished on the DBMS platform is completed on the SAS platform. And that means that eventually some data has to be moved between the platforms, exposing us to network latency.

Because SAS usually has several methods for accomplishing a task, if the end results are the same, we'll want to choose methods most likely to process on the DBMS platform. We will use two different programs that produce the same result using different techniques to demonstrate the issue. In order to see the SQL communicated to the DBMS by the SAS LIBNAME engine, let's activate the SASTRACE, SASTRACELOC, and NOSTSUFFIX system options, which cause the LIBNAME engine to echo all of the SQL it generates in the SAS log. Here is the program statement:

```
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
```

The first program we test uses the SCAN function:

```
proc sql;
select customer_id
       ,customer_name
   from db.big_customer_dim
   where scan(customer_name,1) ='Silvestro';
quit;
```

Figure 7 shows that this produces the desired result:

CUSTOMER_ID	CUSTOMER_NAME
82229	Silvestro Baschera

Figure 7 – Results of SQL Query using the SCAN Function

However, the program ran a long time. The SAS log shows how long and why:

```
TERADATA_2: Executed: on connection 2
SELECT "CUSTOMER_ID","CUSTOMER_NAME" FROM "db"."big_customer_dim"
TERADATA: trget - rows to fetch: 89954
NOTE: PROCEDURE SQL used (Total process time):
      real time           6.53 seconds
      cpu time            0.89 seconds
```

The second program we test uses the ANSI LIKE operator instead of the SCAN function:

```
proc sql;
select customer_id
       ,customer_name
   from db.big_customer_dim
   where customer_name like'Silvestro%';
quit;
```

This also produces same result shown in Figure 7, but the program runs much faster. Once again, the SAS log reveals the reason:

```
TERADATA_2: Executed: on connection 2
SELECT "CUSTOMER_ID","CUSTOMER_NAME" FROM "db"."big_customer_dim"
WHERE  ("CUSTOMER_NAME" LIKE  'Silvestro%' )
TERADATA: trget - rows to fetch: 1
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.65 seconds
      cpu time            0.89 seconds
```

While both programs only extracted two columns from the DBMS table, the first program brought 82,229 rows of data out of the DBMS into SAS for subsetting with the SCAN function, because this function couldn't be converted into DBMS SQL. When we used the ANSI standard LIKE operator, the LIBNAME engine was able to generate SQL code to do the subsetting in-database, and only one row of data had to be brought back to SAS, so this query took about one tenth of the time to process – a significant acceleration! Therefore, here is Rule of Thumb 3 for writing efficient SAS step code:

1. When writing WHERE criteria, use ANSI standard expressions as often as possible.

For really tough performance-critical issues, you might know of arcane DBMS features which the LIBNAME engine is not capable of leveraging. In this case, you might consider writing the DBMS SQL yourself and submitting explicit pass-through code. You will find documentation on how to accomplish this in the SAS/ACCESS Interface documentation for your DBMS or in the *SAS SQL Procedure User's Guide*.

PERFORMANCE CONSIDERATIONS WHEN SUMMARIZING DATA

We've spoken quite a bit about the benefits of conducting as much processing as possible in the DBMS. The best of all worlds would let us leverage a massively parallel processing (MPP) platform for both Read, Write, and computation operations. For many DBMS installations with MPP characteristics, the SAS LIBNAME engine, in cooperation with in-database enabled Base SAS procedures, can generate quite complex SQL to complete a significant amount of the processing in the DBMS. This can significantly speed up summarization processing for very large data sets. Procedures capable of this type of in-database processing include PROC FREQ, PROC RANK, PROC REPORT, PROC SORT, PROC SUMMARY, PROC MEANS, and PROC TABULATE. For more information, consult the *SAS® 9.4 In-Database Products: User's Guide*.

Here is an example of summarizing with the DATA step:

```
data summary_data;
   set db.big_order_fact (keep=customer_id Total_retail_price);
   by customer_id;
   if first.customer_id then do;
      Items=0;
      Value=0;
   end;
   Count+1;
   Value+Total_Retail_Price;
   if last.customer_id then output;
   keep customer_ID Count Value;
run;
```

A peek at the log shows that the process worked, but took an exceptionally long time to process, because SAS had to retrieve all 2,527,232 rows from the DB and conduct the processing on the SAS platform:

```
NOTE: There were 2527232 observations read from the data set
db.big_order_fact.
NOTE: The data set WORK.SUMMARY_DATA has 75 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           1:01.69
      cpu time            8.10 seconds
```

Using an in-database capable procedure like PROC MEANS could improve the processing time:

```
proc means data=db.big_order_fact noprint;
   class customer_id;
   output out=summary_means
   (drop=_: where=(Customer_ID is not missing))
   n=Count sum=Value;
   var Total_retail_Price;
run;
```

The log indicated pretty dramatic improvement in processing speed:

```
TERADATA: Executed: on connection 4
select COUNT(*) as "ZSQL1", MIN(TXT_1."CUSTOMER_ID") as "ZSQL2"
   ,COUNT(*) as "ZSQL3", COUNT(TXT_1."TOTAL_RETAIL_PRICE") as "ZSQL4"
   ,SUM(CAST( TXT_1."TOTAL_RETAIL_PRICE" AS DOUBLE PRECISION)) as "ZSQL5"
   from "db"."big_order_fact" TXT_1
   group by TXT_1."CUSTOMER_ID"
TERADATA: trget - rows to fetch: 75
NOTE: PROCEDURE MEANS used (Total process time):
      real time           3.30 seconds
      cpu time            0.46 seconds
```

We could also write this using only standard SQL code:

```
proc sql;
create table summary as
select customer_id
      ,count(*) as Count
      ,sum(Total_retail_price) as Value
   from db.big_order_fact
   group by customer_id;
quit;
```

The log shows that this technique provides even better performance:

```
TERADATA_60: Executed: on connection 3
select TXT_1."CUSTOMER_ID", COUNT(*) as "Count"
       , SUM(CAST(TXT_1."TOTAL_RETAIL_PRICE" AS DOUBLE PRECISION)) as
"Value"
   from "db"."big_order_fact" TXT_1
   group by TXT_1."CUSTOMER_ID"
TERADATA: trget - rows to fetch: 75
NOTE: PROCEDURE SQL used (Total process time):
      real time           1.28 seconds
      cpu time            0.14 seconds
```

So, we can establish Rule of Thumb 4 for writing efficient SAS code:

1. If possible, write the process using SQL.
   a. If writing the process in SQL is not possible, use an in-database capable procedure.
   b. If an in-database procedure can't do the job, a DATA step will get the job done, but it might take extra time to execute.

Wouldn't it be nice if we could execute DATA step processing in the database?

DS2 AND THE SAS® IN-DATABASE CODE ACCELERATOR

DS2 DATA PROGRAMS

The DS2 language was briefly mentioned in the **DATA ACCESS IN SAS PROGRAMS** section of this paper, where Figure 2 illustrated parallel processing on the SAS platform. DS2 melds the process control we love in the DATA step with the coding ease of SQL to provide an intriguing and useful hybrid language with much syntax in common with the traditional SAS DATA step. FedSQL is the SQL language which pairs with DS2 and shares many of its advantages.

If the log from your DATA step program shows elapsed time is very close to CPU time, the process is likely CPU bound. DS2 provides simple and safe syntax for threaded processing right on the SAS

compute platform. In cases where the original process is CPU bound, the ability to process multiple rows of data in parallel can speed up data throughput. So, here is Rule of Thumb 5:

2. If your DATA step process is CPU bound, consider re-writing it in DS2 to take advantage of threaded processing on the SAS compute platform.

Besides threading, DS2 offers two other distinct benefits:

1. Direct access of DBMS data using a threaded driver, bypassing the LIBNAME engine. The LIBNAME engine translates all ANSI data types to either SAS fixed-width character (CHAR) or numeric (DOUBLE) before exposing the data to SAS processing. While this allows traditional SAS processes to operate as if the data were a native SAS data set, this can lead to a loss of precision, especially if the source data is ANSI BIGINT or DECIMAL type. Because DS2 accesses the data directly, full precision is preserved while reading, manipulating and writing the data back to the database. This characteristic is shared with FedSQL.
2. For Teradata, Hadoop, and Greenplum databases, SAS has created a SAS In-Database Code Accelerator product that--when licensed, installed and properly configured--allows DS2 programs to run on the DBMS platform. This method brings the DS2 code to where the data resides instead of having to bring the data to the SAS compute platform. This can provide astonishing boosts in performance.

PRECISION ISSUES

SAS uses double-precision, floating point data for all numeric values, which can be problematic in today's world. For example, in 1976, the year SAS Institute was incorporated, the total US national debt was approximately $620,387,900,876.52. That's 14 significant digits, and well within the precision of 16 significant digits available with SAS numerics on ANSI platforms. On January 2, 2018, the US national debt stood at $20,492,598,761,860.61 – that would require 17 significant digits to express precisely, which is more than can be accommodated by a double-precision floating point numeric on an ANSI system. For many years, DBMSs have supported fixed-decimal data types, which can accommodate much higher precision. For example, in Teradata the DECIMAL type can have up to 38 digits of precision. These numerics serve better for accounting processes with numbers of this size. Processing these numbers using traditional Base SAS processes via LIBNAME access can result in inadvertent loss of precision in the results.

For example, this program projects the US debt from 2006 to 2018 using a traditional DATA step:

```
data db.debt_data_step;
   year=                2006;
   debt=    8506973899215.23;
   increase= 998802071887.12;
   output;
   do year=2007 to 2018;
     debt=debt+increase;
     output;
   end;
run;
```

Here are the results for the year 2018:

```
20,492,598,761,860.70
```

The same process in DS2 produces more accurate results:

```
proc ds2;
data db.debt_ds2/overwrite=yes;
   dcl int Year;
   dcl decimal (38,2) debt increase;
   method init();
```

```
        year=                 2006;
        debt=     8506973899215.23n;
        increase= 998802071887.12n;
        output;
        do year=2007 to 2018;
          debt=debt+increase;
          output;
        end;
     end;
  enddata;
  run;
  quit;
```

Here are the results from the DS2 process for the year 2018:

```
20,492,598,761,860.67
```

This example is operating right at the edge of the capabilities of the DOUBLE data type, so the difference in the answers is only a few cents. However, in today's financial environment, precision is critical. DS2 and FedSQL enable processing DECIMAL data without loss of precision.

IN-DATABASE PROCESSING

Taking the code to the data with the SAS® In-Database Code Accelerator for MPP platforms like Teradata and Hadoop reduces data movement and enables parallel processing. With DS2 compute logic installed on all the nodes of the MPP system, calculations can often be completed right on the node where the data was originally stored. And if the result set is written back to the same DBMS, code goes in from the SAS platform, and nothing comes out but the log! Figure 8 illustrates how DS2 processes in-database with the SAS® In-Database Code Accelerator installed on the MPP DBMS platform:

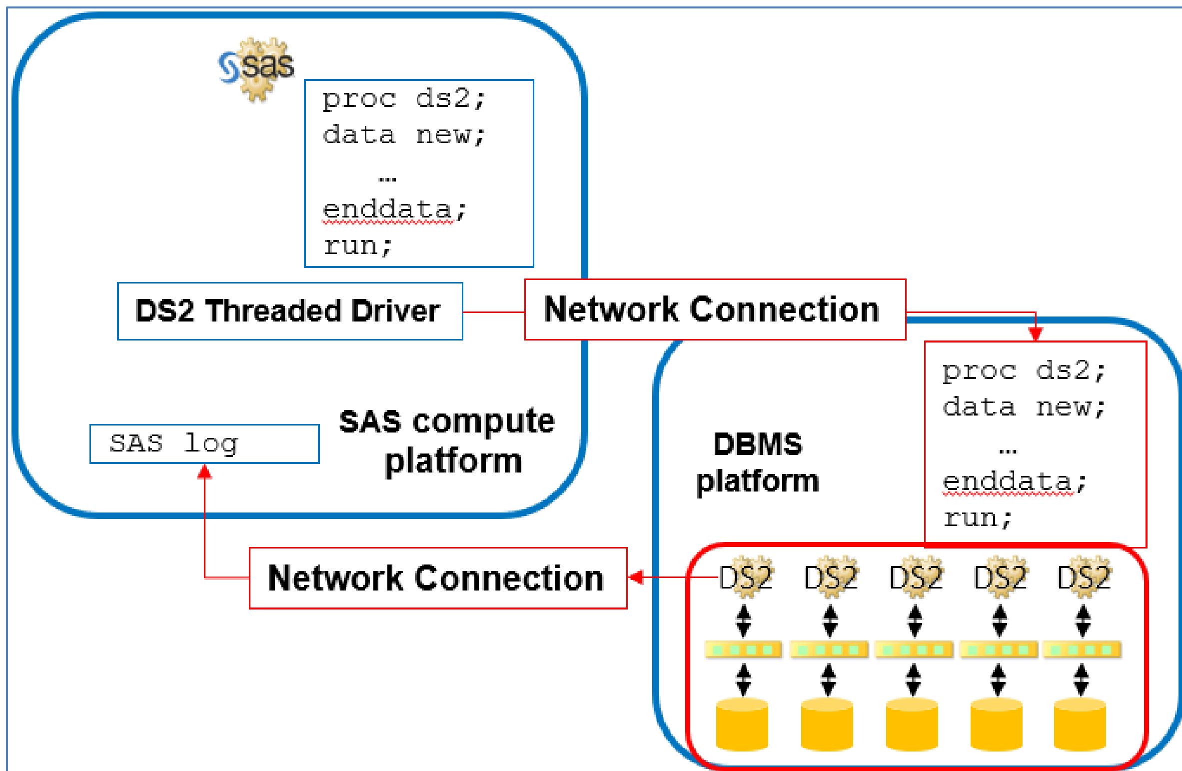

Figure 8 - In-Database Processing with DS2

With DS2 processing in-database, you get the SAS functions and process control of SAS DATA step-style programming, parallel processing distributed across the MPP nodes, all while manipulating data at full native precision. Therefore, here is how we define Rule of Thumb 6:

3. If your data source resides on a Greenplum, Hadoop, or Teradata platform with the SAS® In-Database Code Accelerator installed, writing your process in DS2 is likely to significantly improve efficiency and throughput.

The DS2 in-database processing retrieves the data from DBMS storage in parallel, processes the data in parallel, and writes the data back to storage in parallel. Though parallel Read-Write operations are much faster than traditional serial Read-Write operations, the slowest part of most computer processing is still the disk I/O. Is there any way to parallel process data in SAS while the data persists in memory for multiple uses? And what if my data is not entirely sourced from a single DBMS?

SAS® VIYA® AND CLOUD ANALYTIC SERVICES (CAS)

The new SAS Viya architecture provides many of the benefits of in-database processing while providing speedy access to multiple data sources. A CAS library provides the capability for fast-loading data into memory, conducting distributed processing across multiple nodes, and retaining the data in memory for access by other processes until deliberately saved to disk. CAS has reflective ReST APIs allowing access from just about any client application. CAS actions can be executed on demand from several languages, including Java, Lua, Python, and R. And, of course, CAS processes can be easily accessed using the SAS client in SAS Viya or using SAS 9.4 as the client application.

CAS provides a flexible, fast, fault-tolerant, in-memory, massively parallel processing environment. Figure 9 provides an overview of a Cloud Analytic Services MPP server and its relation to clients and multiple data stores:

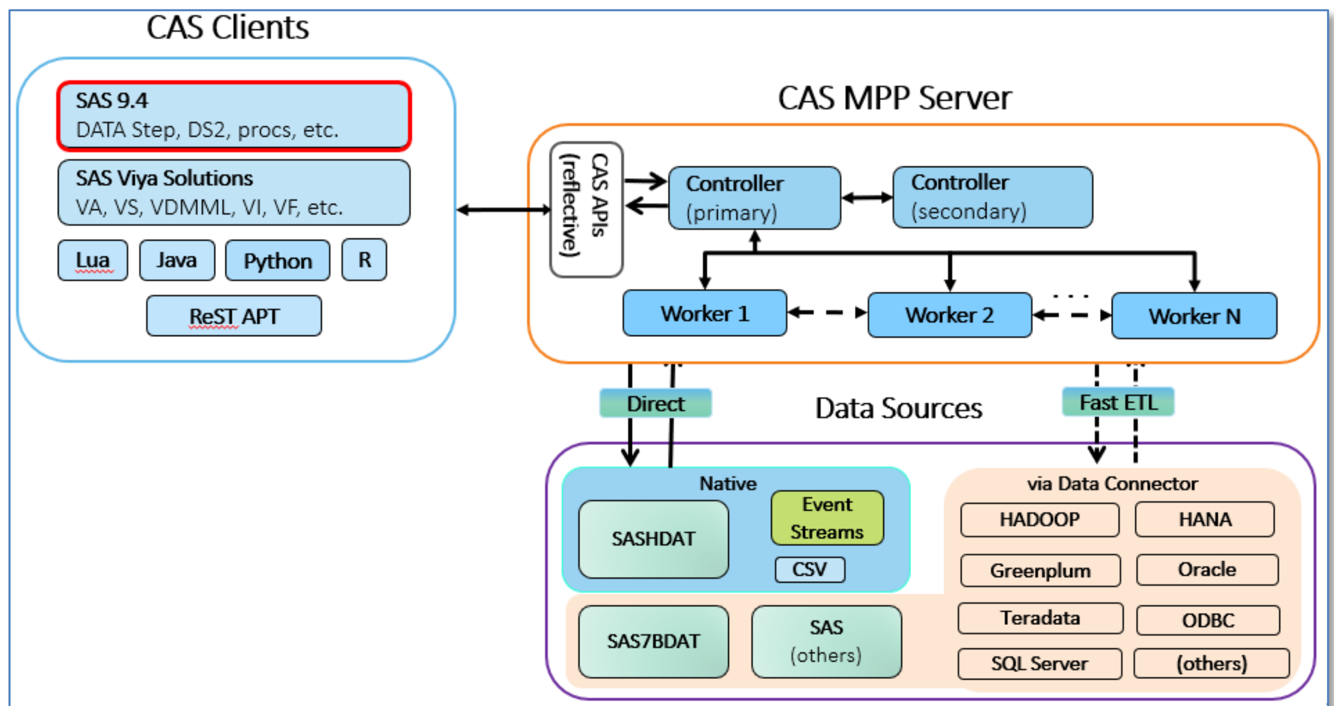

Figure 9 - Cloud Analytic Services Server

CAS can execute traditional DATA step and DS2 programs fully distributed on memory resident data, dramatically improving speed, and throughput.

Consider our original DATA step program used to summarize Big_order_fact. When this program ran on the Base SAS platform, it ran for a very long time. Let's load that data up into CAS and run the same DATA step on it:

```
proc casutil;
   load data=db.big_order_fact
   outcaslib="casuser"
   casout="big_order_fact" promote;
run;

data casuser.summary_data;
   set casuser.big_order_fact (keep=customer_id Total_retail_price);
   by customer_id;
   if first.customer_id then do;
      Items=0;
      Value=0;
   end;
   Count+1;
   Value+Total_Retail_Price;
   if last.customer_id then output;
   keep customer_ID Count Value;
run;
```

Here are the pertinent notes from the SAS log:

```
NOTE: PROCEDURE CASUTIL used (Total process time):
       real time           13.01 seconds
       cpu time            4.89 seconds

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
NOTE: There were 2527232 observations read from the table BIG_ORDER_FACT in
caslib CASUSER.
NOTE: The table summary_data in caslib CASUSER has 75 observations and 3
variables.
NOTE: DATA statement used (Total process time):
       real time           2.35 seconds
       cpu time            0.01 seconds
```

Wow! Traditional SAS took over a minute to execute the original program, while CAS took 2.35 seconds to run the same program without tuning or other code modifications. So, here is our final Rule of Thumb:

4. If you have a SAS 9.4M5 or higher installation with access to SAS® Viya® Cloud Analytic Services, consider running your processes in CAS for an excellent speed boost.

CONCLUSION

SAS provides a diverse set of tools and architectures for accessing and processing data from a wide variety of data sources. The SASTRACE= system option can help you identify processing bottlenecks when accessing DBMS data via the SAS/ACCESS LIBNAME engine. While formal benchmarking is the rule for tuning production programs, which are executed often over long periods of time, benchmarking is time consuming, resource intensive, and can be tedious. For every-day, ad hoc programs, complying with some simple rules of thumb can help avoid many of the issues affecting processing speed when dealing with large data sets.

RULES OF THUMB FOR EFFICIENT SAS CODE

1. When WHERE and IF statements produce the same result, use the WHERE clause for efficiency's sake.
2. If KEEP / DROP statements and KEEP=/DROP= data set options produce the same result, use the KEEP=/DROP= data set options.
3. When writing WHERE criteria, keep the expression as ANSI standard as possible.
4. If possible, write the process using SQL.
    a. If writing a process in SQL is not possible, use an in-database capable procedure.
    b. If an in-database procedure can't do the job, a DATA step will get the job done, but it might take extra time to execute.
5. If your DATA step process is CPU intensive, consider re-writing it in DS2 to take advantage of threaded processing on the SAS Compute platform.
6. If your data source resides on a Greenplum, Hadoop, or Teradata platform with the SAS® In-Database Code Accelerator installed, writing your process in DS2 is likely to significantly improve efficiency and throughput.
7. If you have a SAS 9.4M5 or higher installation with access to SAS® Viya® Cloud Analytic Services, consider running your processes in CAS for an excellent speed boost.

REFERENCES

(SAS Institute, Inc, 2017) *SAS/ACCESS® 9.4 for Relational Databases: Reference, Ninth Edition*, Cary, NC.

(SAS Institute, Inc, 2017) *SAS® 9.4 In-Database Products: User's Guide, Seventh Edition*, Cary, NC.

(SAS Institute, Inc., 2017) *SAS® 9.4 DS2 Programmer's Guide*, Cary, NC.

(SAS Institute, Inc., 2017) *SAS® 9.4 FedSQL Language Reference, Fifth Edition*, Cary, NC.

(SAS Institute, Inc, 2017) *An Introduction to SAS® Viya® 3.3 Programming*, Cary, NC.

(SAS Institute, Inc., 2017) *SAS® SQL Methods and More Course Notes*, Cary, NC.

ACKNOWLEDGMENTS

Special thanks to Brian Bowman of SAS Institute's Research and Development group for his excellent explanation of CAS and the original overview illustration of the CAS MPP server.

RECOMMENDED READING

- Messineo, Martha.2017. *Practical and Efficient SAS Programming: The Insider's Guide.*Cary, NC: SAS Institute Inc.
- Jordan, Mark. 2018. *Mastering the SAS® DS2 Procedure: Advanced Data Wrangling Techniques, Second Edition* Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

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

Mark Jordan
SAS Institute, Inc.
Email: mark.jordan@sas.com
Blog: http://sasjedi.tips
Twitter: @SASJedi
Facebook: http://facebook.com/sasjedi

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.