Efficiency: How Your Data Structure Can Help or Hurt!!!

Toby Dunn, AMEDDC&S, Fort Sam Houston, Tx.

Abstract

Once one gets past the simplistic idea that efficiency is only about the speed at which a program runs, a whole new world opens up. Too many SAS® programmers fall into the trap of thinking that they must write program code based on the data structure they are provided.

This paper introduces the paradigm shift that instead of making SAS fit the data structure, the data structure should be made to fit SAS. One of the most striking aspects is how important the data structure becomes to the overall efficiency; i.e. quality, readability, and maintainability. We will cover some of the basic topics of data structure and how it affects the overall look, complexity, maintainability, and speed of a SAS program. Starting with the basics of what constitutes efficiency, we will review common data structure mistakes and how they adversely affect the overall efficiency of the code. Finally, examples will demonstrate how simple and not-so-simple fixes improve the overall efficiency.

Introduction

What exactly constitutes efficiency? The speed at which a program runs, the simplicity and compactness of the code, how easy the code can be maintained, or is it the ease at which the code can be changed as the requirements change? Personally, I like to think that efficiency isn't any single one of these but rather the totality of these balanced against each other. Sadly though, many people see speed as the only measure of a program's efficiency. This is unfortunate, since the run time makes up a very small portion of the total time in the life cycle of a program.

Speed has come to be synonymous with efficiency in programming; this isn't a surprise given its origins. In the early days of computer programming, computers where extremely slow compared to today's standards. As such, the early computer scientist and programmers looked for any way they could to shorten run times. One of the solutions they found was to tweak the algorithms and this unfortunately became the resulting unit of measure to compare programs. However, something else resulted which was the study of data structures and how changing a data structure impacts the overall efficiency of a program. What they discovered was that not only could changing the data structure speed up a program, it could also reduce the time to write, maintain, and change it.

Data structures and their impact on the overall efficiency of a program is the topic of this paper. In general, optimal data structure will be discussed, and examples will illustrate the difference between how a poor data structure can make the code more complex and less flexible as well as how a good data structure can make coding as easy as breathing. Along the way we will explore some common problems that can occur when a data structure isn't optimal. Finally, we will investigate ways to change the data structure to a form better suited to optimizing the efficiency of a program.

If the data structure can impact the overall efficiency of a program, then just what is the optimal data structure? To answer that question, one needs to look at the underlying algorithms of the language used. The reason that data structure has such a large impact on efficiency is that it works with the functionality of the programming language. In the case of SAS, a narrow but skinny data structure is the most efficient. To illustrate, you may choose to run the following code and see for yourself:

```
Data Narrow ;
Do I = 1 To 1000000 ;
 Output ;
End ;
Run ;

Data Narrow ;
Set Narrow ;
Run ;


Data Wide ;
Array ABC { 1000000 } $ ;
Retain ABC: 'A' ;
Run ;

Data Wide ;
Set Wide ;
Run ;
```

Yikes! I decided to terminate the data step because I didn't have the time to wait for the Wide data steps to be created much less actually ran. Okay, so now it is apparent that SAS prefers skinny and long data structures. Luckily for us, old programmers already figured this out. After careful study they found a few data structures that ensure optimal overall performance. These structures are called Normal Forms. There are a handful of normal forms ranging from First Normal Form to Sixth Normal Form as well as the Boyce-Codd Normal Form and Domain/Key Normal Form. The First Normal Form is the easiest to learn and the one that will likely serve best for most SAS programming purposes. However, I do encourage the reader to investigate all the normal forms and their advantages.

A Data Structure in First Normal Form has at least three attributes. The first rule is there are no rows with duplicate information. That is to say that no rows should have all variables with the exact same values. If the number of duplicate observations has meaning, then consolidate all duplicate observations and add a variable to store the number of duplicate observations. In short, duplicate rows of data do nothing more than consume storage space and cause the programmer to dedup or aggregate the file each time, wasting CPU time and memory.

Example1.)

VEHICLETYPE	MODEL	MAKE	YEAR	COLOR
Truck	1500	Chevy	2008	Blue
Truck	1500	Chevy	2008	Blue

should be reduced to:

VEHICLETYPE	MODEL	MAKE	YEAR	COLOR	COUNT
Truck	1500	Chevy	2008	Blue	2

The second rule for a data set to be in First Normal Form is that each variable must be atomic. This simply means that each variable must contain one and only one value. Imagine trying to determine types of truck packages sold from a variable that is non-atomic…all I can say it woe to the programmer who has do this….

VEHICLETYPE	MODEL	MAKE	YEAR	COLOR	PACKAGE
Truck	1500	Chevy	2008	Red	Sports
Truck	1500	Chevy	2008	Blue	Sports, Standard
Truck	1500	Chevy	2008	Gold	Sports, Sports, Standard

should be restructured as:

VEHICLETYPE	MODEL	MAKE	YEAR	COLOR	NUMSOLD	PACKAGE
Truck	1500	Chevy	2008	Red	1	Sports
Truck	1500	Chevy	2008	Blue	1	Sports
Truck	1500	Chevy	2008	Blue	1	Standard
Truck	1500	Chevy	2008	Gold	2	Sports
Truck	1500	Chevy	2008	Gold	1	Standard

Lastly, to be in First Normal Form, there can not be multiple variables to represent the same information.

VEHICLETYPE	MODEL	MAKE	YEAR	RED	BLUE	GOLD
Truck	1500	Chevy	2008	1	2	3

To correct this, the color and number of vehicles sold should each be given its own variable:

VEHICLETYPE	MODEL	MAKE	YEAR	COLOR	NUMSOLD
Truck	1500	Chevy	2008	Red	1
Truck	1500	Chevy	2008	Blue	2
Truck	1500	Chevy	2008	Gold	3

This is the hardest concept for most SAS programmers to understand simply because it doesn't conform to the way that humans think about data. Humans like to see data spread out longitudinally, which conforms to the way we think and process data. A perfect example is how an Excel spreadsheet propagates data longitudinally. However, computers do not analyze data this way, nor are the underlying algorithms or tools in SAS built to efficiently handle data that is in a wide structure. Even if one knows the maximum number of colors it makes it difficult to solve even some of the simplest of problems. To illustrate, use the wide data structure and try to find all the models of trucks sold that have the same color.

What Shouldn't Be In A Variable Name

Now that we have discussed how a Normalized data set is structured, let us review a few code examples to illustrate the difference between a bad data structure and a normalized one. One of the most common problems in data structure is variable names that really should be variable values in the name. The following example was selected from the SAS Listserve. While names have been stripped and a few grammatical corrections made, the question is left as submitted. This is a real problem from a real person.

I have a question using an array to sum up variables. Suppose I have a data set like this:

```
Data A ;
Input X1 X2 X3 X4 Y1 Y2 Y3 Y4 ;
Cards ;
1 2 3 4 1 2 3 4
2 3 4 5 2 3 4 5
3 4 5 6 3 4 5 6
;
Run ;
```

I want to create new variables so that t1=x1+y1, t2=x2+y2, t3=x3+y3 and t4=x4+y4. I know how to do this manually for a small dataset with a small number of fields, but I have a large data set with hundreds of fields. My question is how to do this using SAS array processing (especially using a multidimensional array).

Now let's review what the code would look like if we use arrays to solve this problem.

```
Data Need1 ( Drop = I ) ;
Set A ;
Array X     { * } X: ;
Array Y     { * } Y: ;
Array Total {4 } ;

Do I = 1 To Dim( X ) ;
 Total( I ) = X( I ) + Y( I ) ;
End ;

Run ;
```

At first glance the provided solution looks like simple code, right? Well, it really isn't all that great. Yes, it does solve the posters problem but does so at a cost. Notice that the Total array has to have the number of elements hard coded. This makes for problems if one has to rerun this code often and the number of required elements changes. In other words, the array may need to be changed when a new or updated data set is used and this will have to be mined from the data beforehand. Along these same lines, the do-loop has to loop through all those elements. While this isn't a big issue if the number of elements is relatively small (as in the example), it does significantly increase the time to run the code if the number of elements increases.

Wouldn't it be nice if we could create code that could do the same thing that could be written once and would grow or shrink with the data? Well, consider the same data, except that now there are only two variables, X and Y.

```
Data Need1 ;
Set A ;

Total = X + Y ;

Run ;
```

Now the code concentrates on the business logic and nothing more. It goes from 9 lines of code to 5, and eliminates do-loops and arrays which need to be maintained.

Eliminate Macros With Data Structure

Poor data structure is one of the major causes of programmers writing macros which are not needed. Macros are harder to write and can potentially lead to hours of frustration due to the extra time necessary to write, debug, and maintain. It stands to reason that minimizing the use of macros to the proper situations can sometimes be a smart choice.

To illustrate, I have two temporary datasets for programs base2000 and donor2000. One problem is the macro works fine until Y resolves to 10 when X resolves to 9. The run before Y resolves to 10 generates a data set base2009 instead of base209.

```
*Creating 2001-2009 projections;
%macro whatever;
%do y = 1 %to 20;
%let x = %eval(&y - 1);

%if &y <= 9 %then %do;
      data base200&y.;
      set base200&x.;
      agenew=age+&y;
      run;

      data donor200&y.;
      set donor200&x.;
      agenew=age; run;

      data base200&y.;
      merge donor200&y. base200&y.(in=a);
      by agenew racesex;
      if a;
      drop agenew;
      run;

      data base200&y; set base200&y;
      if rannum le rate then delete;
      run;
%end;

*create 2010 to 2020 projection*;
%if &y >= 10 %then %do;
            data base20&y.;
            set base20&x.;
            agenew=age+&y;
            run;
            data donor20&y.;
            set donor20&x.;run;

            data base20&y.;
            merge donor20&y. base20&y.(in=a);
            by agenew racesex;
            if a;
            drop agenew;run;

            data base20&y.; set base20&y.;
            if rannum le rate then delete;
            run;
  %end;
%end;

%mend whatever;

%whatever;
options mprint symbolgen;
```

This macro is in need of serious help. A good rule of thumb to follow whether using macro or non macro code is that if the code becomes so complicated and so intricate it is hard to understand, then perhaps it is time for a new approach. Notice that all the preceding macro accomplishes is creating several different data sets from two source files. It only makes one small change to Age in the Base data steps. The problem with this data structure is that it isn't complete, and if one did follow the method shown then several little data sets would be created. Having to process these data sets over and over again for each process that needs to be completed is inefficient on many levels. Instead, consider fixing the Base data set to include everything needed to solve the problem:

```
Data Base ( Drop = I ) ;
Set Base2000 ;

Do I = 1 To 20 ;
 NewAge = Age + I ;
 Year   = 2000 + I ;
 Output ;
End ;

Run ;
```

```
Proc SQL ;
Create Table Master ( Drop = NewAge ) As
Select Base.* , Donor.Var1 , ..... , Donor.VarN
  From Base As Base
        Left Join
        Donor As Donor
        On  ( Base.NewAge  = Donor.Age     )
        And ( Base.RaceSex = Donor.RaceSex )
        And ( RanNum > Rate ) ;
Quit ;
```

To Normalize Or Not To Normalize

At this point, many readers may comment “but the data I have isn’t usually normalized”. While we cannot always dictate the structure of the data we receive we are not bound to keep a bad data structure. Think of it another way, even if you can’t restructure the data and save it as a permanent data file there is no rule against restructuring it and holding as a temporary file. Since restructuring a data set takes time and often quick results are required, just when should one restructure the data and when should one just deal with the really bad structure?

If it takes more time for you to restructure the data set rather than just write code to handle the structure then it is probably a good idea to leave it alone. Now this isn’t a blanket rule because you have to consider if others are using this same data set, how often the data set is used, what type of reports are needed and may be needed in the future. As these different requirements increase, restructuring the data becomes more important.

Restructuring Your Data

In general, the two methods used to restructure data from a denormalized form to a normalized form utilize the Data Step and Proc Transpose. Each has their time and place and it is important to know when to use one over the other. Keep in mind that as you start to normalize your data sets it takes time to learn how to handle weird odd ball cases. While at first it may take you some time to learn how to properly normalize a Data Set, the more you use this process the easier it becomes. Eventually you may find that you detest working with a denormalized Data Set.

Suppose you have a data structure such as:

Name	Age	Score1	Score2	Score3
Alice	13	6.5	7.2	7.8

This data structure needs to be:

Name	Age	Score
Alice	13	6.5
Alice	13	7.2
Alice	13	7.8

Applicable Data Step Code:

```
Data Need ;
Set Have ;
Array Score { * } Score: ;

Do I = 1 To Dim( Score ) ;
   Score = Score( I ) ;
   Output ;
End ;

Run ;
```

Applicable Proc Transpose code:

```
Proc TransPose
  Data = Have
   Out = Need ( Drop = _Name_ Rename = ( Col1 = Score ) ) ;
  By Name Age ;
Run ;
```

One of the problems with using Proc Transpose is that it requires some sort of Primary Id to transpose data. Consider the following data structure:

```
Data Have ;
Infile Cards ;
Input X1 X2 Y1 Y2 ;
Cards ;
1 2 3 4
6 7 8 9
;
Run ;


Proc TransPose
  Data = Have
   Out = Need ;
  Var X: Y: ;
Run ;



Obs   _NAME_   COL1   COL2
  1     X1        1      6
  2     X2        2      7
  3     Y1        3      8
  4     Y2        4      9
```

To solve this problem one needs to artificially add a unique row counter to each observation. The most efficient way to do this is through the use of a Data Step view.

```
Data Temp / View = Temp ;
Set Have ;
ID + 1 ;
Run ;


Proc TransPose
  Data = Temp
   Out = Need ( Drop = ID ) ;
  By ID ;
  Var X: Y: ;
Run ;


Obs   _NAME_   COL1
   1     X1       1
   2     X2       2
   3     Y1       3
  4       Y2            4
  5       X1            6
  6       X2            7
  7       Y1            8
  8       Y2            9
```

Now before you start thinking that the Data Step is the way to go for every possible situation, consider the same code normalized using a Data Step.

```
Data Need1 ( Keep = Col1 _Name_ ) ;
Set Have ;
Array X { * } X: ;
Array Y { * } Y: ;

Do I = 1 To Dim( X ) ;
  Col1   = X( I ) ;
  _Name_ = VName( X( I ) ) ;
  Output ;
End ;

Do I = 1 To Dim( Y ) ;
  Col1   = Y( I ) ;
  _Name_ = VName( X( I ) ) ;
  Output ;
End ;

Run ;
```

Even with the extra Data Step view the Proc Transpose code is shorter since it doesn't have to add N number of Do loops to encompass each set of variables. However, depending on how denormalized the data structure is, the Proc Transpose code can be as complicated and require more steps to change the data to a normalized state.

Many people never normalize their data simply because the end report will need to be denormalized. While the need for the data to be denormalized for reporting purposes is valid, this doesn't mean that one should denormalize the data set. SAS has a few procedures that handle reporting and normalized data structures. Okay, well almost all of the procedures in SAS love normalized data structures. Proc Tabulate and Report are just two of them and they are designed to produce reports.

```
Listserv sample question:\
```

I have a dataset and I need to use PROC REPORT to produce a cross tab report for the data shown below. Please advise.

```
data person;
   infile datalines delimiter=',';
   input product $ GRADE $;
   datalines;
HAF,B
HAF,H
HAF,U
DAF,B
DAF,H
DAF,U
HAF,B
HJA,U
JFK,H
JFK,U
DAF,B
DAF,H
HJA,U
;
RUN;
```

THE RESULTS need to look like

```
product       GRADE    Total
            B   H    U
DAF         2   2    1      5
HAF         2   1    1      4
HJA         0   0   2      2
JFK        0   1   1      2
Total       4  4     5     13
```

Solution:

```
Options Missing='0' ;

Proc Tabulate
  Data = Person ;
  Class Product Grade ;
  Table Product='' All='Total' , Grade*N=''*F=8. All='Total'*N=''*F=8.
         / Box = 'Product' ;
Run ;
```

As shown in the above example, SAS procedures like a normalized data structure. In fact, the use of normalized data structures is ingrained in the very fabric of the SAS language. For example, consider By and Class statements. These statements exist to help the SAS programmer handle normalized data structures. These features allow the programmer to forget about specific values and allow SAS to handle the specifics so the programmer can handle the business logic. Using By or Class statements in a procedure, like the one used in the above example, allows the data to drive the reports created. If the data this month comes in with 15 groups then 15 reports are created, if the data comes in with only 5 reports then 5 reports are created. No need for the programmer to mess with these details in some complex and convoluted macro. Let SAS handle the hard jobs it was designed for so you can concentrate on the business logic and finding the best approach to solving the problem at hand.

Eliminating Variables

SAS is an I/O bound language, that means that it brings in one observation of information, processes that observation, and then writes out. The physical act of porting information into memory and then writing it out to disk requires clock time. Needless to say the less information that SAS has to bring in and then write out will speed your program up considerably. I can not stress enough that if you don't absolutely need a variable then drop it out of the data set. If you don't believe me then test your data sets to prove this statement for yourself. As an example, on a recent project I dropped around 200 or so variables that weren't needed. I was matching a little of over 87 thousand observations to a data set with approximately 1 million observations. By dropping these unneeded variables the time it took to match these two data sets was reduced by 41 minutes. Now that is bang for your buck.

Conclusions

We have only scratched the surface of data structures and their implications to SAS code. I highly recommend that each and every programmer go out and further study the different types of Normal forms and how this affects their code. Normalizing a data set as early as possible simplifies the code, making the code easier to maintain and debug as well as simplifying the developing process. It allows the business logic to clearly be expressed through the code and makes hard coding values almost a thing of the past. Lastly, when it comes to reporting, it allows the data to drive not only the program but the reports. Remember that while your data may not come to you in the best form for processing, it doesn't mean that you can't reshape it to fit SAS.

References

The Power Of The By Statement
Toby Dunn And Paul Choate , SGF 2007
www2.sas.com/proceedings/forum2007/222-2007.pdf

SAS 9.1.3 XP Platform
SAS Institute Inc., Cary, NC

SAS OnlineDoc 9.1.3 for the Web
SAS Institute Inc., Cary, NC

Recommended Reading

SAS-L@LISTSERV.UGA.EDU
http://listserv.uga.edu/archives/sas-l.html

Documentation for SAS Products and Solutions
http://support.sas.com/documentation/onlinedoc/index.html

Special Thanks:

To Paul St Louis, for his patience and awesome ability to make sense of my chicken scratch. All the great people on the SAS Listserve (SAS-L) who ask such great question and those who come up with all those great solutions.

Contact Information

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

Toby Dunn,

AMEDDC&S, Fort Sam Houston, TX.

toby.dunn@amedd.army.mil

San Antonio. TX
E-mail: tobydunn@hotmail.com

Join the SAS-L! @ listserv.uga.edu or Google Groups