SESUG Paper 136-2023

SAS ® Program Efficiency for Beginners

Bruce Gilsen, Federal Reserve Board

ABSTRACT

This paper presents simple efficiency techniques that can benefit SAS ® software users on all platforms. While appropriate for beginners, users at all levels of experience can benefit from these techniques.

Efficiency techniques are presented for the following programming tasks.

1. Create a SAS data set by reading long records from a flat file with an INPUT statement. Keep selected records based on the values of only a few incoming variables.
2. Create a new SAS data set by reading an existing SAS data set with a SET statement. Keep selected observations based on the values of only a few incoming variables.
3. Select only some observations from a SAS data set. The selected data are used as input to a SAS procedure, but are not otherwise needed.
4. In IF, WHERE, DO WHILE, or DO UNTIL statements, use OR operators or an IN operator to test if at least one of a group of conditions is true. In IF, WHERE, DO WHILE, or DO UNTIL statements, use AND operators to test if all of a group of conditions are true.
5. Select observations from a SAS data set with a WHERE statement.
6. In a DATA step, read a SAS data set with many variables to create a new SAS data set. Only a few of the variables are needed in the DATA step or the new SAS data set.
7. Create a new SAS data set containing all observations from two existing SAS data sets. The variables in the two data sets have the same length and type.
8. Process a SAS data set in a DATA step when no output SAS data set is needed. This could occur when a DATA step is used to write reports with PUT statements, examine a data set's attributes, or generate macro variables with the CALL SYMPUT statement.
9. Execute a SAS DATA step in which the denominator of a division operation could be zero.

INTRODUCTION

This paper presents simple efficiency techniques that can benefit SAS ® software users on all platforms. While appropriate for beginners, users at all levels of experience can benefit from these techniques.

Efficiency techniques are frequently documented as follows:

- Describe an efficiency technique.
- Demonstrate the technique with examples.

The drawback to this approach is that it can be difficult for users to determine when to apply the techniques. This paper takes an alternate approach designed to make it easier to determine when to apply the techniques to an application or data set, as follows:

- Describe an application or data set.
- Present simple efficiency techniques for the application or data set.

Efficiency techniques are presented for the following programming tasks.

1. Create a SAS data set by reading long records from a flat file with an INPUT statement. Keep selected records based on the values of only a few incoming variables.
2. Create a new SAS data set by reading an existing SAS data set with a SET statement. Keep selected observations based on the values of only a few incoming variables.

3. Select only some observations from a SAS data set. The selected data are used as input to a SAS procedure, but are not otherwise needed.
4. In IF, WHERE, DO WHILE, or DO UNTIL statements, use OR operators or an IN operator to test if at least one of a group of conditions is true. In IF, WHERE, DO WHILE, or DO UNTIL statements, use AND operators to test if all of a group of conditions are true.
5. Select observations from a SAS data set with a WHERE statement.
6. In a DATA step, read a SAS data set with many variables to create a new SAS data set. Only a few of the variables are needed in the DATA step or the new SAS data set.
7. Create a new SAS data set containing all observations from two existing SAS data sets. The variables in the two data sets have the same length and type.
8. Process a SAS data set in a DATA step when no output SAS data set is needed. This could occur when a DATA step is used to write reports with PUT statements, examine a data set's attributes, or generate macro variables with the CALL SYMPUT statement.
9. Execute a SAS DATA step in which the denominator of a division operation could be zero.

READ LONG RECORDS FROM A FLAT FILE, KEEPING ONLY SELECTED RECORDS

TASK

Create a SAS data set by reading long records from a flat file with an INPUT statement. Keep selected records based on the values of only a few incoming variables.

TECHNIQUE

First, read only the variables needed to determine if the record should be kept. Test the values of the variables, and only read the rest of the record if necessary.

EXAMPLE

Read 2000 byte records from a flat file. Keep the record (include it in the resulting SAS data set) if NETINC is greater than 100, and otherwise discard it.

Method 1, less efficient.

```
data income;
  infile salesdata;
  input @1    shoetype 8.
        @9    netinc 8.
        @17   nextvar 8.
              .
              .
        @1993 lastvar 8.;
  if netinc > 100;
run;
```

Method 2, more efficient.

```
data income;
  infile salesdata;
  input@9 netinc 8. @;
  if netinc > 100;
  input @1    shoetype 8.
        @17   nextvar 8.
               .
               .
```

```
        @1993 lastvar 8.;
run;
```

In method 1, all 2000 bytes are read, and the current record is kept if NETINC is greater than 100. In method 2, the first INPUT statement reads only the variable NETINC. If NETINC is greater than 100, the second INPUT statement reads the rest of the current record, and the current record is kept. The trailing @ at the end of the first INPUT statement ensures that the current record is available to re-read. If NETINC is not greater than 100, the current record is discarded, and only 8 bytes are read instead of 2000.

NOTES

1. The following statement is an example of a subsetting IF statement.

```
if netinc > 100;
```

Subsetting IF statements test a condition. If the condition is true, the SAS system continues to process the current observation. Otherwise, the SAS system discards the observation and begins processing the next observation. Subsetting IF statements can be distinguished from IF-THEN/ELSE statements because they do not contain a THEN clause.

The following statements are equivalent.

```
if netinc > 100;
if not (netinc > 100) then delete;
if netinc <= 100 then delete;
```

2. If all 2000 bytes have the same informat, then the following code is equivalent to method 2. Moving the informat to the end of the second INPUT statement makes the code easier to read.

```
data income;
  infile salesdata;
  input @ 0009 netinc 8. @;
  if netinc > 100;
  input @1 shoetype 8.
        @17
        (nextvar
            .
        lastvar)
        (8.) ;
run;
```

SUBSET A SAS DATA SET TO CREATE A NEW SAS DATA SET

TASK

Create a new SAS data set by reading an existing SAS data set with a SET statement. Keep selected observations based on the values of only a few incoming variables.

TECHNIQUE

Use a WHERE statement instead of a subsetting IF statement.

A WHERE statement and a subsetting IF statement both test a condition to determine if the SAS system should process an observation. They differ as follows:

- A WHERE statement tests the condition before an observation is read into the SAS program data vector (PDV). If the condition is true, the observation is read into the PDV and processed. Otherwise, the observation is not read into the PDV, and processing continues with the next observation.
- A subsetting IF statement tests the condition after an observation is read into the PDV. If the condition is true, the SAS system continues processing the current observation. Otherwise, the

observation is discarded, and processing continues with the next observation.

EXAMPLE

Create SAS data set TWO from SAS data set ONE. Keep only observations where SALES is not equal to zero and INCOME is greater than 10.

Method 1, less efficient.

```
data two;
  set one ;
  if sales ne 0 and income > 10 ;
  more SAS statements
run;
```

Method 2, more efficient.

```
data two;
  set one ;
  where sales ne 0 and income > 10 ;
  more SAS statements
run;
```

In method 1, all observations are read, and observations not meeting the selection criteria are discarded.

In method 2, the WHERE statement ensures that observations not meeting the selection criteria are not read.

NOTES

1. At one time, a WHERE statement could be much more efficient than a subsetting IF statement, but in recent SAS releases, performance is quite similar.
2. WHERE statements can perform less efficiently if they include SAS functions and in a few other cases.
3. Because WHERE statements process data before they are read into the PDV, they cannot include variables that are not part of the incoming data set, such as the following:

- Variables you create in the current DATA step.
- Variables automatically created by the SAS system in the DATA step, such as FIRST.variables, LAST.variables, and _N_.

If data set ONE does not include the variable TAX, the following DATA step generates an error.

```
data two;
  set one ;
  tax = income / 2 ;
  where tax > 5 ;
  more SAS statements
run;
```

To prevent this error, use a subsetting IF statement instead of a WHERE statement, as follows.

```
if tax > 5 ;
```

4. To improve efficiency for large data sets, experienced users can combine WHERE statements with indexes.

SUBSET A SAS DATA SET FOR USE IN A PROC STEP

TASK

Select only some observations from a SAS data set. The selected data are used as input to a SAS

procedure, but are not otherwise needed.

TECHNIQUE

Use a WHERE statement in the PROC step to select observations and eliminate the DATA step.

EXAMPLE

Execute PROC PRINT on observations in data set ONE in which SALES is greater than 0.

Method 1, less efficient.

```
data two;
  set one ;
  if sales > 0 ;
run;
proc print data = two;
run;
```

Method 2, less efficient.

```
data two;
  set one ;
  where sales > 0 ;
run;
proc print data = two;
run;
```

Method 3, more efficient.

```
proc print data = one;
 where sales > 0 ;
run;
```

In method 1, the IF statement could be less efficient than a WHERE statement, and an intermediate data set is created. In method 2, an intermediate data set is created. In method 3, no unnecessary data sets are created.

NOTES

1. The following statement is equivalent to the statements in method 3.

```
proc print data=one(where=(sales>0));
```

TEST MULTIPLE CONDITIONS WITH IN, OR, OR AND OPERATORS

TASK

In IF, WHERE, DO WHILE, or DO UNTIL statements, use OR operators or an IN operator to test whether at least one of a group of conditions is true.

In IF, WHERE, DO WHILE, or DO UNTIL statements, use AND operators to test whether all of a group of conditions are true.

TECHNIQUE

Order the conditions to minimize the number of comparisons required by the SAS system, as follows:

- For OR operators or an IN operator, order the conditions in descending order of likelihood. Put the condition most likely to be true first, the condition second most likely to be true second, and so on.
- For AND operators, order the conditions in ascending order of likelihood. Put the condition most likely to be false first, the condition second most likely to be false second, and so on.

When the SAS system processes an IF, WHERE, DO WHILE, or DO UNTIL statement, it tests the minimum number of conditions needed to determine if the statement is true or false. This technique is known as Boolean short circuiting. Prior information is often available about data being processed. Use this information to improve program efficiency. Order the conditions in IF, WHERE, DO WHILE, and DO UNTIL statements to take advantage of Boolean short circuiting.

An IN operator can be more efficient than a series of OR operators, though this is probably only noticeable when the number of conditions is large. The IN operator can make programs easier to read.

EXAMPLES

The examples in this section use SAS data set ONE, which has 50,000 observations, as follows:

- CTRY is expected to be 'czech' in about 20,000 observations, 'hungary' in about 5000 observations, 'belgium' in about 2000 observations, and 'slovakia' in about 1000 observations.
- SALES is expected to be greater than 900 in about 10,000 observations.
- INCOME is expected to be greater than 500 in about 1000 observations.

1. Use an IF-THEN statement to set TYPE to 1 if CTRY is equal to one of four values. The following two examples are efficient because they order the values of CTRY from most likely to least likely. The first statement could be slightly more efficient because it uses the IN operator.

Using an IN operator.

```
data two ;
  set one ;
  if ctry in('czech','hungary','belgium', 'slovakia') then type = 1 ;
```

Using OR operators.

```
data two ;
  set one ;
  if ctry = 'czech' or ctry = 'hungary' or ctry= 'belgium'
     or ctry= 'slovakia' then type = 1 ;
```

2. Use a WHERE statement to select observations in which CTRY is equal to 'czech' or 'slovakia'. The following two examples are efficient because they order the values of CTRY from most likely to least likely. The first statement could be slightly more efficient because it uses the IN operator.

Using an IN operator.

```
data two ;
  set one ;
  where ctry in('czech','slovakia') ;
```

Using an OR operator.

```
data two ;
  set one ;
  where ctry = 'czech' or ctry = 'slovakia' ;
```

3. Use a WHERE statement to select observations in which CTRY is equal to 'czech', SALES is greater than 900, and INCOME is greater than 500. The following example is efficient because it orders the conditions from most likely to be false to least likely to be false.

```
data two ;
  set one ;
  where income > 500 and sales > 900 and ctry = 'czech' ;
```

NOTES

1. If prior information about the data is not available, PROC FREQ can provide information about the frequency of the conditions being tested.

2. SAS sometimes converts code with a series of OR operators to equivalent code with an IN operator to improve efficiency, as in the following code. There's no known rule for when this conversion takes place.

```
data test; /* Make data to test with */
    myvalue=1;
run;
data test2;
  set test;
  where myvalue=1 or myvalue=2 or myvalue=3 or myvalue=4;
run;
```

When the code was run in SAS 9.4TS1M4 on Windows and SAS 9.4TS1M6 in Linux, the log contained the following result for the second DATA step. The WHERE clause uses the IN operator.

```
4       data test2;
5         set test;
6         where myvalue=1 or myvalue=2 or myvalue=3 or myvalue=4;
7       run;

NOTE: There were 1 observations read from the data set WORK.TEST.
      WHERE myvalue in (1, 2, 3, 4);
NOTE: The data set WORK.TEST2 has 1 observations and 1 variables.
```

SUBSET A SAS DATA SET WITH WHERE STATEMENT OPERATORS

TASK

Select observations from a SAS data set with a WHERE statement.

TECHNIQUE

Previous sections of this paper demonstrated the WHERE statement. This section describes *WHERE statement operators*, several useful operators that can be used only in WHERE statements.

BETWEEN-AND OPERATOR

The BETWEEN-AND operator is used to test whether the value of a variable falls in an inclusive range. This operator provides a convenient syntax but no additional functionality. The following three statements are equivalent.

```
where salary between 4000 and 5000 ;
where salary >= 4000 and salary <= 5000 ;
where 4000 <= salary <= 5000 ;
```

CONTAINS OR QUESTION MARK (?) OPERATOR

The CONTAINS operator is used to test whether a character variable contains a specified string. CONTAINS and ? are equivalent.

The CONTAINS operator is case sensitive; upper and lower case characters are not equivalent. In the following DATA step, NAME is a character variable in data set ONE. Observations in which NAME contains the characters 'JO' are selected. For example, JONES, JOHN, HOJO are selected, but JAO or John are not selected. The CONTAINS operator is somewhat comparable to the SAS functions INDEX and INDEXC.

```
data two;
  set one ;
  where name contains 'JO' ;
  more SAS statements
```

The following statements are equivalent.

```
where name ? 'JO' ;
where name contains 'JO' ;
```

IS MISSING or IS NULL operator

The IS MISSING operator is used to test whether a character or numeric variable is missing. IS MISSING and IS NULL are equivalent.

Example 1. Select observations in which the value of the numeric variable SALES is missing. The following three statements are equivalent.

```
where sales is null ;
where sales is missing ;
where sales = . ;
```

Example 2. Select observations in which the value of the character variable NAME is not missing. The following three statements are equivalent.

```
where name is not null ;
where name is not missing ;
where name ne "" ;
```

The IS MISSING operator allows you to test whether a variable is missing without knowing if the variable is numeric or character, preventing the errors generated by the following statements.

- This statement generates an error if NAME is a character variable.

  ```
  where name = . ;
  ```

- This statement generates an error if SALES is a numeric variable.

  ```
  where sales = "" ;
  ```

A numeric variable whose value is a special missing value (a-z or an underscore) is recognized as missing by the IS MISSING operator.

LIKE OPERATOR

The LIKE operator is used to test whether a character variable contains a specified pattern. A pattern consists of any combination of valid characters and the following wild card characters:

- The percent sign (%) represents any number of characters (0 or more).
- The underscore (_) represents any single character.

The LIKE operator provides some of the functionality of the UNIX command grep. It is much more powerful than the SAS functions INDEX or INDEXC, which must be used multiple times to search for complex character patterns. The LIKE operator is case sensitive; upper and lower case characters are not equivalent.

Example 1. Select observations in which the variable NAME begins with the character 'J', is followed by 0 or more characters, and ends with the character 'n'. John, Jan, Jn, and Johanson are selected, but Jonas, JAN, and AJohn are not selected.

```
data two;
  set one ;
  where name like 'J%n' ;
  more SAS statements
```

Example 2. Select observations in which the variable NAME begins with the character 'J', is followed by any single character, and ends with the character 'n'. Jan is selected, but John, Jonas, Jn, Johanson, JAN, and AJohn are not selected.

```
data two;
  set one ;
```

```
  where name like 'J_n';
  more SAS statements
```

Example 3. Select observations in which the variable NAME contains the character 'J'. Jan, John, Jn, Johanson, Jonas, JAN, AJohn, TAJ, and J are selected, but j, Elijah, and jan are not selected.

```
data two ;
  set one ;
  where name like '%J%';
  more SAS statements
```

SOUNDS-LIKE (=*) OPERATOR

The Sounds-like (=*) operator uses the Soundex algorithm to test whether a character variable contains a spelling variation of a word. This operator can be used to perform edit checks on character data by checking for small typing mistakes, and will uncover some but not all of the mistakes.

In the following example, the WHERE statement keeps all but the last two input records.

```
data one;
  input tankname $1-20;
  length tankname $20;
  datalines;
new york tank
new york tank.
NEW YORK tank
new york tnk
new yrk tank
ne york tank
neww york tank
nnew york tank
ew york tank
new york mets
;
run;

data two ;
  set one;
  where tankname=* 'new york tank' ;
run;
```

To identify cases where leading characters are omitted from a character variable, use the Sounds-like operator multiple times in a WHERE statement, removing one additional character in each clause of the WHERE statement, as follows.

```
where tankname=* 'new york tank'
or tankname=* 'ew york tank'
or tankname=* 'w york tank'
or tankname=* 'york tank' ;
```

SAME-AND OPERATOR

The SAME-AND operator is used to add more clauses to a previous WHERE statement within a step without reentering the original clauses. The SAME-AND operator reduces keystrokes during an interactive session, but can make programs harder to understand. It is not recommended for large programs or applications that have code stored in multiple files.

Example. The second WHERE statement for data set TWO selects observations in which SALES is greater than 100, INCOME is less than 10, and SHOETYPE is 1.

```
data two ;
```

```
set one ;
where sales > 100 and income < 10 ;
more SAS statements
where same-and shoetype = 1 ;
more SAS statements
```

NOTES

1. See the “WHERE-Expression Processing” chapter in *SAS 9.4 Language Reference: Concepts, Sitxth Edition* for more information about WHERE statement operators.

READ A SAS DATA SET, BUT NEED ONLY A FEW OF MANY VARIABLES

TASK

In a DATA step, read a SAS data set with many variables to create a new SAS data set. Only a few of the variables are needed in the DATA step or the new SAS data set.

TECHNIQUE

Use the KEEP= or DROP= option in the SET statement to prevent unnecessary variables from being read into the SAS program data vector (PDV) or the new data set.

EXAMPLE

Create SAS data set TWO from SAS data set ONE. Data set ONE contains variables A and X1-X1000. The only variables needed in data set TWO are A and B.

Method 1, less efficient.

```
data two;
  set one ;
  b = a*1000 ;
run;
```

Method 2, less efficient.

```
data two;
  set one ;
  keep a b ;
  b = a*1000 ;
run;
```

Method 3, incorrect result.

```
data two (keep = b) ;
  set one (keep = a) ;
  b = a*1000 ;
run;
```

Method 4, more efficient.

```
data two ;
  set one (keep = a) ;
  b = a*1000 ;
run;
```

In method 1, 1001 variables are read into the PDV from data set ONE, and 1002 variables are written to data set TWO. 1000 of the variables are not needed.

In method 2, 1001 variables are read into the PDV from data set ONE, and two variables are written to data set TWO. The KEEP statement specifies that only A and B are written to data set TWO. Variables

X1-X1000 are not written to data set TWO, but are still read unnecessarily into the PDV from data set ONE.

In method 3, one variable, A, is read into the PDV from data set ONE, and one variable, B, is written to data set TWO.

In method 4, one variable, A, is read into the PDV from data set ONE, and two variables, A and B, are written to data set TWO. This is the most efficient method.

NOTES

1. If variables X1-X1000 are needed during the execution of DATA step TWO (for example, if they are used to calculate B), but are not needed in the output data set, then use method 2.
2. If A is the only variable needed during the execution of DATA step TWO, and B is the only variable needed in the output data set, then use method 3.
3. In method 3, the following statement is equivalent to (KEEP = A);

```
(drop = x1-x1000);
```

4. In method 2, either of the following statements are equivalent to KEEP A B ;.

```
drop x1-x1000;
data two(keep = a b);
```

CONCATENATE (APPEND) ONE SAS DATA SET TO ANOTHER

TASK

Create a new SAS data set containing all observations from two existing SAS data sets. The variables in the two data sets have the same length and type.

TECHNIQUE

Use PROC APPEND instead of a SET statement. The SET statement reads both data sets. PROC APPEND reads the data set to be concatenated, but does not read the other data set, known as the BASE data set.

EXAMPLE

Concatenate SAS data set TWO to SAS data set ONE.

Method 1, less efficient.

```
data one;
  set one two ;
run;
```

Method 2, more efficient.

```
proc append base = one data = two ;
run;
```

In method 1, the SAS system reads all observations in both data sets. In method 2, the SAS system reads only the observations in data set TWO.

NOTES

1. Since PROC APPEND reads only the second data set, set BASE= to the larger data set if, as in the following example, the order of the data sets does not matter.

```
proc append base = one data = two ;
run ;
proc sort data = one ;
```

```
  by var1 ;
run ;.
```

2. The documentation for PROC APPEND in the *Base SAS 9.4 Procedures Guide, Seventh Edition* describes how to concatenate data sets that contain different variables or variables with the same name but different lengths or types (character versus numeric).

EXECUTE A DATA STEP, BUT NO OUTPUT SAS DATA SET IS NEEDED

TASK

Process a SAS data set in a DATA step when no output SAS data set is needed. This could occur when a DATA step is used to write reports with PUT statements, examine a data set's attributes, or generate macro variables with the CALL SYMPUT statement.

TECHNIQUE

Name the data set the reserved name "_NULL_" to avoid creating an output SAS data set.

EXAMPLE

Use PUT statements to write information from data set ONE. No output data set is needed.
Method 1, less efficient.

```
data two;
  set one;
  put @1 var1 @11 var2 ;
  more PUT and printing statements
run;
```

Method 2, less efficient.

```
data one;
  set one;
  put @1 var1 @11 var2 ;
  more PUT and printing statements
run;
```

Method 3, more efficient.

```
data _null_;
  set one;
  put @1 var1 @11 var2 ;
  more PUT and printing statements
run;
```

In method 1, the SAS system creates an unnecessary data set, TWO. In method 2, data set ONE is unnecessarily recreated. In method 3, an output data set is not created.

NOTES

1. Using the statement DATA; instead of DATA _NULL_; causes the creation of an output data set called DATA*n*, where *n* is 1,2,3,... (the first such data set is called DATA1, the second is called DATA2, and so on).

EXECUTE A SAS DATA STEP IN WHICH THE DENOMINATOR OF A DIVISION OPERATION COULD BE ZERO

TASK

Execute a SAS DATA step in which the denominator of a division operation could be zero.

TECHNIQUE

Test the value of the denominator, and divide only if the denominator is not zero. Programs execute substantially faster if you prevent the SAS system from attempting to divide by zero.

EXAMPLE

Execute a simple DATA step. Set the variable QUOTIENT to the value NUM/DENOM.

Input data set for this example.

```
data one;
  input num denom ;
datalines;
2 1
6 2
10 0
20 2
9 0
;
run;
```

Method 1, less efficient.

```
data two;
  set one;
  quotient = num/denom;
run;
```

Method 2, more efficient.

```
data two;
  set one;
  if denom ne 0
  then quotient = num/denom;
  else quotient =.;
run;
```

Method 3, different result.

```
data two;
  set one;
  quotient = divide(num,denom);
run;
```

Method 1 is less efficient than Method 2 because the SAS system attempts to divide by zero. Since attempting to divide by zero results in a missing value, Methods 1 and 2 generate the same output data set, TWO, as follows.

OBS	NUM	DENOM	QUOTIENT
1	2	1	2
2	6	2	3
3	10	0	.
4	20	2	10
5	9	0	.

Method 3 is also more efficient than Method1 but generates a different result when DENOM is zero. In observations 3 and 5 of DATA set TWO, QUOTIENT is the special missing value .I, which is displayed as I.

```
OBS             NUM        DENOM    QUOTIENT

 1               2           1         2
 2               6           2         3
 3              10           0         I
 4              20           2        10
 5               9           0         I
```

In general, for QUOTIENT=DIVIDE(NUM/DENOM), if DENOM is 0, QUOTIENT has the following values.

Value of NUM	**Value of QUOTIENT**
Positive value	.I
Negative value	.M
0 or .	.
Any special numeric missing value	The value of NUM (e.g., if NUM=.M, QUOTIENT=.M)

NOTES

1. When the SAS system attempts to divide by zero, a note similar to the following is printed to the SAS log.

```
NOTE: Division by zero detected at line 329 column 11.
NUM=10 DENOM=0 QUOTIENT=. _ERROR_=1 _N_=3
NOTE: Division by zero detected at line 329 column 11.
NUM=9 DENOM=0 QUOTIENT=. _ERROR_=1 _N_=5
NOTE: Mathematical operations could not be performed at the following
places.
The results of the operations have been set to missing values.
   Each place is given by: (Number of times) at (Line):(Column).
   2 at 329:11
```

2. A tremendous performance improvement from preventing the SAS system from attempting to divide by zero was first achieved in Release 6.06 of the SAS system for MVS mainframes. A program processed a SAS data set with 200,000 observations and 16 variables, performing 8 divisions per observation, of which approximately half were attempts to divide by zero. Recoding the program to test for division by zero reduced the CPU time from 4 minutes to 7 seconds.

3. Subsequent tests with Linux and Windows SAS releases through SAS 9.4 continue to show large performance differences. For example, a recent test with Linux SAS 9.4TS1M6 was as follows:

- A data set had 50 million numeric values (1 million observations, 50 variables), and 20% of the values (10 million) were 0.
- All 50 million values were used in a division operation, 10/*value*, so the denominator was zero 10 million times.
- Method 1 took 34 seconds, and Methods 2 and 3 took 2 seconds.

CONCLUSION

This paper presented simple efficiency techniques that can benefit inexperienced SAS software users on all platforms. Each section of the paper began with a description of an application or data set, followed by efficiency techniques for that application or data set. The author of this paper hopes that presenting the techniques this way will make it easier for users to determine when to apply these techniques.

REFERENCES

Polzin, Jeffrey A, (1994), "DATA Step Efficiency and Performance," in the Proceedings of the Nineteenth Annual SAS Users Group International Conference, 19, 1574 - 1580.

SAS Institute Inc. (2017), "Base SAS® 9.4 Procedures Guide, Seventh Edition, Second Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2016), "SAS 9.4 Language Reference: by Name, Product, and Category," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2016), "SAS® 9.4 Language Reference: Concepts, Sixth Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1990), "SAS Programming Tips: A Guide to Efficient SAS Processing," Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The following people contributed extensively to the development of this paper: Donna Hill, Julia Meredith, Steve Schacht, and Peter Sorock who were previously at the Federal Reserve Board; Mike Bradicich at Vistech; and Ross Bettinger, Rick Langston, Kevin McGowan, and Jason Secosky who were previously or are currently at SAS Institute. Their support is greatly appreciated.

CONTACT INFORMATION

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

Bruce Gilsen
bruce.gilsen@frb.gov