ABSTRACT

Efficiency may be defined as conservation of resources in performing a task. Optimizing the use of computer resources is important in a clinical data processing environment because large data sets are routinely manipulated and then analyzed and reported in batch mode. Program maintenance time also needs to be minimized because SAS® code is often copied, modified, and shared by programmers, statisticians, and other users. This paper presents programming techniques for improving performance on most platforms without attempting to quantify efficiency through benchmarking. The techniques are illustrated using specific clinical research applications.

INTRODUCTION

There are several obstacles to writing efficient programs in clinical data processing. Time pressures often force the programmer to provide output to users without considering resource utilization or maintenance requirements. Also, SAS code may be called from external files such as autocall macros which were developed by a different programmer. Programs developed in version 5 of the SAS system may not have been updated to take advantage of the efficient features available in version 6.

The following techniques are provided to illustrate alternative ways of accomplishing tasks. The programmer should determine which resources (memory, cpu, storage, programmer time, etc.) are most limited in the environment and maximize performance subject to the constraints.

PROGRAMMING TECHNIQUES

1. Use formats and informats to reduce sorts and merges.

Example: Clinical diary data (data set clin.diary) are received from an outside source with patients identified by a screening number (variable spat) and multiple records per patient. All data sets require the randomized patient number (variable rpat) for analysis and reporting. One data set, clin.patid, exists with one record per patient containing, among other variables, spat and rpat. The task is to enter RPAT into the permanent diary data set.

```sas
/*** Common Solution: Sort and Merge ***/
PROC SORT DATA=CLIN.DIARY OUT=DIARY;
BY SPAT;
RUN;
PROC SORT DATA=CLIN.PATID(KEEP=SPAT RPAT) OUT=PATID;
BY SPAT;
RUN;
DATA CLIN.DIARY;
MERGE DIARY(IN=INDY) PATID;
BY SPAT;
IF INDY;
RUN;
```

```sas
/*** Efficient Technique: Generate Format. Saves cpu time by reducing costly sorts of large data sets and slow merges ***/
DATA IN(RENAME=(SPAT=START RPAT=LABEL»;
SET CLIN.PATID(KEEP=SPAT RPAT);
RETAIN FMTNAME 'SRNUM';
RUN;
/*** Data set IN with variables START, LABEL, and FMTNAME is an input control data set for use with PROC FORMAT. For numeric variables, use an informat ***/
PROC FORMAT CNTLIN=IN;
RUN;
/*** Update permanent data set, length of RPAT is $4 ***/
DATA CLIN.DIARY; SET CLIN.DIARY;
RPAT = PUT (SPAT, $RNUM4.);
RUN;
```

2. Use PROC SQL joins for many-to-many merges.

Example: Laboratory normal ranges are stored in a SAS data set (RANGE) with lab code, an age range, and the reference range. A subset appears below.

<table>
<thead>
<tr>
<th>Labcode</th>
<th>Agelo</th>
<th>Agehi</th>
<th>Rangelo</th>
<th>Rangehi</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0</td>
<td>20</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>001</td>
<td>21</td>
<td>40</td>
<td>15</td>
<td>125</td>
</tr>
<tr>
<td>001</td>
<td>41</td>
<td>99</td>
<td>20</td>
<td>150</td>
</tr>
<tr>
<td>002</td>
<td>0</td>
<td>20</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>002</td>
<td>21</td>
<td>40</td>
<td>110</td>
<td>220</td>
</tr>
<tr>
<td>002</td>
<td>41</td>
<td>99</td>
<td>125</td>
<td>240</td>
</tr>
</tbody>
</table>
The patient laboratory data is in data set LAB with variables PATIENT, AGE, VISIT, LABCODE, and LABVALUE as follows.

<table>
<thead>
<tr>
<th>Patient</th>
<th>Age</th>
<th>Visit</th>
<th>Labcode</th>
<th>Labvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>25</td>
<td>1</td>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>0101</td>
<td>25</td>
<td>2</td>
<td>001</td>
<td>150</td>
</tr>
<tr>
<td>0101</td>
<td>25</td>
<td>1</td>
<td>002</td>
<td>200</td>
</tr>
<tr>
<td>0101</td>
<td>25</td>
<td>2</td>
<td>002</td>
<td>250</td>
</tr>
</tbody>
</table>

The task is to create a data set containing the patient lab data with the correct normal range, based on the patient’s age. Merging the data sets using LABCODE as the by-variable would create incorrect results because the MERGE statement creates a data set containing the maximum number of observations per by-variable while retaining variables from the data sets with less observations. This is sometimes called a “fuzzy” merge. SAS hints that something is wrong with the following message in the log: NOTE: MERGE statement has more than one data set with repeats of BY variables.

The proper method of selecting the normal range based on age is to join each observation in the lab data set with each observation in the range data set. The result is known as the Cartesian product of the two data sets. The data sets are matched based on lab code and the correct age range is selected. The simplest way of accomplishing the task is using PROC SQL.

PROC SQL;
CREATE VIEW LABRANGE AS
SELECT LAB.*, RANGELO, RANGEHI
FROM LAB, RANGE
WHERE LAB.LABCODE = RANGE.LABCODE
AND AGELO <= AGE <= AGEHI;

The SQL view created, WORK.LABRANGE, may be processed in the same manner as any SAS data set.

3. Optimize the use of arrays.

Arrays can be made more efficient by using temporary arrays to store constants and setting the lower bound of the arrays to 0 instead of 1.

Example: Vital sign values are compared to ranges to check for out of range data. Values outside of the range are output to a data set containing patient number, visit, the variable out of range, and value for reporting. The five vital sign variables contain systolic BP, diastolic BP, pulse, respiration rate, and temperature.

PROC SQL;
CREATE VIEW LABRANGE AS
SELECT LAB.*, RANGELO, RANGEHI
FROM LAB, RANGE
WHERE LAB.LABCODE = RANGE.LABCODE
AND AGELO <= AGE <= AGEHI;

The SQL view created, WORK.LABRANGE, may be processed in the same manner as any SAS data set.

3. Optimize the use of arrays.

Arrays can be made more efficient by using temporary arrays to store constants and setting the lower bound of the arrays to 0 instead of 1.

Example: Vital sign values are compared to ranges to check for out of range data. Values outside of the range are output to a data set containing patient number, visit, the variable out of range, and value for reporting. The five vital sign variables contain systolic BP, diastolic BP, pulse, respiration rate, and temperature.

*** Processing arrays efficiently: The constants for the lower and upper limits of the ranges are stored in temporary arrays. The array bounds are set to {0:4} instead of the default {1:5}. The CALL VNAME routine is used to put variable names from the array into variable VARNAME. ***/

DATA OUTRANGE; SET CLIN.VITALS;
ARRAY _VAR [0:4] _BPSYS _BPDIA _PULSE _RESP _TEMP;
ARRAY _LO [0:4] _TEMPORARY_ (90 50 50 10 95); /* Lower limits */
ARRAY _HI [0:4] _TEMPORARY_ (145 100 120 24 102); /* Upper limits */
LENGTH VARNAME $8;
DO I = 0 TO 4;
IF ( _Z < _VAR[I] < _LO[I] )
OR _VAR[I] > _HI[I] THEN DO;
CALL VNAME(_VAR[I], VARNAME);
VALUE = _VAR[I];
OUTPUT;
END;
END;
KEEP PATIENT VISIT VARNAME VALUE;
RUN;

4. Use WHERE statements in procedures instead of creating subset data sets.

Example: An efficacy data set contains patient number, visit, a coded variable representing the parameter measured (PAR), and the evaluation (VALUE) sorted by PATIENT, PAR, and VISIT. The task is to calculate the baseline value as the mean of visits 0 through 2.

*** Inefficient --- unnecessary data step ***/
DATA BASE; SET EFFICACY;
IF 0 <= VISIT <= 2;
RUN;
PROC MEANS DATA=BASE MEAN=BASE;
*** More statements and options ***/

PROC MEANS DATA=EFFICACY NOPRINT WHERE 0 <= VISIT <= 2;
BY PATIENT PAR;
OUTPUT OUT=BASELINE MEAN=BASE;
RUN;
5. Use a CLASS statement in the MEANS and SUMMARY procedures for unsorted data sets.

Example: A summary table requires n, mean, and standard deviation statistics on patient age grouped by drug from an unsorted demographics data set.

```sas
/*\*\* Sort data set and use by-statement ***/
PROC SORT DATA=CLIN.DEMOG OUT=AGE;
   BY DRUG;
RUN;
PROC MEANS N MEAN STD DATA=AGE NOPRINT;
   BY DRUG;
   VAR AGE;
OUTPUT OUT=AGE N=AGEN MEAN=AGEMEAN STD=AGESTD;
RUN;

/*\*\* The CLASS statement produces group statistics without the need to sort the input data set. The NWAY option specifies that statistics be output for each drug and not for the entire data set. ***/
PROC MEANS N MEAN STD DATA=CLIN.DEMOG NWAY NOPRINT;
   CLASS DRUG;
   VAR AGE;
OUTPUT OUT=AGE N=AGEN MEAN=AGEMEAN STD=AGESTD;
RUN;

The CLASS statement causes the procedure to use more cpu time and memory than a BY statement. This is offset by eliminating the PROC SORT. If the input data set is already sorted, then BY-group processing is more efficient.

Without the NWAY option, the output data would contain additional observations with statistics for the entire data set and for all combinations of class variables. The observations are identified by default variable _TYPE_. In the example, statistics for the entire data set would have a _TYPE_ value of 0, and each drug group would have statistics with a _TYPE_ value of 1.

6. Use the INPUT and PUT functions to explicitly convert data types.

SAS automatically performs numeric-character conversions on values when required by the operation. The conversions are costly in terms of cpu time and should be avoided by using the INPUT and PUT functions.

Example: A diary data set contains patient severity scores (0 to 5) for 4 symptoms in character variables (length 5) SYMP1-SYMP4. A total symptom score is calculated as the sum of the four symptoms.

```sas
/*\*\* Automatic conversion: INEFFICIENT ***/
SYMPTOT = SUM (SYMP1, SYMP2, SYMP3, SYMP4);
The following message is written to the SAS log.
NOTE: Character values have been converted to numeric values at the places given by: (Line):(Column).

/*\*\* More Efficient: Use INPUT function to convert values ***/
SYMPTOT = SUM(INPUT(SYMP1,1.), INPUT(SYMP2,1.), INPUT(SYMP3,1.), INPUT(SYMP4,1.));
```

7. Use the NMISS function to avoid missing value generation.

Calculations in which 1 or more variables are missing cause SAS to propagate the missing value and return a missing result. The following message is written to the SAS log.

NOTE: Missing values were generated as a result of performing an operation on missing values. Each place is given by (Number of times) at (Line):(Column).

Performing operations on missing values and tracking of each occurrence consumes cpu time. The larger the number of missing values generated, the more inefficient the program.

Example: An efficacy data set contains a variable representing result (VALUE) and a variable representing baseline (BASE), either of which may be missing in some observations. The task is to calculate the difference from baseline (DIFF).

```sas
/*\*\* The NMISS function returns the number of missing values, including special missing (.A, .B, etc.). Only if value returned is 0 (neither value missing) will the calculation be performed. ***/
IF NMISS(VALUE, BASE) = 0
   THEN DIFF=VALUE-BASE;
```
8. Subset external data files early in the data step.

When reading external data into a SAS data set, computer resources can be conserved by using an INPUT statement with a trailing @ sign and then subsetting based on input variables. The subset variables should be read first, then the condition tested for, and the other variables read in a second INPUT statement.

Example: Lab data is received from a central laboratory to be read into SAS. A 4-character field in the input data identifies patient identifier, from which only randomized patients are selected. Randomized patients contain only numbers (0-9) in the patient identifier, non-randomized patients contain letters in the patient identifier. The task is to read the lab data, selecting only the randomized patients.

FILENAME CLINFILE 'CLINICAL.LAB.DAT';
DATA LAB;
INFILE CLINFILE MISSOVER;
*** Read patient number and hold input line with trailing @ ***
INPUT @10 PATIENT $4. @;
*** Only read data for randomized patients ***
IF VERIFY(PATIENT,'0123456789') = 0 THEN INPUT @20 LABDATE MMDDYY8. @30 LABCODE $3. @40 LABVALUE$12.;
*** Return to top of data step if not randomized ***
ELSE DELETE;
*** Other SAS processing ***
RUN;

9. Use LENGTH statements to conserve storage.

Storage space can be significantly reduced by using the LENGTH statement to avoid default lengths assigned by SAS. The default length of 8 bytes for numeric variables is usually appropriate for decimal values, but should be reduced for integers. SAS determines the length of a character variable during the compile phase based on the first occurrence of a value. Some factors to consider in the use of the LENGTH statement follow.

The LENGTH statement works differently on character and numeric variables. For character variables, the LENGTH statement determines the length both in the program data vector and in the output data set. For numeric variables, only the output data set variable lengths are affected. Consequently, placement of the LENGTH statement in the data step is important.

FILENAME CLINFILE 'CLINICAL.LAB.DAT';
DATA LAB;
INFILE CLINFILE MISSOVER;
*** Read patient number and hold input line with trailing @ ***
INPUT @10 PATIENT $4. @;
*** Only read data for randomized patients ***
IF VERIFY(PATIENT,'0123456789') = 0 THEN INPUT @20 LABDATE MMDDYY8. @30 LABCODE $3. @40 LABVALUE$12.;
*** Return to top of data step if not randomized ***
ELSE DELETE;
*** Other SAS processing ***
RUN;

/*** Reduce lengths of date variables to 4 bytes. Input date lengths are 8 bytes, output lengths are 4 bytes ***/
DATA EFFICACY;
SET EFFICACY;
LENGTH EFDATE BASEDATE 4;
RUN;

/*** A similar data step generates a warning from SAS. Input length of EVALCODE is 8 ***/
DATA EFFICACY;
SET EFFICACY;
LENGTH EVALCODE $1;
RUN;
SAS issues the following warning and the length of the variable is not changed.
WARNING: Length of character variable has already been set.
Use LENGTH statements as the very first statement in the DATA step to declare length of character variables.

It is good practice to put all length statements at the beginning of the data step.

DATA EFFICACY;
LENGTH EFDATE BASEDATE 4 EVALCODE $1;
SET EFFICACY;
RUN;

The SCAN, REPEAT, and SYMGET functions return character variables of length 5200. Use a LENGTH statement to avoid this problem.

Exporting a data set to transport format increases, by 1 byte, the lengths of all numeric variables with input lengths less than 8 bytes.
In the following example an adverse experience data set is put into transport format using PROC COPY to allow movement of the data set to another host.

/*** The XPORT engine name in the LIBNAME statement causes the procedure to output the data set to transport format ***/
LIBNAME CLIN 'CLIN.SAS.DATA.LIB';
LIBNAME TRAN XPORT 'CLIN.XPORT.DAT';
PROC COPY IN=CLIN OUT=TRAN;
SELECT ADVEXP;
RUN;

Data set ADVEXP contains onset date and remission date of the adverse experience. The lengths of date variables are 4 bytes in the clinical data set. Using PROC
CONTENTS, you will notice that the lengths are 5 bytes in the transport file. The 5 byte length is maintained when the transport file is imported on the new host.

SAS increases the length of all numeric variables up to 8 bytes in order to preserve numeric precision. Since this is not an issue for most dates and other integers, use a LENGTH statement to save disk space.

10. Use RETAIN statements to save CPU time.

By default, SAS sets the values of variables assigned by INPUT or assignment statements to missing before each iteration of a data step. The RETAIN statement is used to hold the values of variables assigned by INPUT or assignment statements from the current observation for use in later observations.

It is more efficient to assign values to constants using a RETAIN statement than with an assignment statement because the RETAIN statement assignment occurs once, during data step compilation. Using an assignment statement causes SAS to assign the value to the variable at every observation.

/*** Assign Protocol and Country variables to a SAS data set ***/
DATA CLINDATA; SET CLINDATA;
RETAIN PROTOCOL 'JRF1000' COUNTRY 'USA';
/*** Other SAS processing ***/

The RETAIN statement can be used to compare values at the current observation to values at previous observations, as in comparing a value to baseline. This can save data steps compared to creating a baseline data set to be merged for the comparison.

Example: An efficacy data set contains PATIENT, VISIT, and VALUE, sorted by PATIENT and VISIT. The visits range from 0 to 10, with some visits possibly having missing values. The baseline is defined as the last value at or before visit 2. Starting at visit 3, a difference from baseline is to be calculated.

DATA EFFICACY; SET EFFICACY;
BY PAT VISIT;
/*** Hold the baseline value ***/
RETAIN BASE;
/*** IMPORTANT: Initialize to missing at beginning of new patient ***/
/*** Without this statement, values can be retained across patients ***/
IF FIRST.PAT THEN BASE=.;
/*** Get baseline from visits 0-2 ***/
IF VISIT<=2 AND NMISS(VALUE)=0 THEN BASE=VALUE;

/* Difference from baseline at later visits ***/
ELSE IF VISIT=>3 AND NMISS(BASE, VALUE)=0 THEN DIFF=VALUE-BASE;
RUN;

11. Use output from PROC CONTENTS to avoid hard coding data set names.

If all data sets in a data library are to be processed, the data set names may be read from PROC CONTENTS. By creating macro variables containing data set names, many processes can be automated.

Example: Generate macro variables containing the name of each data set (&DS1, &DS2, etc.) and the total number of data sets (&TOTDS) in a SAS data library with libref CLIN.

/*** Read all member names in SAS data library ***/
PROC CONTENTS DATA=CLIN._ALL_ MEMTYPE=DATA OUT=CONT(KEEP=MEMNAME);
RUN;
/*** Write macro variables with no output data set created ***/
DATA _NULL_; SET CONT END=EOF;
BY MEMNAME NOTSORTED;
/*** Select last of multiple observations per member ***/
IF LAST.MEMNAME THEN DO;
COUNT+1;
/*** Data set names from variable MEMNAME ***/
CALL SYMPUTCDS('COMPRESS(PUT(COUNT,4.)),MEMNAME);
END;
/*** Total number of members from final COUNT at end of file ***/
IF EOF THEN CALL SYMPUT('TOTDS',PUT(COUNT,4.));
RUN;

The macro variables can be used to process all members of the SAS data library. For example, the following macro enters the protocol into each data set.

%MACRO PR0T;
%DO I = 1 %TO &TOTDS;
   DATA CLIN.&DS&I; SET CLIN.&DS&I;
   RETAIN PROTOCOL 'JRF1000';
   RUN;
%END;
%MEND PROT;
CONCLUSION

Most programs can be made more efficient by reducing the number of steps in the processing and by changing programming techniques. The SAS log contains messages concerning efficiency which are often ignored, including missing value propagation and data type conversions. The techniques above are provided as a general guideline for common clinical data processing tasks.

REFERENCES


SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

CONTACT

Denis Michel
Janssen Pharmaceutica Inc.
1125 Trenton-Harbourton Road
Titusville, NJ 08560-0200