Paper 053-2022

Best Practices for Efficiency and Code Optimization in SAS programming.

Jay Iyengar, Data Systems Consultants LLC

ABSTRACT

There are multiple ways to measure efficiency in SAS programming; programmers' time, processing or execution time, memory, input/output (I/O) and storage space considerations. As data sets are growing larger in size, efficiency techniques play a larger and larger role in the programmers' toolkit. This need has been compounded further by the need to access and process data stored in the cloud, and due to the pandemic as programmers find themselves working remotely in distributed teams. As a criteria to evaluate code, efficiency has become as important as producing a clean log, or expected output. This paper explores best practices in efficiency from a processing standpoint, as well as others.

INTRODUCTION

Some SAS developers refer to efficiency techniques as performance tuning techniques. Besides performance of your SAS code, efficiency techniques can encompass other tasks which SAS programmers perform, and entail methods to reduce the time required to perform these tasks. For instance, programmers need to develop their code. They also need to update and modify their code periodically. In measuring efficiency, these tasks can be thought of as development time, and maintenance time respectively. In terms of skill level, the ability to apply efficiency techniques separate beginning programmers from mid-level or Senior SAS programmers. Programmers who have the first-level SAS certification, BASE SAS Programmer for SAS 9, aren't really tested on their knowledge of efficient programming methods. Efficiency techniques or methods of optimizing SAS code and processes are tested on the Advanced Programmer for SAS 9 certification exam.

DEFINING AND MEASURING EFFICIENCY

To determine the amount of time it takes to run or execute SAS code, SAS provides performance metrics which can be used to measure real processing time and CPU time. Other resources such as memory and storage space can be measured through the utilities available on your operating system. Your operating system provides information on how much memory, such as RAM, you have access to, 32K or 64K for example. It also provides how many bytes of storage space you have. Most PC's usually have Gigabytes (GB) of storage space.

Input/output or I/O time can be a difficult performance metric to measure. The timing statistics reported in the SAS log using the STIMER or FULLSTIMER option don't report it. Input/output time is really the time it takes SAS to read data from a data set, and the time it takes to write and output data to an output data set. I/O time is a subset of CPU or processing time. To the extent of my knowledge, there really is no direct performance statistic to measure it.

The time it takes to develop code is known as programming time. The time it takes to update or modify code can be thought of as maintenance time. Programming time and maintenance time can also be elusive metrics. They can be measured in relative amounts such as the more lines of code written, the more programming time is required. The fewer lines of code, the less maintenance time is needed.

CONDITIONAL LOGIC

DATA STEP programming is the data manipulation tool of BASE SAS. A common data manipulation task which programmers perform is deriving new variables based on an existing variable. This process is known as conditional logic. The BASE SAS package includes both the DATA STEP and PROC SQL which feature several conditional logic constructs. If you’re using the DATA STEP you have a choice between the IF-THEN-ELSE construct and the SELECT-WHEN construct. The conditional logic construct in PROC SQL is the case logic expression.

For performance, in the DATA STEP there are conditions when a programmer is better off using IF-THEN-ELSE over SELECT-WHEN. If you’re using character variables, and creating a new variable based on a character variable, then IF-THE-ELSE is the more efficient construct. On the other side of the coin, if you’re working with a numeric variable and creating a new variable based on a numeric variable, then SELECT-WHEN is the more efficient construct. SELECT-WHEN also processes a larger number of conditions more efficiently then IF-THEN-ELSE.

I’m going to focus more exclusively on the IF-THEN-ELSE construct in the DATA STEP. The example in Figure 1 below takes the numeric variable AGE, and creates a character variable, Age_Group. which classifies Age into 10-year Age categories. Notice that the series of IF-THEN statement does not include the ELSE keyword. As a consequence, SAS will execute each one of the 12 IF-THEN statements in this conditional logic series.

```
Data TESTING2;
     Length Age_Group $8;
   Set TESTING;

   Age = Round((Today()-BirthDate_Text__c)/365.25, 1);

   If _N_<=20 Then Put Age=;

                /* Create age group variables for Age Cohorts */
   If 0<=Age<=9 Then age_group="0-9";
   If 10<=Age<=19 Then age_group="10-19";
   If 20<=Age<=29 Then age_group="20-29";
   If 30<=Age<=39 Then age_group="30-39";
   If 40<=Age<=49 Then age_group="40-49";
   If 50<=Age<=59 Then age_group="50-59";
   If 60<=Age<=69 Then age_group="60-69";
   If 70<=Age<=79 Then age_group="70-79";
   If 80<=Age<=89 Then age_group="80-89";
   If 90<=Age<=99 Then age_group="90-99";
   If Age>=100 Then age_group="100+";
   If Age=. Then age_group="Unknown";
Run;
```

Figure 1. DATA STEP with series of IF-THEN Statements

SAS processes every statement in the IF-THEN series, regardless of whether it’s true or not. Suppose that the condition in one of the first statements happens to be true, meaning Age falls in the range of the condition. Even though the rest of the IF-THEN statements are false, SAS still executes them. The inclusion of the ELSE keyword helps solve this problem.

The example in Figure 1 has been replicated in Figure 2 below. In the code in Figure 2, an ELSE keyword has been inserted before each of the IF-THEN statements besides the first. In this example, if one of the conditions is true, then the remaining statements will not be executed, the processing stops right there. Each statement is only executed if the statements before it have already been declared false.

```
  Data TESTING2;
       Length Age_Group $8;
    Set TESTING;

    Age = Round((Today()-BirthDate_Text__c)/365.25, 1);

    If _N_<=20 Then Put Age=;

             /* Create age group variables */
   If 0<=Age<=9 Then age_group="0-9";
   Else If 10<=Age<=19 Then age_group="10-19";
   Else If 20<=Age<=29 Then age_group="20-29";
   Else If 30<=Age<=39 Then age_group="30-39";
   Else If 40<=Age<=49 Then age_group="40-49";
   Else If 50<=Age<=59 Then age_group="50-59";
   Else If 60<=Age<=69 Then age_group="60-69";
   Else If 70<=Age<=79 Then age_group="70-79";
   Else If 80<=Age<=89 Then age_group="80-89";
   Else If 90<=Age<=99 Then age_group="90-99";
   Else If Age>=100 Then age_group="100+";
   Else If Age=. Then age_group="zPending further info";

Run;
```

Figure 2. DATA STEP with a series of IF-THEN-ELSE statements

By executing only necessary statements, fewer lines of code are processed. Using IF-THEN-ELSE instead of IF-THEN results in reduced processing time or CPU. From an efficiency standpoint, it’s a best practice to use the ELSE keyword in conditional processing as IF-THEN-ELSE is more efficient than IF-THEN.

SUBSETTING DATA SETS

In the DATA STEP, SAS provides multiple constructs to subset data from SAS data sets or from external files. This encompasses both selecting observations and selecting variables. The available constructs for subsetting observations include the WHERE statement or data set option and the IF statement. The techniques for selecting variables are the KEEP and DROP data set options and statements.

As we’ll demonstrate through examples, some of these techniques are more efficient than others for particular processes. This could be because of use of one construct vs. another, a different form of the construct (statement vs. option) or the location of the construct in the DATA STEP.

The metrics used to measure the efficiency of these techniques are input/output or I/O which measures the time it takes read and write data. Input/output is considered a part of processing time or CPU.

WHERE AND IF STATEMENTS AND OPTIONS

In a DATA STEP, both WHERE and IF can be used to subset observations. WHERE can be specified as a data set option, or alternately as a statement. The data set option can be an input or output data set option. Whereas IF can only be specified as a statement, known as the subsetting IF statement.

The difference between the IF and WHERE constructs in terms of efficiency depend on which construct controls the reading vs. the writing of data in the DATA STEP. The WHERE statement or input data set option controls which observations are read from the input or source data set. However, the subsetting IF statement doesn’t control the reading of observations, but instead controls the writing of observations to an output SAS data set.

In the example in Figure 5 below, I use both techniques to subset records from a SAS data set (NEWFILE.NDF) containing over a million observations. In the first DATA STEP I use the subsetting IF statement to select records. In the second DATA STEP, I use the WHERE statement.

```
56          DATA NDF_MD1;
57          SET NEWFILE.NDF;
58          IF STATE = 'MD';
59          RUN;

NOTE: There were 1048575 observations read from the data set NEWFILE.NDF.

61           DATA NDF_MD2;
62           SET NEWFILE.NDF;
63           WHERE STATE = 'MD';
64           RUN;

NOTE: There were 22518 observations read from the data set NEWFILE.NDF
WHERE STATE='MD';
```

Figure 5. DATA steps with IF and WHERE statement.

Figure 5 contains an excerpt from the SAS log. Notes in the SAS log corresponding to each DATA STEP provide the volume of data which was read using the IF statement vs. the WHERE statement. In the first DATA STEP, the complete data set was read using the IF statement, where as in the second DATA STEP with the WHERE statement only the subset was read from the input SAS data set.

Subsetting using the WHERE statement was more efficient because a much smaller amount of data was read, which entails a significant reduction in Input/output (I/O) time.

Also, because a much smaller amount of data was processed, processing time or CPU should significantly decrease as well. For these reasons, I recommend using the WHERE statement or data set option over the IF statement for subsetting data in the DATA STEP.

If you choose to use the IF statement to subset observations its best to position it at the top of the DATA STEP. Placing IF at the top ensures it will be executed before any remaining code in the DATA STEP. The remaining code is only executed if the IF condition is true. Figure 6 shows a DATA step with optimal placement of the IF statement.

```
DATA MedU16;
      SET MedUtiliz16;

      IF CITY=’Bethesda’;

      IF PROVIDER_CITY=’Baltimore’ THEN PROVIDER_STATE=’MD’;
      ELSE IF PROVIDER_CITY=’Arlington’ THEN PROVIDER_STATE=’VA’;

      IF BILLTYPE = 13 THEN BILLTYPEDSC = ‘Hospital Outpatient’;
      ELSE IF BILLTYPE = 11 THEN BILLTYPEDSC = ‘Hospital Inpatient’;
      ELSE IF BILLTYPE = 33 THEN BILLTYPEDSC = ‘Home Health Agency’;

RUN;
```

Figure 6. DATA STEP with optimal placement of the IF statement.

In the code in Figure 6, the code highlighted in grey will be executed only if the IF statement is true. The code in grey contains two IF-THEN-ELSE blocks of conditional logic. The two blocks contain a total of 5 lines of code. When the IF statement is false, 5 fewer lines of code are executed. SAS uses CPU to perform conditional logic operations. When we avoid processing the 5 lines of code, we save processing time or CPU. Its optimal to place the IF statement at the top of the DATA STEP to avoid unnecessary processing.

USING INDEXES

As we’ve seen in the previous topics, the DATA STEP can be a resource-intensive and time-consuming construct to process data. By default, the DATA STEP performs a sequential and exhaustive read of an input SAS data set. Using different methods for subsetting data, such as choosing WHERE over IF is one way to speed up the processing of data. Another technique is to create an index on the data set, and then use the index to perform subsetting.

An Index is a file which is attached to the data set that allows you to directly access observations on the data set, and therefore bypass a sequential read and processing of the data set. It’s kind of like a map of the data set which contains observation numbers of primary key variables, which SAS uses to locate observations. You can create an index on a single variable, known as a simple index, or on multiple variables, a composite index.

You create an index using one of several methods; PROC DATASETS, PROC SQL, or the DATA STEP. Once the index is created on a data set, SAS stores it along with the data set. When you subset a SAS data set with an index, using WHERE or IF, SAS will decide whether or not to use the index to perform the subsetting. In general, SAS will use the index when the subset is relatively small in comparison to the size of the data set.

In order to demonstrate the power of indexes, I ran a few tests using a SAS data set containing over 4 million observations. In the example in Figure 7, I used a DATA STEP with a WHERE= data set option to subset the data set TESTING. An excerpt from the SAS log with the results of this example is provided in Figure 7.

```
402  Data Testing_Subset;
403    Set Testing
404      (Where=(STATUS='Administratively Converted'));
405  Run;

NOTE: There were 66836 observations read from the data set WORK.TESTING.
      WHERE STATUS='Administratively Converted';
NOTE: The data set WORK.TESTING_SUBSET has 66836 observations and 67
variables.
NOTE: DATA statement used (Total process time):
      real time           21.93 seconds
      cpu time            12.68 seconds
```

Figure 7. Subsetting SAS data set using WHERE= in DATA STEP.

The subset I'm producing contains 66386 observations, comprising approximately 1.9% of the input data set. The DATA STEP executed in only 22 seconds of real time. This test was run without an index, and the processing time was very short to begin with.

Nevertheless, I created an index on the TESTING data set to run a performance comparison test. As shown in the SAS log excerpt in Figure 8, I used PROC DATASETS and the MODIFY statement to define the index. Here I created a simple index on the variable STATUS.

```
412  Proc Datasets Library=Work;
413     Modify Testing;
414      Index Create STATUS / NOMISS;
NOTE: Simple index Status has been defined.
415  Quit;

NOTE: MODIFY was successful for WORK.TESTING.DATA.
NOTE: PROCEDURE DATASETS used (Total process time):
      real time           20.87 seconds
      cpu time            14.37 seconds
```

Figure 8. Using PROC DATASETS to create a simple index on TESTING data set

After creating the index, I re-ran the same example using a DATA STEP to extract a subset from the TESTING data set using the STATUS variable. In this example, the TESTING data set now has an index on the STATUS variable. Figure 9 shows the results of the test with an index in the SAS log.

```
Data Testing_Subset_ac;
  Set Testing
    (Where=(STATUS='Administratively Converted'));
Run;

NOTE: There were 66836 observations read from the data set WORK.TESTING.
      WHERE STATUS='Administratively Converted';
NOTE: The data set WORK.TESTING_SUBSET_AC has 66836 observations and 67
variables.
NOTE: DATA statement used (Total process time):
      real time           3.80 seconds
      cpu time            1.03 seconds
```

Figure 9. Subsetting SAS data set using WHERE= in DATA STEP

The results in Figure 9 demonstrate a performance improvement using an index. The DATA STEP executed in under 4 seconds of real time in comparison to 22 seconds without the index. This translates to a reduction in processing time of over 80%.

As we pointed out earlier in the paper, when an index is created on a data set, SAS will decide whether or not to use it to pull the subset. In general, SAS will use the index if the size of the subset is small relative to the full data set. The small size of our subset (1.9%) and the fast processing time indicate that SAS used the index to process it.

TESTING CODE

As a programmer, whether you're developing a new program from scratch, or you're modifying existing code, you need to test run your code to ensure its error and warning free. You might be running a program containing hundreds of lines of SAS code, perhaps even thousands. Consequently, the code might execute slowly and take very long to process.

To substantially reduce the amount of time to run the test, it's worthwhile to use a small sample of the data to run the test on. This is especially true if your data sets are large, in excess of millions of records.

To select a sample of the data, you can use the OBS= system option. With OBS you can specify or adjust the number of observations to be processed. In the example below is an OPTIONS statement containing the OBS= system option.

```
OPTIONS OBS=1000;
```

In the example, the sample was set at 1000 observations. Regardless of the size of your dataset, a subset of 1000 observations is small enough to efficiently test your code.

Alternately, you can set the number of observations so SAS will process your code without reading or writing any observations. As demonstrated below, you use OBS to set the number of observations to zero.

```
OPTIONS OBS=0;
```

Depending on the complexity of the code, a single test run might not be sufficient to confirm its validity. It's useful to run additional tests at different sample sizes or subsets.

Perhaps you'll run 3 tests and stagger them at increasing sample sizes. If you ran the first test on the sample of 1000 observations, then increase the sample and run your second test as below…

```
OPTIONS OBS=10000;
```

For the third test, you'll increase the sample size proportionately again to an appropriate level as below…

```
OPTIONS OBS=100000;
```

Along with the OBS option, you can use the FIRSTOBS option to direct SAS to begin reading a data set at a specified observation number. Using both options allows you to construct test cases in different segments of the data set, while limiting the sample size. For instance, if you can set the sample size at 10000 using OBS, and then read in the middle, and end of the data set, using FIRSTOBS.

The following examples assume a data set size of 1 million observations. For the first test case you won't need FIRSTOBS, since you're reading from the beginning. The second test case, reading from the middle would look like below…

```
OPTIONS FIRSTOBS=500000 OBS=10000;
```

The third test case, reading from the end, would look like the following…

```
OPTIONS FIRSTOBS=990000 OBS=10000;
```

Using the global system options, It's a good rule of thumb to specify FIRSTOBS and OBS at the start of your program, so it'll apply to all the steps in your code. Both FIRSTOBS and OBS can be specified as a data set option as well. Using them as data set options is appropriate for testing individual steps within your code.

Once you're done testing, and you want to reset the number of observations back to the default, you can use the OBS option again, as below.

```
OPTIONS OBS=MAX;
```

KEEP AND DROP STATEMENT AND OPTIONS

The KEEP and DROP statements and data set options are used to select variables from a SAS data set or an external file. Specifically, KEEP is used to include the variables which you want or need. DROP is used to remove or exclude the variables which you don't want. Both constructs can be used as a statement or a data set option. Furthermore, as a data set option they can be used as an input data set option or an output data set option.

The relative efficiency of the different forms of the method depend on which technique controls the reading of variables from a SAS data set, or the writing of variables to a SAS data set. The KEEP and DROP statement controls the writing of variables to the output SAS data set.

```
Data DeathsDemo2;
      Set DeathsDemo;
      Keep ALF Age Any_Cong_Setting Case County Date_of_Birth
           Date_of_Death Epi_Report_Date Ethnicity Facility_Name
           First_Name GH Intake_LTCF_NH Last_Name Location_Name
           Location_Type Notes Number Race Sex Staff_or_Resident
           Town ZCTA_D;
run;
```

Figure 10. DATA STEP using KEEP statement to select variables.

In Figure 10 above, a KEEP statement is used to select variables which will appear on the output SAS data set, DeathsDemo2. The results would be the same if we used the KEEP= output data set option to select the same set of variables on the DATA statement. Neither of these techniques is the most efficient because the full set of variables on the input data set is selected. Depending on how many variables it contains, this could be hundreds of variables.

```
Data DeathsDemo2;
      Set DeathsDemo
        (Keep=ALF Age Any_Cong_Setting Case County Date_of_Birth
             Date_of_Death Epi_Report_Date Ethnicity Facility_Name
             First_Name GH Intake_LTCF_NH Last_Name Location_Name
             Location_Type Notes Number Race Sex Staff_or_Resident
             Town ZCTA_D);
run;
```

Figure 11. DATA STEP using KEEP data set option to select variables.

In Figure 11 above, the same example from Figure 10 is revisited, except the KEEP= input data set option is used to select variables instead. This technique selects variables to be read from the input SAS data set. Only variables listed in KEEP= are read from the source data set and loaded into the program data vector (PDV).

This technique is more efficient because it allows us to read only necessary variables. It requires a lesser amount of input/output (I/O) operations to perform, and thus results in decreased input/output time. The reduction in input/output time can be translated into a reduction in processing time or CPU. KEEP= should be used to save I/O and CPU.

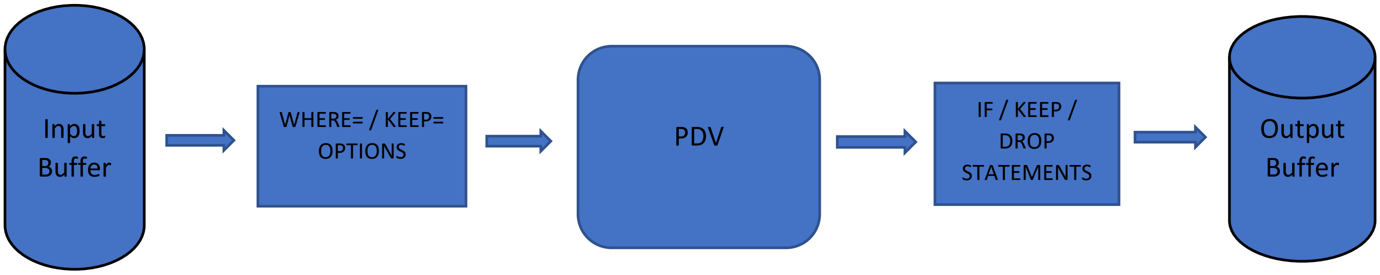

Figure 12. Data Flow Diagram of DATA STEP processing.

Figure 12 provides a flow chart of DATA STEP processing which illustrates the point during the DATA STEP loop where the subsetting techniques of WHERE, IF, KEEP, and DROP statements and options are applied. It shows that WHERE= and KEEP= are applied before the data is loaded to the PDV.

CONCATENATING DATA SETS

Concatenating data sets refers to the process of combining SAS data sets vertically. This is in contrast to the process of merging, which involves combining data sets horizontally. Concatenating is also referred to as appending or stacking data sets.

This process can be performed in the DATA STEP using the SET statement to read both data sets. It also can be performed using several SAS procedures. These procedures include PROC APPEND, PROC SQL, and PROC DATASETS.

In general, using the DATA STEP to read and process SAS data sets can be a resource-intensive construct to use. In the DATA STEP, using the SET statement, SAS data sets are read and processed sequentially. This means SAS reads observations in order from the top of the input data set one at a time. Then it cycles through the DATA STEP, and writes observations to the output data set. It continues until the last observation from the input data set is read.

If you use the DATA STEP to concatenate data sets, SAS will read all source data sets. To perform this operation, you specify all input data sets in the SET statement. When the code is executed, SAS will start with the first data set, and read each observation. Then it will continue to the second data set, and read each observation in the second data set. Because of all the reading and writing of data, SAS consumes substantial input/output (I/O), in addition to CPU or processing time.

The other technique, PROC APPEND is capable of combining two data sets, the base data set, and the data= data set. When executed, SAS skips over the base data set, and starts processing from the second data set, the data= data set. It simply takes observations from the data= data set and appends them to the end of the base data set. In this manner, It reduces the amount of data processing in relation to the DATA STEP.

In Figure 13 below, the process of appending is illustrated using both techniques; the DATA STEP and PROC APPEND. Highlighted in blue are the sets or subsets of data which are processed. Unlike the DATA STEP, with PROC APPEND, processing starts with the second data set, and no processing occurs with the first data set.

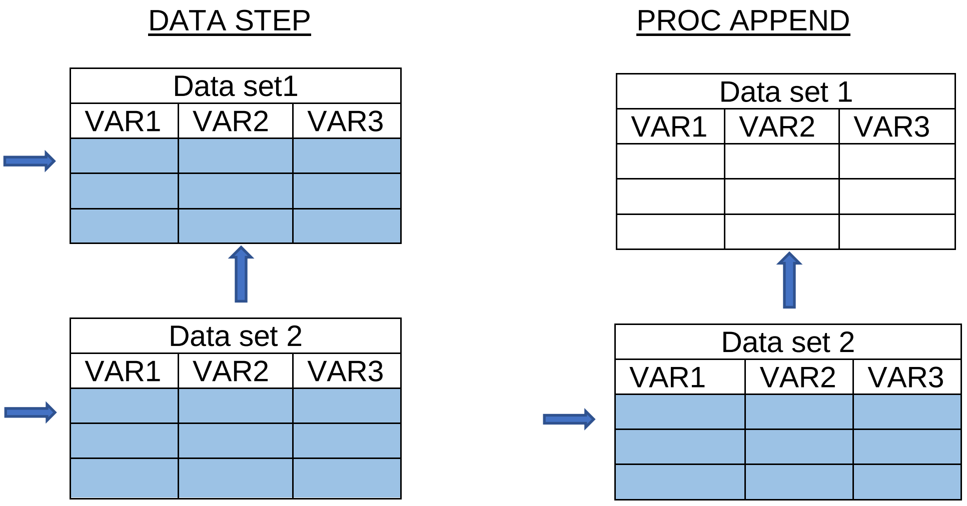

Figure 13. Comparison of DATA STEP with PROC APPEND in processing data sets.

This means that fewer resources are consumed using PROC APPEND. The only drawback to PROC APPEND is that its capable of combining a maximum of two data sets. With the DATA STEP, you can combine an unlimited number of data sets. However, this advantage doesn't have a direct impact on efficiency.

To illustrate the difference in efficiency, I ran a testcase using the DATA STEP and PROC APPEND, where I combine two data sets, each containing over 500,000 records. Figure 14 below contains an excerpt from the SAS log of the test run.

```
Data NDF_DSC;
 Set NDF_1994
     NDF_1995;
Run;

NOTE: There were 515155 observations read from the data set NDF_1994.
NOTE: There were 533420 observations read from the data set NDF_1995.
NOTE: DATA statement used (Total process time):
           real time   50.36 seconds
           cpu time    5.44 seconds

Proc Append Base=NDF_1994 Data=NDF_1995;
Run;

NOTE: Appending NDF_1995_AFTER to NDF_1994.
NOTE: There were 533420 observations read from the data set NDF_1995.
NOTE: 533420 observations added.
NOTE: PROCEDURE APPEND used (Total process time):
           real time    27.33 seconds
           cpu time     2.80 seconds
```

Figure 14. Example of DATA STEP and PROC APPEND concatenating data sets.

Using real time and CPU time as our performance metric, PROC APPEND performed better than the DATA STEP in this test case scenario. In real time, the DATA STEP executed in 50.36 seconds compared to 27.33 for PROC APPEND, a decrease of 46%. In CPU time, the DATA STEP executed in 5.44 seconds compared to 2.80 for PROC APPEND, a decrease of 48%.

From a conceptual and practical standpoint, PROC APPEND is the more efficient technique for concatenating SAS data sets. PROC APPEND is a best practice over the DATA STEP for combining SAS data sets vertically.

MERGING DATAETS

Merging SAS data sets is a fairly typical and routine data manipulation task which SAS programmers need to perform. Merging is also known as joining data sets or performing table look-ups. The process of Table lookups can be defined as looking up the value of one or more variables in another table based on the value of a common variable. In contrast to combining data sets vertically, known as concatenating data sets, the process of merging is defined as horizontally combining data sets.

The conventional technique for merging SAS data sets is the DATA STEP Merge. This usually requires sorting the input data sets using PROC SORT prior to merging. The DATA STEP merge is also known as the match merge and combines data from input data sets by combining records from each data set on a one to one basis. It reads one record from the first data set, then it reads one record from the second dataset and matches them based on the BY variable, and so on.

In Figure 15 below is an example of a DATA STEP merge using PROC SORT on both of the data sets, which includes timing information from the SAS log. I'm running the merge to perform a table lookup on a SAS data set, GDELT_ALL, containing 13.6 million observations. Some of the notes in the log have been excluded or re-organized.

```
Proc Sort Data=Gdelt.Gdelt_All Out=Gdelt_All;
      By EventCode;
Proc Sort Data=Gdelt.Cameo_Event_Codes Out=Cameo_event_codes Nodupkey;
      By CameoEventCode;
Run;

NOTE: PROCEDURE SORT used (Total process time):
      real time           14.31 seconds
      cpu time            12.60 seconds

Data Gdelt_All_V2;
  Merge Gdelt_All(IN=A)
        Cameo_event_codes(Rename=(CameoEventCode=EventCode) IN=B);
   By EventCode;
  If A and B;
Run;

NOTE: DATA statement used (Total process time):
      real time           37.56 seconds
      cpu time            4.57 seconds
```

Figure 15. Example of DATA STEP Merge to combine data sets.

In real time, PROC SORT took 14.31 seconds, and the DATA STEP merge took 37.56 seconds. The combined processing time of PROC SORT and the DATA STEP merge is about 52 seconds. In general, the processing time isn't very long, but for relative performance we need to examine other techniques.

An alternative to the DATA STEP merge is the PROC SQL Join. The PROC SQL join uses a different process to combine data horizontally. It executes what's called a cartesian product. The cartesian product is where it combines each value in one data set with every value in the other data set. The result set includes all possible permutations. Then it subsets the cartesian product based on the WHERE or ON join condition. Before it joins them, PROC SQL sorts its data sets implicitly, and therefore doesn't require PROC SORT to be run on each data set.

To assess comparative performance, I re-ran the same table lookup example from Figure 15, using a PROC SQL join. Figure 16 contains the SAS log from the test using PROC SQL join. Notice that since it only requires one step the PROC SQL join coding is more compact.

```
Proc Sql;
    Create Table gdelt_All_V2 as
    Select A.*, B.EventDescription
    From Gdelt.Gdelt_All as A, Gdelt.Cameo_Event_Codes as B
    Where A.EventCode=B.CameoEventCode;
Quit;

NOTE: PROCEDURE SQL used (Total process time):
       real time           14.56 seconds
       cpu time            6.46 seconds
```

Figure 16. Example of PROC SQL Join to combine data sets.

The PROC SQL join executed in 14.56 seconds of real time. This is a dramatic reduction in processing time from the DATA STEP merge results. It also ran in less than half the CPU time of the DATA STEP merge.

Because it avoids PROC SORT by sorting the data sets implicitly, a PROC SQL join should be a more efficient alternative to the DATA STEP merge. Out performance tests support this finding. Although, the results may vary depending on the platform used, the structure of your data, and the resources available at your site.

Although not examined in this paper, a HASH TABLE is an advanced table lookup technique which provides another efficient alternative to the DATA STEP merge. The HASH table utilizes in-memory processing, rather than disk-based processing which substantially speeds-up processing. For these reasons, the PROC SQL join and the HASH table are a best practice for performing table lookups.

MINIMIZING THE NUMBER OF PASSES THROUGH THE DATA

When you use the DATA STEP to perform data manipulation, computing resources are consumed. To reiterate, the DATA STEP reads data sequentially from an input data set, and writes data to an output data set. This process requires input/output or I/O to read and write the data, as well as CPU, another useful resource.

This is especially the case if you're referencing large data sets in excess of millions of records. The larger the volume of data you're working with, the more resources which are expended in processing it. A recommended best practice is to perform your work with fewer DATA STEPS or to use other constructs besides the DATA STEP.

Typically, there are many programming tasks which are performed in separate DATA STEPS rather than consolidating these tasks into one DATA STEP. For example, a programmer may need to rename variables on a data set, or drop specific variables from a data set. A separate DATA STEP is created solely for performing these tasks of renaming and dropping variables. However, a separate DATA STEP is not needed. These tasks can be performed in the previous DATA STEP.

In the example in Figure 17 below there are two DATA STEPS. In the first DATA STEP, a DATA STEP merge is performed, and a variable MATCH is created to contain the results of the merge. In the 2nd DATA STEP, the variable MATCH and a few other variables are used to produce several output data sets.

```
data ridoc4;
     merge SF1 (in=a) LL1 (in=b);
     by IDL_FirstName IDL_LastName IDL_DOB collection_date;
     format match $50.;
     if a and b then match="Both";
     if a and not b then match="SF Only";
     if b and not a then match="Linelist Only";
run;

data ridoc5 discrep SF2 LL2;
     set ridoc4;
     If match="Both" AND ((IDL_CF_RorE in ("Resident","Unknown") AND
          LL_RorE="Resident") OR IDL_CF_RorE in
          ("Employee","Unknown") AND LL_RorE="Employee")) then
          output ridoc5;
     else if match="Both" then output discrep;
     if match="SF Only" then output SF2;
     if match="Linelist Only" then output LL2;
run;
```

Figure 17. DATA STEP merge with two DATA STEPS.

The more DATA STEPS which you include in your program, the more temporary or WORK data sets which are created. Additional copies of data sets are stored on your hard drive or on disk and take up storage space. Storage space may be a limited resource, and your system may not have enough space for these data sets. By consolidating DATA STEPS in your code, and reducing the number of data sets which are created, you can save valuable storage space.

In Figure 18, is the same example presented in Figure 17. Figure 18 features one DATA STEP instead of the two steps in Figure 17. The two steps in Figure 17 have been consolidated into one step in Figure 18. What took two steps to accomplish in Figure 17 is accomplished in one step in Figure 18.

```
Data ridoc4 discrep SF2 LL2;
   Merge SF1 (in=a) LL1 (in=b);

     By IDL_FirstName IDL_LastName IDL_DOB collection_date;

    If A and B and ((IDL_CF_RorE in ("Resident","Unknown") and
       LL_RorE="Resident") Or (IDL_CF_RorE in ("Employee","Unknown")
       AND LL_RorE="Employee")) then Output ridoc4;
    Else if A and B then Output discrep;

    If A and not B then output SF2;
    If B and not A then output LL2;
Run;
```

Figure 18. DATA STEP Merge in one consolidated step.

The DATA STEP in Figure 18 creates the same 4 data sets which were created by two steps in Figure 17. In Figure 18, with only one DATA STEP, the reading and writing of data is performed once. In Figure 18, with both DATA STEPS, the data is processed, read and written twice. By consolidating the steps we've reduced the amount of processing, and read and write operations which are required. Consequently, valuable computing resources such as CPU and I/O are saved.

In the first example, two data sets, RIDOC4 and RIDOC5 were created. In Figure 18, only RIDOC4 is created. Since fewer SAS data sets are created, the amount of hard drive or disk space required to store these data sets is less. By minimizing the passes through the data we've also consumed less storage space, another important computing resource. An alternate technique to manage storage space is to use PROC DATASETS to delete unnecessary temporary data sets.

As a side benefit, the code in Figure 18 is more compact, and easier to read. With fewer lines of code, the program is easier to maintain. For these reasons, it's a best practice to reduce the number of passes through the data by limiting the number of DATA STEPS in your code.

DATA SET COMPRESSION

Another technique to manage storage space is data set compression. Data set compression reduces the size of your data set by compressing empty space within it. Compression can reduce the size of your data set by as much as 80 or 90%. Data sets which make good candidates for compression usually contain variables with a lot of missing values.

There are different types of compression which can be applied to your data set. There is BINARY compression as well as CHAR compression. To invoke compression, you can use the COMPRESS= global system option as in the example below.

OPTIONS COMPRESS=BINARY;

Compression can also be applied to a data set on an individual basis using COMPRESS= as a data set option. To turn off compression, you use the COMPRESS= option again in the following example.

OPTIONS COMPRESS=NO;

One of the disadvantages of data set compression is the overhead required to read compressed data sets. The overhead increases overall processing time. SAS is able to determine whether using compression will result in a decrease in data set size. If using compression results in a decrease in size, SAS will not apply compression to a data set.

CONCLUSION

In many environments, SAS programmers face constant pressure to produce results, and to produce them as quick as possible. Knowledge of efficient programming techniques makes a difference in reducing processing time, minimizing storage space requirements, or otherwise streamlining your SAS code. Many programmers don't possess knowledge of efficiency techniques. You can measure the difference applying these techniques makes by using resource metrics such as CPU time, memory, and storage space. For the above reasons, It's important to utilize these best practices in your SAS code, and to train others on them.

REFERENCES

Iyengar, Jay, and Horstman, Joshua M. "Look Up Not Down: Advanced Table Lookups in Base SAS" Proceeding of the 2020 Southeast SAS User Group Conference. https://www.lexjansen.com/sesug/2020/SESUG2020_Paper_150_Final_PDF.pdf

Iyengar, Jay. "If you need these OBS and these VARS, then drop IF, and keep WHERE". Proceedings of the 2017 Midwest SAS User Group Conference. https://www.mwsug.org/proceedings/2017/SA/MWSUG-2017-SA02.pdf

Iyengar, Jay. "Tips, Traps, and Techniques in BASE SAS for vertically combining SAS data sets". Proceedings of the 2018 Midwest SAS User Group Conference. https://www.lexjansen.com/mwsug/2018/SP/MWSUG-2018-SP-76.pdf

Lafler, Kirk P. "Top Ten SAS® Performance Tuning Techniques." Proceeding of SAS Global Forum 2012 Conference. https://support.sas.com/resources/papers/proceedings12/357-2012.pdf

Lafler, Kirk P. "SAS® Performance Tuning Strategies and Techniques" Proceedings of the 2009 Midwest SAS Users Group Conference. https://www.lexjansen.com/mwsug/2009/how/MWSUG-2009-H06.pdf

SAS Institute Inc., 2010. *SAS® Certification Prep Guide – Advanced Programming for SAS® 9*. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The author would like to thank Tasha Chapman, WUSS 2022 Operations Chair, Lisa Mendez, WUSS 2022 Academic Chair, Kirby Sinclair, Programming Section Chair, and the WUSS Executive Committee and Conference Team for accepting my abstract and paper and for organizing this great conference.

CONTACT INFORMATION

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

Jay Iyengar
Data Systems Consultants LLC
datasyscon@gmail.com
https://www.linkedin.com/in/datasysconsult/

Jay Iyengar is director of Data Systems Consultants LLC. He is a SAS consultant, trainer, and SAS Certified Advanced Programmer. He's been co-leader and organizer of the Chicago SAS Users Group (WCSUG) for the last 5 years. He's presented papers and training seminars at SAS Global Forum (SGF), Midwest SAS Users Group (MWSUG), Wisconsin Illinois SAS Users Group (WIILSU), Northeast SAS Users Group (NESUG), and Southeast SAS Users Group (SESUG) conferences. He has been using SAS since 1997. His industry experience includes International Trade, Health care, Database Marketing and Educational Testing.

TRADEMARK CITATION

APPENDIX I

```
347  Data TESTING2;
348     Set TESTING;
349
350      Age = Round((Today()-BirthDate_Text__c)/365.25, 1);
351
352      If _N_<=20 Then Put Age=;
353
354      Length Age_Group $8;
355
356                  /* Create age group variables for Age Cohorts */
357      If 0<=Age<=9 Then age_group="0-9";
358      If 10<=Age<=19 Then age_group="10-19";
359      If 20<=Age<=29 Then age_group="20-29";
360      If 30<=Age<=39 Then age_group="30-39";
361      If 40<=Age<=49 Then age_group="40-49";
362      If 50<=Age<=59 Then age_group="50-59";
363      If 60<=Age<=69 Then age_group="60-69";
364      If 70<=Age<=79 Then age_group="70-79";
365      If 80<=Age<=89 Then age_group="80-89";
366      If 90<=Age<=99 Then age_group="90-99";
367      If Age>=100 Then age_group="100+";
368      If Age=. Then age_group="zPending further info";
369
370  Run;

Age=18
Age=71
Age=62
Age=4
Age=50
Age=10
Age=40
Age=3
Age=9
Age=16
Age=13
Age=9
Age=11
Age=11
Age=14
Age=39
Age=9
Age=17
Age=8
Age=23
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).
      409 at 350:11   409 at 350:25   409 at 350:44
```

```
NOTE: There were 4116153 observations read from the data set WORK.TESTING.
NOTE: The data set WORK.TESTING2 has 4116153 observations and 69
variables.
NOTE: Compressing data set WORK.TESTING2 decreased size by 75.66 percent.
      Compressed is 24434 pages; un-compressed would require 100395 pages.
NOTE: DATA statement used (Total process time):
      real time           46.21 seconds
      cpu time            38.54 seconds



372  Data TESTING2;
373    Set TESTING;
374
375      Age = Round((Today()-BirthDate_Text__c)/365.25, 1);
376
377      If _N_<=20 Then Put Age=;
378
379      Length Age_Group $8;
380
381                  /* Create age group variables for different reports
*/
382      If 0<=Age<=9 Then age_group="0-9";
383      Else If 10<=Age<=19 Then age_group="10-19";
384      Else If 20<=Age<=29 Then age_group="20-29";
385      Else If 30<=Age<=39 Then age_group="30-39";
386      Else If 40<=Age<=49 Then age_group="40-49";
387      Else If 50<=Age<=59 Then age_group="50-59";
388      Else If 60<=Age<=69 Then age_group="60-69";
389      Else If 70<=Age<=79 Then age_group="70-79";
390      Else If 80<=Age<=89 Then age_group="80-89";
391      Else If 90<=Age<=99 Then age_group="90-99";
392      Else If Age>=100 Then age_group="100+";
393      Else If Age=. Then age_group="zPending further info";
394
395
396  Run;

Age=18
Age=71
Age=62
Age=4
Age=50
Age=10
Age=40
Age=3
Age=9
Age=16
Age=13
Age=9
Age=11
Age=11
```

```
Age=14
Age=39
Age=9
Age=17
Age=8
Age=23
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).
      409 at 375:11   409 at 375:25   409 at 375:44
NOTE: There were 4116153 observations read from the data set WORK.TESTING.
NOTE: The data set WORK.TESTING2 has 4116153 observations and 69
variables.
NOTE: Compressing data set WORK.TESTING2 decreased size by 75.66 percent.
      Compressed is 24434 pages; un-compressed would require 100395 pages.
NOTE: DATA statement used (Total process time):
      real time           48.27 seconds
      cpu time            40.03 seconds
```