Paper 119--2010

Stuff We All Do: Mistakes We've Made That You Won't Have To

Joe Novotny, dunnhumby USA, Cincinnati, OH
Michael Bramley, dunnhumby USA, Cincinnati, OH

ABSTRACT
Ever felt alone? Like you're the only SAS® Programmer up until 3am looking for that missing %end statement in a septuplly-nested macro? Fear not, there are at least two other Programmers out there just like you (and we venture to say thousands more who just won't admit it). This paper helps to answer some questions you may (or should) have been asking but were too embarrassed to ask. Specifically, we delve into the depths of 1) How to keep DATA step execution under a dozen hours by appropriately setting the BUFSIZE and BUFNO options, 2) How to remain friends with your system administrator by not executing seemingly innocent PROC SQL code which consumes all your system resources, 3) How to keep your sanity by understanding when you need to force a step boundary, 4) How to keep your job by correctly creating summary variables outside vs. inside of an iterating macro do loop, 5) How to speed up your SAS jobs by not trying to download the entire Oracle data table to your local environment. Learning all these (plus a few additional techniques) will keep you employed and in good graces with the systems gurus at your company.

IN THE BEGINNING
It is not the 11th hour. If it were the 11th hour, you would know that your work is unsalvageable and you will need to reset expectations on when your project can be delivered. No, you recognize the problem with enough time that if you work 18 hrs a day for the coming four days, you think you have a better than 50/50 shot at delivering your project on time. Your heart sinks. If only you had seen this issue sooner. If only your code ran faster so rerunning everything wouldn't be such a huge issue.

If only…

Scenes like this play themselves out all the time. Sometimes, if you're lucky (or reading this paper), you are only a witness. But sometimes you are the unfortunate SAS Programmer who is directly affected. The goal, with a plethora of processes and a cornucopia of QA, is to catch issues before projects are delivered. And the authors assure you, gentle reader, that no projects were harmed in the writing of this paper. However, learning from mistakes is part of Life As A Programmer. To use a favorite climbing metaphor, "if you don't fall once in a while, you're not trying hard enough". But it is how we learn from our mistakes that differentiates the bad from the good. And it is how we learn from others' mistakes that differentiates the good from the great. It is never a good time to have to rerun thousands of lines of code because of a "simple" mistake made early in your software bank. This paper springs out of several "situations" the authors have seen (or caused). We hope, through our learning, you will be able to avoid these types of situations in the future too. Specifically, we will cover the following topics:

1. BUFSIZE and BUFNO DATA set options
2. Step boundaries
3. Recoding missing numeric values to 0
4. Using Data Set Indexes
5. Creating summary variables inside vs Outside of an iterating macro do loop
6. "Innocent" SQL
7. Reading the same data repeatedly, repeatedly
8. SQL Pass-through vs. using a SAS libname to reference an RDBMS table
9. Conditionally SET-ing datasets

There are countless online resources, written both by users and by SAS professionals, describing all the nuanced details of the topics discussed in this paper. We are not trying to be an exhaustive reference for each of the above, but rather an introduction to some of these more advanced topics, based on user's experiences and examples, with sufficient detail to whet your appetite enough to explore them on your own.

1. BUFSIZE AND BUFNO (Incorrect DATA Set Options Slow Me Down)

About a year ago, the first author inherited a bit of code which had originally been written (correctly, I might add) by the second author. The code had made it's rounds, being passed from one analyst to another before finally making it into the hands of the first author. When I finally received it, conceptually, the inherited code had numerous DATA Steps and iterations of PROC SQL which all looked something like the following.

Example 1

```
1  data two(bufsize=50 bufno=10);
2    set one;
3  run;
4
5  proc sql;
6    create table three(bufsize=50 bufno=10) as
7      select *
8        from two;
9  quit;
```

Importantly, the datasets the code was built to work on were large – measuring in the hundreds of millions of records each and containing roughly 40 variables each, approximately half numeric (8.) and half character (mostly $40.). Aside from the question of whether it is appropriate to be carrying $40 variables across hundreds of millions of observations, the real issue is that the above BUFSIZE and BUFNO DATA set options are not set appropriately. The original code was written more like the following:

Example 2

```
1  data two(bufsize=50K bufno=10);
2    set one;
3  run;
4
5  proc sql;
6    create table three(bufsize=50K bufno=10) as
7      select *
8        from two;
9  quit;
```

Somewhere along the way, the "K" had been dropped, intentionally or otherwise. So what, exactly, are the two bufsize and bufno options doing in the processing of our data?

Creating a data set is one of the more resource-intensive processes in the SAS system. Data sets must be saved somewhere (or "written" for us UNIX folks). Whether you are creating permanent data sets or work data sets, for SAS to process them, they exist in some physical space. The BUFSIZE and BUFNO data set options help control the flow of data input/output (I/O) into that physical space. You can think of this I/O data transfer as consisting of two parts: 1) the size of the bucket you will use to carry your data and 2) the number of buckets you will use.

As you might expect, BUFSIZE corresponds to the size of the buckets. In SAS, these are pages of data and BUFSIZE determines how much data may be moved via a data page in one I/O transfer. The page size (buffer) becomes an attribute of your data set and will affect all future processing of that data set. In the examples above, it is used on the output data set creation (e.g., the "two" and "three" datasets as they are created). BUFSIZE can only be changed when you are creating/recreating a data set – in an output data set. So trying to use this option when reading data into a DATA step or SAS Procedure has no effect (other than to produce a WARNING message in your SAS log file). Interesting note, however, is that if you are sorting a SAS Data set in place using the SORT Procedure (i.e., not using the out=option), you can apply both options on the data set and SAS will know to apply the BUFNO to both the input and output processing, and apply the BUFSIZE option only to the output data set. For example, the following code uses 10 buffers to input and output the data set and uses buffers of size=50K to output the sorted data set.

```
1  proc sort data=work.one(bufno=10 bufsize=50K);
2    by mysortvar;
3  run;
```

It is also important to recognize that maximizing the amount of data in a buffer may help to speed up data set processing, but keep in mind that increasing buffer size also increases your memory usage.

SAS online documentation shows that BUFSIZE may be set as follows:

BUFSIZE= *n* | *n*K | *n*M | *n*G

where n is an integer and K, M and G represent kilobytes, megabytes and gigabytes, respectively (additional options, such as hex and MAX are available for specifying BUFSIZE and you are encouraged to explore these as needed).

Similarly, BUFNO corresponds to the number of buffers (buckets) you're going to use to move your data around. BUFNO can be set as follows (note that the hex option also applies for BUFNO should you need it – see the SAS documentation for details):

BUFNO= *n* | *n*K | MIN | MAX

where n is an integer and K represents kilobytes. The default BUFNO setting is MIN, which sets the number of buffers to 0 causing SAS to use the minimum optimal value in your operating environment (note that system administrators in some environments will overwrite this setting to default to 1 instead of 0 in this case). Using the MAX option would likewise set the number of buffers to the largest possible number in your operating environment. Note that while BUFSIZE allows you to assign a permanent attribute to your output data set, the BUFNO option does not assign a permanent attribute to a data set and can be used when reading in or writing out data.

So, Example 1 above was trying to move a huge amount of data around using ten 50-byte buckets (500 bytes) for each I/O transfer. To put that into context, 500 bytes is 302 characters less than the number of characters in this paragraph. When the code in example 1 was run as is, it was still running when I arrived to work the next morning, finally consuming about 16 hours of processing time before completing. Eliminating the user assignment of the BUFSIZE and BUFNO options altogether (i.e., using the defaults for our system) actually sped things up to complete in about 6-7 hours. Optimizing settings for these two options further trimmed run time to approximately 4-5 hours. So, correctly using these options can drastically improve your run time, primarily when dealing with very large Data sets.

An analogous option exists when you are working with data being referenced via SAS views. The OBSBUF= option allows you to control the number of observations which can be read into the view buffer at once. Recall that a SAS view is really a window through which you are able to retrieve data, based on the "filters" that the view implements on your data. When an observation meets the conditions specified in your view, it is held in the view buffer until it is full. When full, SAS transfers the observation(s) from the view buffer to the requesting routine, either a Data step or PROC. More transferring means more I/O which means longer waiting time. So minimizing the number of data transfers can help improve your performance. Specifying OBSBUF=n allows you to control the number of observations kept in the view buffer before they are transferred. Of course, decreasing your waiting time while jobs run also means decreasing the amount of time you have available to surf the web, but the authors aren't really in a position to comment further on this topic.

A little knowledge can be a dangerous thing. As the authors have found, spending some time to research the best way to use the above techniques can vastly improve job performance. Consider what this means for time consumed during both development and production runs and you may have just given your boss a reason to give you a raise. You're welcome.

2. STEP BOUNDARIES (Your Best Friend and Worst Enemy)
Q: What's wrong with the below code?

```
1  title 'MYDATASET1';
2  proc print data=mydata.mydataset1(obs=10);
3
4  title 'MYDATASET2';
5  proc print data=mydata.mydataset2(obs=10);
6  title;
7  run;
```

A: Nothing. The code above runs just fine. It produces no ERRORs or WARNINGs and the first 10 rows of mydata.mydataset1 and mydata.mydataset2 print just as expected. Well, almost. The 'MYDATASET2' title is displayed over the printout for MYDATASET1 and no title is displayed for MYDATASET2. But why?

We're glad you asked. Line 1 assigns the first title, 'MYDATSET1'. Line 2 shows a syntactically correct (dare we say syntactically perfect?) coding of the PRINT Procedure. Line 4 reassigns our title to 'MYDATSET2' And line 5…ah ha…line 5 is where SAS actually executes the first PROC PRINT. The key here is to understand that for SAS to execute our PROC PRINT, it must hit a step boundary in the form of 1) a "run" statement, 2) a DATA step or 3) another PROC. A step boundary is something that SAS interprets to mean "Ok, I'm done with that last bit. I'll execute it before moving on to the next bit". In the example above, by the time SAS encounters the first step boundary and executes, the title has been reassigned from 'MYDATASET1' to 'MYDATASET2' and our output is misleading.

The above code can be fixed by simply inserting a "run" statement on line 3 so that the first PRINT PROC actually executes before the second title statement executes, clearing the title assigned on line 1. This seemingly small bit of SAS esoterica can befuddle you in the wee hours of the morning when you're putting the finishing touches on a piece of code that executes perfectly and yet you are left scratching your head trying to figure out why your title is not taking effect, preventing you from correctly interpreting your output. At least, that's what we've heard. From other SAS programmers.

The code below shows an example that, on it's face looks counter to what we just saw.

```
1  proc sql;
2    title 'Data from mydata.mydataset1';
3    select myvar1, myvar2, myvar3 from mydata.mydataset1;
4
5    select myvar1 into :mymacrovar1 from mydata.mydataset1;
6
7    title "&mymacrovar1. Data from mydata.mydataset2";
8    select myvar1, myvar2, myvar3, myvar4 from mydata.mydataset2;
9  quit;
```

Based on what we have seen so far, you might expect the "quit" statement to be the step boundary for these statements and perhaps, therefore, the last title statement to be the only one that takes effect. However, this example prints the results of the first "select" clause with the expected title as well as the results of the second "select" clause also with the expected title. Why is this? PROC SQL runs differently from the DATA Step and other SAS PROCs. It executes when you submit it, does not require a "run" statement, and therefore the titles above display along with their corresponding select clause. Rather than having to wait for a "run" or "quit" statement, the semi-colon ending each select clause *is* the step-boundary here. Additionally, the macro variable &mymacrovar1 is created in the second select clause and is available for use in the second title statement. This will be compared with the creation of a macro variable using the "call symput" routine next.

Since we want our titles (and our code in general) to be a bit more flexible and to make use of in-stream data rather than relying on hard-coded user input, we want to use macro variables to take advantage of SAS's ability to do this for us. The code below shows how to use CALL SYMPUT to create a macro variable from the value of the data set variable myvar1.

```
1  data _null_;
2    set mydata.mydataset1;
3    call symput('mymacrovar1',myvar1);
4    if &mymacrovar1. gt x then
5      do;
6        more code here…
7      end;
8  run;
```

Because &mymacrovar1 is created with “call symput”, it is not available until we hit the run statement. Or is it? More accurately, we can’t use &mymacrovar1 as written above. The answer lies in the RESOLVE function (documented under the Macro Language Dictionary in the SAS online documentation). If we replace line 4 above with

```
4    if resolve(‘&mymacrovar1.’) gt x then
```

we will be able to make the desired comparison within this Data step. Importantly, note the use of single quotes around &mymacrovar1 here. Use of double quotes here would result in a WARNING message being printed to your SAS log and your code not working as expected.

More information on step boundaries can be found in the SAS Online Documentation under
→ Base SAS
→ SAS Language Reference: Concepts
→ DATA Step Concepts
→ DATA Step Processing
→ About DATA Step Execution

and more information on the RESOLVE function can be found in the First & Ronk SUGI 31 paper referenced in the Reference section of this paper.

3. RECODING MISSING NUMERIC VALUES TO 0

```
1  data two;
2    set one;
3    array mynums {*} _numeric_;
4    do i=1 to dim(mynums);
5      if mynums{i} = . then mynums{i}=0;
6    end;
7  run;
```

Again, a little knowledge is a dangerous thing. On the surface, the above code should do what we want – if what we want is to recode all numeric variables from missing to zero instead. Unfortunately, what SAS can’t do is ask us “Do you really want to do this?” Before performing an operation like this, at the very least we must know specifically which variables we are affecting. Do we have a numeric key variable used to merge or join tables together in a relational database? And consider, for purposes of this section of the paper, what happens when you have missing date values?

To be included in processing of many SAS Statistical Procedures, data records are required to contain no variables with missing values. We often get around this by reassigning missing numeric values to 0. In some cases, software tools may be written specifically to reassign all numeric variables on a data set which contain missing values to 0. The Array processing code shown above using the _numeric_ keyword is another way to efficiently recode variables with missing values and set them equal to 0. When these data points are fed into a Statistical Procedure, SAS processes them and no records are dropped from the analysis. We interpret results and make business decisions.

Answering the question of whether missing values in your data contain meaningful information which may appropriately be recoded is something only the Analyst can do. However, when our modeling data set contains missing date values, recoding them to 0 effectively tells SAS to impute January 1, 1960 for all missing values. You probably don’t want to do this. Said a different way, you definitely don’t want to do

this. This is an especially vexing problem because SAS will do exactly what you request, producing no errors, warnings or notes in your log alerting you to the potential problem. And your resulting model will be biased at best, completely invalid at worst. Quite simply, this section is a call out to you to be very, very careful when dealing with missing values. Learn from what others have done and don't make this mistake yourself. Of course, we have also provided some efficient code for doing exactly what we have warned you not to do.

4. **USING DATA SET INDEXES (Appending and Dropping and Sorting, Oh My!)**
Using Data set Indexes to facilitate direct data access (as opposed to sequential access) when reading data can vastly improve the efficiency of your code. That said, there are several things to watch out for when using Data Set Indexes which can sneak up and bite you if you're not aware of them.

The first of these that we'll cover is a fairly intuitive case. A colleague of one of the authors ran the following code and encountered the highlighted NOTEs in her SAS log:

```
1     proc append base=work.mydataset
2                 data=perm.mydataset(keep=hh_id var2 var3 var4);
3     run;

NOTE: Appending PERM.MYDATASET to WORK.MYDATASET.
NOTE: BASE data set does not exist. DATA file is being copied to BASE
      file.
NOTE: Index VAR1 not created because of dropped variables.
NOTE: Simple index VAR2 has been defined.
```

The first thing to notice from the above list of NOTEs is that we can infer that the data set PERM.MYDATASET has two Indexes associated with it, one on VAR1 and one on VAR2. Since VAR1 is not kept when PERM.MYDATASET is appended to WORK.MYDATASET, the associated Index, VAR1, is not created. When our colleague came for a SAS consult as to what this meant, the point of interest around this was not that the VAR1 Index was not created, but rather that SAS kept the Index on VAR2 when it copied the permanent data set to the work library. In other words, we have a case of SAS doing the best it can without ERROR-ing out and stopping processing. The "Simple" Index refers to the fact that the Index is created based on the values of a single variable. In this example, when VAR1 was dropped, the Index on VAR2 was kept. But what would have happened if we had a Composite Index composed of VAR1 and VAR2?

A Composite Index is based on the values of multiple variables, combined to form a single concatenated value which can be used to increase efficiency when selecting data using WHERE processing. If we had had a Composite Index composed of VAR1 and VAR2 and then dropped VAR1 as we appended the permanent dataset into the working library, there would be no Indexes present on the data set created in the work library. SAS still does as much as it can for us here, but unfortunately, it doesn't get very far and in this case does not produce the results we want. SAS continues program execution in this case, writing a note similar to the one shown above that the Index was not created because of dropped variables. However, since it does not alert us by issuing an ERROR or a WARNING message in our log, our processing efficiency will be impacted when we go to access this data set because it is no longer indexed as expected. Caveat Programmer!

Next, our colleague came to us with another question, this time resulting in the following ERROR in her SAS log:

```
1     proc sort nodupkey data=work.mydataset;
2       by hh_id;
3     run;

ERROR: Indexed data set cannot be sorted in place unless the FORCE
       option is used.
```

The SAS documentation indicates that this is the case. The workaround for this is to sort the data set into a different data set. However, when sorting one data set into a second, your original Index is not

automatically kept. Using the INDEX= Data set option allows you to get around this "on-the-fly", so that your final data set is structured exactly how you want with all appropriate Indexes in place as well. The code segment shown below demonstrates how to use this option to create both a Simple as well as a Composite Index on an output data set:

```
1     proc sort data=work.mydataset
2                out=work.mydataset2(index=var2 v34=(var3 var4))
3               nodupkey;
4       by hh_id;
5     run;
```

By using the steps above, WORK.MYDATASET2 now supports Index processing on VAR2 and on the Composite Index for the VAR3/VAR4 combination. You can read more about creating and using Simple and Composite Indexes (including factors to consider when creating Indexes and all the various ways to create them) in the informative chapter of the SAS Online documentation:

→ Base SAS
→ SAS Language Reference: Concepts
→ SAS Files Concepts
→ SAS Data Files
→ Understanding SAS Indexes

5. CREATING SUMMARY VARIABLES INSIDE VS. OUTSIDE AN ITERATING MACRO DO LOOP

So, just hypothetically, let's say you want to create some z-score variables to help you analyze your data. Further, you want to create your z-scores separately for each unique value of some ID variable – let's say by a household ID variable. Finally, you want to create an overall score for each household for each category that is the sum of these z-score variables with the goal of differentiating across categories but within household. You hope your new scoring methodology will help you make some more informed business decisions. Some example data is shown below in Table 1 to clarify what we're working with. Code to process these data are shown following.

Table 1: Example Z-score data for HH #1

HH_ID	CATEGORY	VAR1	AVG VAR1	STD VAR1	Z_SCORE VAR1
1	1	19	57	30.042	-1.265
1	2	38	57	30.042	-0.632
1	3	57	57	30.042	0.000
1	4	76	57	30.042	0.632
1	5	95	57	30.042	1.265

The above shows an example for HH #1 for VAR1. Suppose we have similar data for five additional variables (e.g., AVG, STD and Z-score variables for a total of 6 variables). We want to sum across the 6 z-score variables for each HH_ID for each CATEGORY to create a composite score. Code to do all this is shown below.

```
 1  %let varlist=var1 var2 var3 var4 var5 var6;
 2
 3  %macro zcode;
 4     %let i=1;
 5     %do %while(%scan(&varlist.,&i.,%str( )) ne %str());
 6       %let thisvar=%scan(&varlist.,&i.,%str( ));
 7
 8       proc means data=mydata noprint;
 9         by idvar;
10         var &thisvar.;
11         output out=work.&thisvar._stats
12               mean=avg_&thisvar.
13                std=std_&thisvar.;
14       run;
```

```
15
16       data mydata;
17         merge mydata(in=a) work.&thisvar._stats(in=b);
18         by idvar;
19         if std_&thisvar. in (.,0) then std_&thisvar.=1;
20         z_score_&thisvar.=(&thisvar. - avg_&thisvar.) /
21                           std_&thisvar.;
22         z_score_overall=sum(of z_score:);
23       run;
24
25       %let i=%eval(&i.+1);
26     %end;
27  %mend zcode;
28  %zcode;
```

Line 1 sets up the list of variables on which we will run our analyses. All analyses between the %do and the %end statement will happen once for each variable in this list.

Line 3 initiates the "zcode" macro.

Line 4 initializes the macro variable we will use to iterate through the variable list contained in the macro variable &varlist.

Lines 5-6 do the bulk of the work that allows us to iterate through the variable list. We begin by scanning &varlist and selecting the first variable so long as the result of the scan function does not return a blank value [e.g., `%do %while(%scan(&varlist.,&i.,%str( )) ne %str())`]. Line 6 creates &thisvar which will change values each time &i is incremented.

Lines 8-14 run the MEANS Procedure by IDVAR (e.g., our household ID variable) and output a dataset containing the mean and standard deviation for each household for each variable in &varlist.

Lines 16-21 merge the mean and standard deviation data just derived back onto the original data set so that we'll have all the information we need to derive our z-scores. Because our z-score formula is:

z = (observation – average) / standard deviation

for our purposes we want to ensure that the standard deviation never equals 0 or missing, so in these cases, we recode it to equal 1. Lines 20-21 then use the above formula to derive the z-score for each idvar (household).

Line 22 is where things run amuck. We want to create a variable summarizing each of the 6 z-score variables for each Household for each CATEGORY. Rather than passing through the data (yet again), we decide to add our summary calculation into this data step by adding the Z_SCORE_OVERALL variable as a sum of all the z_score variables on the data set. Since we're iterating through &varlist and creating more z-score variables each time we do, we think we have been quite astute by starting the name of the summary variable using the same naming convention as the individual z-score variables. We can now reference all our variables very efficiently using "z_score:". Do you see the issue yet?

Since we have created this variable inside the iterating %do loop, the value of the summary score (e.g., Z_SCORE_OVERALL) is also summarized each time the macro iterates, along with all the previously created individual z-score variables. Following is a conceptual view of what the code would actually be doing during each iteration of the %do loop:

```
When &i=1, then z_score_overall=sum(of z_score_var1);

When &i=2, then z_score_overall=sum(of z_score_var1
                                    z_score_overall
                                    z_score_var2);

When &i=3, then z_score_overall=sum(of z_score_var1
                                    z_score_overall
                                    z_score_var2
                                    z_score_var3);

etc…
```

Because we have created our summary variable inside the %do loop and named it using the same naming convention as the contributing variables, we are effectively overweighting the variables that come first in &varlist. A better solution (i.e., a correct one) is to create the final summary variable in a subsequent DATA step – and outside the iterating %do loop.

Of course all of this could be avoided by simply using the STANDARD Procedure with a BY statement. But that's a topic for another paper.

6. "INNOCENT" SQL (Is The Bane Of My Existence)

It is currently 3pm in the afternoon and this is the umpteen millionth time you have examined this code. In fact, you have read this simple-looking program so many times that every line is burned into your synaptic pathways. Diagnostically, you note that everything works fine until the SQL step where you access the RDBMS: it seems to linger there for a while holding out a glimmer of hope that the step will prevail and then rudely aborts with nasty messages that by now feel like invective.

Out of desperation you scan the office around you, seeking validation of your sanity. Instead you discover with pent up glee nothing but company for your misery. Yes, you are not the only one suffering and the light bulb burns brightly for an instant: perhaps it is not your inferior coding skills, but rather something outside your control that is ruining your day. You immediately pick up the phone and dial the number of your DBA (every programmer's best friend, next to system admins). Yes, comes the reply – there is something odd happening – everyone's jobs appear to get queued and then destroyed.
A little later the email comes with the most innocuous SQL query:

```
1   Select *
2   From   MyCustomers a,
3          Customer_Segments b
4   Where  a.Customer_ID = b.Customer_ID
```

At first glance, this is a fairly straightforward request – return customer segment metrics for all customers matching those 10M customers in the MyCustomer table. The only problem is that this person had failed to recognize that the partitioned Customer_Segments table contains segments on a monthly basis for the past ten years. Thus, the RDBMS, in trying to fulfill this request, has allocated almost 90% of its 1T of work space to this job and thus, was completely unable to service all of requests.

You work with the DBA to kill this person's job (with prejudice) and become the hero in your group. You then spend a few valuable moments with that person on one of the most important topics: knowing your data and ensuring your code reflects what you really want – segment data for one month. The revised query now runs in 15 minutes:

```
1   Select *
2   From   MyCustomers a,
3          Customer_Segments b
4   Where  a.Customer_ID = b.Customer_ID
5   And    b.Period = "Jan/2010"
```

7. READING THE SAME DATA REPEATEDLY, REPEATEDLY

There appears to be a myth propagated by some SAS programmers and perhaps by the SAS log that the use of a where clause in a data step miraculously allows you to skip over unwanted observations in the data set. It is as if Einstein waives the laws of physics for those lucky SAS users who invoke this magical statement. Thus, the author has witnessed code such as the following:

```
1   %Macro Process(Code) ;
2       Proc Means Data = SASDATA.MasterData NoPrint ;
3          Where Code EQ &Code ;
4          Var Y ;
5          Output Out = Summary&Code Mean= Std= / Autoname ;
6       Run ;

7       Proc GLM Data = SASDATA.MasterData NoPrint ;
8          Where Code EQ &Code ;
9          Class X Z ;
10         Model Y = X|Z ;
11         Output Out = GLM&Code PRESS COOKD DFFITS ;
12      Run ;
        ...more processing...
13  %Mend ;

14  %Process(001) ;
       ...
15  %Process(999) ;
```

The SAS log – in collusion – will report that the number of observations read equals only those that met the where condition: Not the total number of observations. Clearly, the impact is not felt as significantly when the master data is small, say less than 10G. However, as the data set size and/or the number of times the data are read increases, the processing time tends to increase *exponentially*. At this point, most programmers throw up their hands and declare that this is the most efficient way to accomplish this task. There is nothing to do but wait.

For data sets large and small, may I suggest another method that may prove beneficial: *By Group Processing*. This method requires that the originating data set be sorted (or at least ordered) according to a variable list. Then you simply add the statement "BY varlist" for each process. The above macro simplifies to the following:

```
1   Proc Means Data = SASDATA.MasterData NoPrint ;
2      By Code ;
3      Var Y ;
4      Output Out = Summary Mean= Std= / Autoname ;
5   Run ;

6   Proc GLM Data = SASDATA.MasterData NoPrint ;
7      By Code;
8      Class X Z ;
9      Model Y = X|Z ;
10     Output Out = GLM PRESS COOKD DFFITS ;
11  Run ;

    ...more processing...
```

The above code replaces the WHERE statement with a BY and processes the data exactly once for each procedure. SAS uses divide and conquer when it encounters the BY statement: each group of observations that contain the same values in their BY variables are treated as a separate entity (hence, the term "BY GROUP"). Thus, SAS effectively reduces the problem for you, you do not have to know all of the data values in advance (making the code more versatile), and your overall code is simpler. You

are additionally rewarded with the bonus that all of the output data sets contain the variables in the BY statement duly populated.

8. SQL PASS-THROUGH VS. USING A SAS LIBNAME TO REFENCE AN RDBMS TABLE

Many programmers who obtain their data from an external RDBMS, such as Oracle or Teradata, never exploit the power of these products. Suppose you have a RDBMS that contains transactional level data for the past year and you need to provide sales summarized at the department level. The transactional data is stored in an Oracle table as is the department information for each product in the transactional data.

SAS programmers have a tendency to write in SAS, and thus, the authors often encounter the following:

```
1   Libname Oracle Oracle (...oracle connection information) ;

2   Proc SQL NoPrint ;
3      Create Table SalesSummary As
4      Select b.DepartmentName, Sum(a.Sales) As DeptSales
5      From   Oracle.TransactionData a,
6             Oracle.Departments     b
7      Where  a.Product = b.Product
8      Group By b.DepartmentName ;
9   Quit ;
```

The above SQL looks and feels like ordinary SAS code. However, when one looks under the hood, we see exactly how wasteful this approach is. First, when SAS encounters a libname reference, it converts everything to its native SAS format, regardless of the data source or amount. Having SAS perform all of this work can be a boon for programmers when used sparingly. However, in this case, SAS is requesting that every row from TransactionData and Departments be returned from the RDBMS (through a relatively small funnel), converted to SAS native format, then matched by Product and grouped according to DepartmentName. Then, SAS creates the SalesSummary data set with the DepartmentName and Total Sales for each department. The problem, if you do not see it yet, is that this query is retrieving and converting a massive amount of data and completely ignoring the fact that the RDBMS can do this task faster and more efficiently.

To do this, we need to use SQL PASS-THROUGH. This means that we *pass through* to the RDBMS an SQL query that will execute in the RDBMS. In this revised scenario, the RDBMS performs all of the matching, grouping, and summarization on data in its native format, and only then transfers to SAS the *much smaller* table. The resulting code using SQL pass-through appears as follows:

```
1   Proc SQL NoPrint ;
2      Connect To Oracle (connection information to Oracle RDBMS) ;
3      Create Table SalesSummary As
4      Select *
5      From Connection To Oracle
6      (Select b.DepartmentName,
7              Sum(a.Sales) As DeptSales
8       From   Oracle.TransactionData a,
9              Oracle.Departments     b
10      Where  a.Product = b.Product
11      Group By b.DepartmentName
12     ) ;
13  Quit ;
```

For the most part, the SQL looks the same, with the exception of that funny connection to bit. Actually, the SQL deconstructs quite nicely into what happens in SAS and what happens inside the RDBMS. Line 2 creates a logical connection to the RDBMS, analogous to a libname statement. The difference is that this method allows you to tell the RDBMS to do things not possible with a libname. Lines 3-4 are straightforward SAS SQL – create a SAS data set. Line 5 represents the dividing line between SAS and your RDBMS, or where SAS leaves Kansas. This line tells SAS that everything in the parentheses (i.e.,

lines 6-11) will be executed by the RDBMS and that the result will be returned. Now, if you look at the query inside the parentheses you can see that it is basically the original query. Only now, we are having it return the results to SAS.

Another way to look at it is that this is a query with a sub-query that executes in the RDBMS, not in SAS. This method may seem a little odd at first, but after a while, it will become quite natural to tell your RDBMS to do things for you (and in record time). You should be aware that SAS/SQL is not likely to conform to the same standard of SQL employed by your RDBMS. This may cause problems with syntax initially, but this is where your friendly DBA will come in handy.

Finally, SAS has some intelligence built-in when accessing RDBMS tables. For instance, consider the following code:

```
1   Libname Oracle Oracle (...oracle connection information) ;

2   Proc Sort Data = Oracle.Departments Out = Work.Depts ;
3      By Product ;
4   Run ;
```

Line 1 sets up the connection to the RDBMS and lines 2-4 run the sort procedure, specifying the Departments table from the RDBMS as input with the result being the temporary SAS data set Depts. When this executes, SAS will notify you in the log that it knows the RDBMS can do the sort, and thus, has pushed through instructions to that effect. That is, the sorting is performed in the RDBMS, not on your overworked and underpowered box. This magic occurs without your having to know it in advance, but you do get to take as much credit for it as you think you can get away with.

9. CONDITIONALLY SETTING DATA SETS (TO BUILD THE PDV CONSISTENTLY)

This author has come across many programs with code similar to the following:

```
1   Data <some data set> ;
2      Length Var1 <length> Var2 <length> Var3 <length> ... ;
3      ...processing...
4   Run ;
```

More often this code is copied / pasted throughout the program (or to other programs). While this may appear to be an efficient way to set up these variable definitions, what happens if they change? What if someone changes them inadvertently in one program and does not notice? It is at this point that well heeled programs begin to kick out errors messages about metadata not matching, most likely during a MERGE or APPEND.

May we suggest another simpler (and dynamic) method to build up your PDV (program data vector). It is based on the principal that you should build it once or not reinvent the wheel. In effect, if the variable definitions already exist in another location, then steal them:

```
1   Data <some data set> ;
2      If 0 Then Set <DataSet with Needed Variables> ;
3      ...processing...
4   Run ;
```

The change to line 2 tells the SAS compiler to pull the variable definitions from the dataset and add them to the PDV. However, since zero is never true (in a logical sense) then data is never read from this source. It is as if you had spent many hours typing in all of those variable attributes, including formats and labels (which can be so helpful).

If the variables to be defined are to match another data source (as is often the case), then this method can minimize changes to programs and reduce time spent tracking down sources of errors. More importantly, using this technique ensures that you get maximum benefit from a minimal amount of effort.

CONCLUSION

The general theme of this paper has been “a little knowledge can be a dangerous thing”. We have seen quite a few examples of how trying to get fancy and do lots of “really cool stuff” can get you into trouble if you haven’t carefully explored all the intricacies of the options you’re using. We hope that our calling out these techniques will help you learn from them, improve your code, make fewer “mistakes” yourself, get home earlier so your do will love you even more and, with just a little luck, get yourself the status of “SAS Guru” emblazoned on a certificate suitable for framing.

Happy Coding!

REFERENCES

First, Steven and Ronk, Katie, 2006, Intermediate and Advanced SAS® Macros, presented at the 31st Annual SAS User’s Group International, SUGI, meetings (March 2006) and published in the proceedings of the 31st Annual SUGI Conference, 2006..

SAS OnlineDoc® 9.1.3. SAS Institute Inc., Cary, NC, USA.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You can contact the authors below:

Joe Novotny
dunnhumby USA
444 W. 3rd Street
Cincinnati, OH 45202
joe.novotny@us.dunnhumby.com

Michael Bramley
dunnhumby USA
444 W. 3rd Street
Cincinnati, OH 45202
michael.bramley@us.dunnhumby.com

AUTHOR BIOGRAPHIES

Joe Novotny began programming SAS during his stint in Happy Valley in grad school at Penn State (1996). After receiving his Master’s in Human Development, he switched industries to pharmaceutical research where he did not sample the product, but did use SAS for 8 years analyzing phase II and III clinical trials. In 2007, Joe switched industries again and began working with transactional marketing data for dunnhumbyUSA as a Direct Marketing Analyst. Joe has presented at MWSUG and NESUG and is currently involved with the Cincinnati SAS User’s Group (CinSUG) and MWSUG. When he is not hard at work coding SAS, Joe thinks about coding SAS. Joe dreams in IML and recommends never letting a co-author sit in your seat.

Michael Bramely first learned SAS in a small one computer village without internet. He found it easier to type after he evolved opposable thumbs. This spurred him on to earn degrees in Math, Medieval History and something called ‘Statistics’ (you may call it worse things). He worked in the pharmaceutical industry as one of those ‘statistic-guys’ until he saw the light and moved to dunnhumbyUSA in 2006. He is rumoured to be involved in CinSUG.