Paper SA04-2012

Getting the Code Right the First Time

Michael C. Frick, General Motors, Warren, MI

ABSTRACT

Writing code to support data analysis has a unique set of problems and is a vastly different task than writing traditional software. More often than not, it is throwaway code whose ultimate direction is not always known at the start of the project – intermediate results drive further investigative steps. Additionally, there is often a time-sensitivity to these types of projects. This limits the usefulness of traditional requirements gathering and technical specification writing. Even so, writing code to support analytics can be tricky and one must stick to simple, repeatable techniques to confidently write code that works the first time, every time. In this paper, I will describe a process of work developed over the last 30 years aimed at eliminating both coding and data interpretation errors as they occur. The methods are not rocket science, but rather rely on straight-forward techniques and "data diligence".

INTRODUCTION

"Approximately half the articles published in medical journals that use statistical methods use them incorrectly." [1] Published accounts of statistical errors in the literature come directly from a review of the literature itself. But how can one identify faulty data preparation from a literature review? Great statistical analysis on top of bad data is potentially more troublesome than mediocre analysis on top of solid data. As this paper will focus heavily on coding techniques and not so much on statistical techniques, a fair question would be "Why isn't this paper being presented in a BASE SAS® forum?" My response is simple: garbage in – garbage out. Data preparation may not be sexy, but it is a critical part of the statistical analysis process that is often short-changed even by experienced analysts.

One wintery day in the mid 80's, we got a call from a government researcher stating that he was having difficulty reproducing numbers from one of our published papers on traffic safety.[1] He was sufficiently sure of his results that he had already reported his findings to both his and our senior leadership. As our paper was considered by many to be a land-mark work in the area of safety belt effectiveness, was based on publicly available fatal accident data maintained by the US government, and was being challenged by one of its own highly regarded traffic safety researchers; we were obviously concerned.

It turned out to be faulty data filters in the government study that had caused the difference; and, as a young analyst, I thought long and hard on whether I had been good or just lucky. Either way, I learned a valuable lesson – you cannot over check the impact of your filters. It also leads one to wonder – just how much of the literature out there is wrong? Given their confidence in their original results, it is likely that the government study referenced above would have been published with incorrect results. This paper is focused on four major themes:

1. An "iterative programming" technique which breaks the code development process up into small chunks that are fully developed and tested before going on to the next "chunk".
2. Use of simple, repeatable coding techniques that have stood the test of time.
3. Data filtering is king. Check and recheck – know your row counts, your data, and your filters. Treat the data filtering trail as a police detective treats his evidence trail.
4. SAS® software is the perfect environment to implement Steps 1 – 3.

ITERATIVE APPROACH TO WRITING CODE

I believe very strongly in the divide and conquer strategy when it comes to any task, especially coding. I took a graduate systems course from an extraordinary professor who happened to be both blind and Jewish while teaching at a Jesuit school.[2] When passing out coding assignments, he gave very explicit instructions as to the number, length, and content of subroutines for each project. It irritated me until one day I stopped by to see him in his office and found him engaged with a teaching assistant who was reading source code to him. I realized then that by forcing us to use small routines, he was forcing us to provide him with chunks of code that could be processed in his head.

[1] Readers interested in traffic safety research (and perhaps misdirected government policy) should visit my long-time research partner and mentor's web-site at www.scienceservingsociety.com.

[2] Abraham Nemeth taught at the University of Detroit for over 30 years. He is well known for his contributions to mathematics, computer science, and educational aids for the blind (e.g. the Nemeth Braille Code for Mathematics and Science Notation).

This made a distinct impression on me and to this day I seldom write more code at one time than I can actively process in my mind in one chunk before I test. As an example, let's assume I need to build a DATA step to read a text file, recode some of the variables, and filter the data based on values of one of the input variables. Rather than write the entire DATA step in one fell swoop, I would break this task into 4 steps, each of which is tested fully before going on to the next task.

1. Set up the DATA step so that it only reads the text file and prints out the first 10 or so lines. Validate that the SAS output actually matches the input table for each field (that is, did I read the raw data correctly.)

2. Let SAS read the rest of the file, run the FREQ procedure on the categorical variables, run the UNIVARIATE procedure on numeric variables, and validate that values read are legitimate (if working with a new data set, you most likely will need to check with a subject matter expert). Double check that it read the correct number of rows. The SAS notes are invaluable in this process.

3. Add the code to recode the variables. Use PROC FREQ (cross-tab) to double check that the encoding is correct.

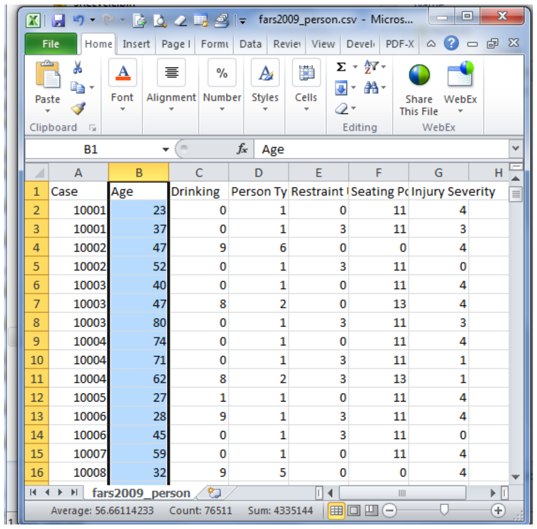
4. Finally add the code to filter the data. Use PROC FREQ to determine the expected number of observations before filtering and then double check the counts after filtering the datasets.

By breaking the larger task into these smaller tasks and testing as I go, I have a better chance of identifying bad code as it is written; i.e., while my brain is still fully engaged with what I was trying to accomplish with the code. It also gives me a chance to uncover problems in the data (or my understanding of the data) prior to writing the code to process that data.

As an example, take a look at the Microsoft Excel [2] sheet in Figure 1[3]. It has 76,510 observations with 7 fields. Each row corresponds to a person who was involved in a fatal accident in the US during calendar year 2009. Assume our task is to read this comma delimited text file and subset the data to deceased, restrained drivers of motorized vehicles. We have been told that each field has a code for missing values (999 for age, 99 seating position, and 9 for all other fields.) Although we do not know all of the individual codes for the variables, we have been told that a value of "0" implies no drinking and "1" implies drinking. Further a restraint code of "3" implies a person was wearing a standard 3 point lap/shoulder belt, a seating position code of "11" implies that the person was sitting in the left front seat, and that an injury severity of "4" implies the person died as a result of the accident.

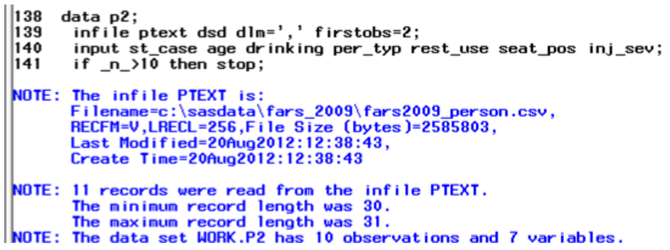
STEP 1 – ENSURE PROPER READING OF THE STRUCTURE OF THE DATA

As a first step, I want to focus on the mechanics of reading the file correctly. The DATA step in Figure 2 reads the first 10 lines from the file which can be dumped to the output window. Notice the wealth of information in the notes which tell us if we indeed read the correct file, the expected number of records, and whether or not the actual file is wider than expected (more characters, SAS defaults to 256.) Further the SAS NOTES would have flagged illegal data if present (say character data in a numeric field). Also, by inspection of the output window, we can get a quick read on whether or not we are reading the individual fields properly and starting with the correct first row.



	A	B	C	D	E	F	G
1	Case	Age	Drinking	Person Ty	Restraint	Seating Pc	Injury Severity
2	10001	23	0	1	0	11	4
3	10001	37	0	1	3	11	3
4	10002	47	9	6	0	0	4
5	10002	52	0	1	3	11	0
6	10003	40	0	1	0	11	4
7	10003	47	8	2	0	13	4
8	10003	80	0	1	3	11	3
9	10004	74	0	1	0	11	4
10	10004	71	0	1	3	11	1
11	10004	62	8	2	3	13	1
12	10005	27	1	1	0	11	4
13	10006	28	9	1	3	11	4
14	10006	45	0	1	3	11	0
15	10007	59	0	1	0	11	4
16	10008	32	9	5	0	0	4

Figure 1: Extract from 2009 NHSTA Fatal Accident Reporting System



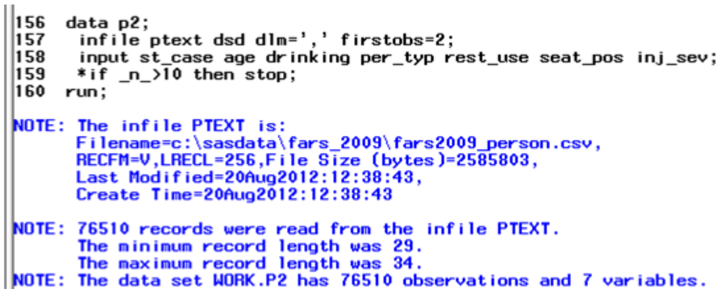
```
data p2;
 infile ptext dsd dlm=',' firstobs=2;
 input st_case age drinking per_typ rest_use seat_pos inj_sev;
 if _n_>10 then stop;

NOTE: The infile PTEXT is:
      Filename=c:\sasdata\fars_2009\fars2009_person.csv,
      RECFM=V,LRECL=256,File Size (bytes)=2585803,
      Last Modified=20Aug2012:12:38:43,
      Create Time=20Aug2012:12:38:43

NOTE: 11 records were read from the infile PTEXT.
      The minimum record length was 30.
      The maximum record length was 31.
NOTE: The data set WORK.P2 has 10 observations and 7 variables.
```

Figure 2: DATA Step to Read First 10 Lines

[3] The data in this sheet is from an extract of the Person File from the US Fatal Accident Reporting System (FARS) data for calendar year 2009. FARS is a database of fatal vehicle accidents in the United States that is maintained by the National Highway Transportation Safety Administration (NHSTA) and is made publically available on their web-site.

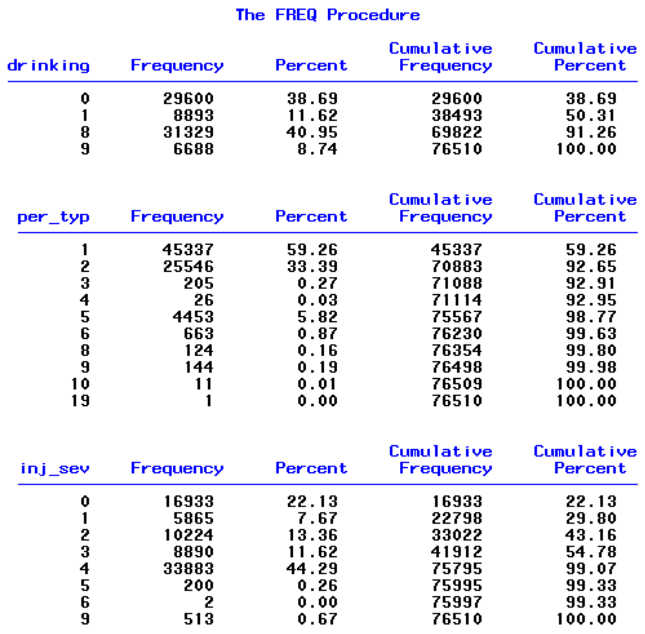
STEP 2 – ENSURE PROPER READING OF THE FIELDS, STRUCTURE AND CONTENT

As a second step, I want to focus on whether the data in the individual fields match what I was expecting. Notice from the SAS NOTES in Figure 3 that I am now reading 76,510 records as expected. Do not underestimate the value of these notes. PROC FREQ is used to check field values. Based on the starting information, I was expecting to see values of 0 (no alcohol), 1 (alcohol), or 9 (missing/unknown). Yet from Figure 4 we can see that 41% of the responses have a value of 8. NHSTA makes a distinction between not reported (value of 8) and missing/unknown (value of 9). What if I had simply written the code to drop 9's and count everything greater than zero as alcohol related? (I have seen stranger code than this). It is exactly this type of issue I am trying to uncover in the second step. Finding an error in either the data itself or in my understanding of the data before I have recoded or filtered any variables saves time in the long run. Also notice that every record has a coded value for each of the three variables listed below. This is not always the case, even if a code for missing values exists.



```
data p2;
 infile ptext dsd dlm=',' firstobs=2;
 input st_case age drinking per_typ rest_use seat_pos inj_sev;
 *if _n_>10 then stop;
run;

NOTE: The infile PTEXT is:
      Filename=c:\sasdata\fars_2009\fars2009_person.csv,
      RECFM=V,LRECL=256,File Size (bytes)=2585803,
      Last Modified=20Aug2012:12:38:43,
      Create Time=20Aug2012:12:38:43

NOTE: 76510 records were read from the infile PTEXT.
      The minimum record length was 29.
      The maximum record length was 34.
NOTE: The data set WORK.P2 has 76510 observations and 7 variables.
```

Figure 3: DATA Step to Read Entire Text File



```
                          The FREQ Procedure

                                          Cumulative    Cumulative
drinking    Frequency     Percent         Frequency      Percent
-------------------------------------------------------------------
       0       29600        38.69            29600        38.69
       1        8893        11.62            38493        50.31
       8       31329        40.95            69822        91.26
       9        6688         8.74            76510       100.00

                                          Cumulative    Cumulative
 per_typ    Frequency     Percent         Frequency      Percent
-------------------------------------------------------------------
       1       45337        59.26            45337        59.26
       2       25546        33.39            70883        92.65
       3         205         0.27            71088        92.91
       4          26         0.03            71114        92.95
       5        4453         5.82            75567        98.77
       6         663         0.87            76230        99.63
       8         124         0.16            76354        99.80
       9         144         0.19            76498        99.98
      10          11         0.01            76509       100.00
      19           1         0.00            76510       100.00

                                          Cumulative    Cumulative
 inj_sev    Frequency     Percent         Frequency      Percent
-------------------------------------------------------------------
       0       16933        22.13            16933        22.13
       1        5865         7.67            22798        29.80
       2       10224        13.36            33022        43.16
       3        8890        11.62            41912        54.78
       4       33883        44.29            75795        99.07
       5         200         0.26            75995        99.33
       6           2         0.00            75997        99.33
       9         513         0.67            76510       100.00
```

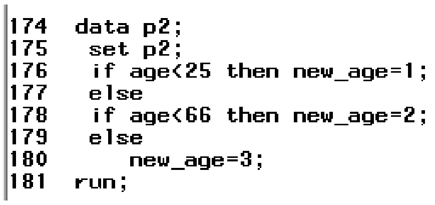
Figure 4: Frequency Counts of Key Variables

At this point, it is important to understand legal values for fields and, if at all possible, expected relative penetration of each value. In the case of the FARS data, one can check the coder's manual to find out legal values for every field and NHTSA publications for penetration. Injury severity of 4 implies a fatality. The 33,883 fatalities in Figure 4 exactly match the expected total [3]. For internal data sources, local subject matter experts can be invaluable sources for confirmation of data levels.

Depending on the data source, I will run queries outside of SAS against the input file and compare to the results from PROC FREQ to help further validate that I have read the data properly. In the old days with fixed column input, I would use the "find all" command of a text editor to give me counts of levels for a given field. Today, I take the time to load the data into Excel and run a PIVOT table to get frequency counts. This level of effort may seem extreme to some, but my goal is to make 100% certain that I have loaded the data properly and have confidence that the data is valid. I never work on a new data set without running a PROC FREQ on all fields and PROC UNIVARIATE on numeric fields and then checking values with either known references or subject matter experts. If I have an established code running against a known data set, I will add code to check for legal values and ABORT the data step if a value is found outside of the good data range. When working with a customer at the start of an important project, my preferred approach is to validate the univariate results with them prior to starting the actual analysis. Again, my goal is to do everything in my power to know the data is solid before starting an analysis.

STEP 3 – RECODE VARIABLES

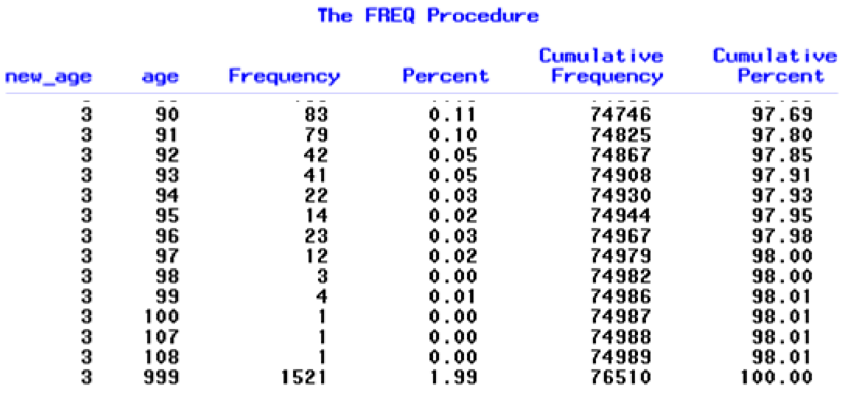
In the third step, it is time to recode variables if needed. Assume we want to split people into 3 age groups (young: 0-24, middle: 25-65, and old: over 65)[4]. In this case, the "IF statement" is straightforward. Even so, resist the urge to just plug in and go. Take the time to do a cross-tab of the original and the re-coded variables as a double check. Can you see the flaw in the code shown in Figure 5? The mistake may be trivial, but I will bet that just about everyone has made it. Results from this DATA step are shown in Figure 6.



```
data p2;
 set p2;
 if age<25 then new_age=1;
 else
 if age<66 then new_age=2;
 else
    new_age=3;
run;
```

Figure 5: Flawed Data Step to Recode a Variable

[4] There was a time when I used to break middle-aged and old at age 55 - no longer.



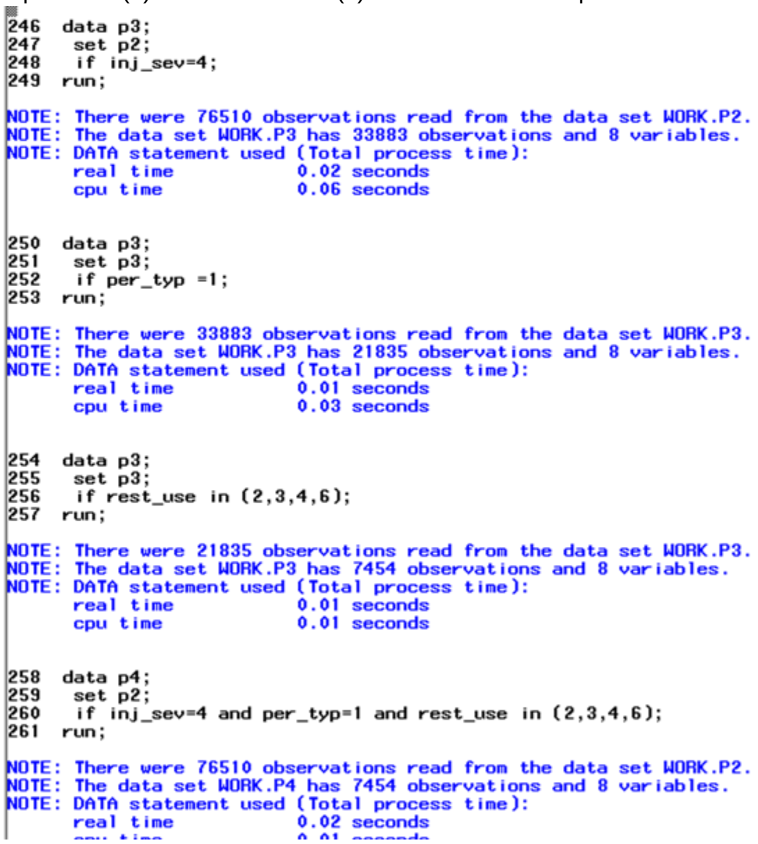
```
                         The FREQ Procedure

                                                  Cumulative    Cumulative
new_age     age    Frequency      Percent         Frequency       Percent
        3      90          83         0.11            74746         97.69
        3      91          79         0.10            74825         97.80
        3      92          42         0.05            74867         97.85
        3      93          41         0.05            74908         97.91
        3      94          22         0.03            74930         97.93
        3      95          14         0.02            74944         97.95
        3      96          23         0.03            74967         97.98
        3      97          12         0.02            74979         98.00
        3      98           3         0.00            74982         98.00
        3      99           4         0.01            74986         98.01
        3     100           1         0.00            74987         98.01
        3     107           1         0.00            74988         98.01
        3     108           1         0.00            74989         98.01
        3     999        1521         1.99            76510        100.00
```

Figure 6: Coded versus Un-coded Age Variable, Age>=90

STEP 4 – FILTER THE DATA

Finally, in the fourth step, we are ready to filter the data. Remember that our task was to restrict the data to dead, restrained drivers of motorized vehicles. Again, resist the temptation to take shortcuts. I would initially do this filtering in 3 separate DATA steps: (1) Filter on dead persons. (2) Filter on drivers. (3) Filter on restrained persons. This allows me to understand the impact of each filter and to double check against known percentages. The first 3 DATA steps in Figure 7 show the results. The 33,883 fatalities is a known number. About 60% of the general population is drivers so the reduction to 21,835 seems reasonable. Although not shown in the paper, the reduction to 7,454 is also reasonable compared to the general population. In practice, I would do a PROC FREQ between each of the data steps to validate I am dropping the correct number of observations and to visually double check I am dropping what I think I am dropping. Remember the “Drinking” example where a code of “8” for “not reported” existed without my knowledge. I may need to take corrective action with such a large number of observations coded as “not reported”.

Finally, the fourth DATA step in Figure 7 applies all three filters in one fell swoop. Clearly it is simpler code, easier to read, and evens run nominally faster. However, unless one is very familiar with the data; how do you check that losing 90% of your records is reasonable?



```
data p3;
 set p2;
 if inj_sev=4;
run;

NOTE: There were 76510 observations read from the data set WORK.P2.
NOTE: The data set WORK.P3 has 33883 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.06 seconds


data p3;
 set p3;
 if per_typ =1;
run;

NOTE: There were 33883 observations read from the data set WORK.P3.
NOTE: The data set WORK.P3 has 21835 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds


data p3;
 set p3;
 if rest_use in (2,3,4,6);
run;

NOTE: There were 21835 observations read from the data set WORK.P3.
NOTE: The data set WORK.P3 has 7454 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


data p4;
 set p2;
 if inj_sev=4 and per_typ=1 and rest_use in (2,3,4,6);
run;

NOTE: There were 76510 observations read from the data set WORK.P2.
NOTE: The data set WORK.P4 has 7454 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
```

Figure 7: DATA Step Filtering

ITERATIVE PROGRAMMING SUMMARY

The motivation behind this technique is to focus on one task, get it programmed, get it tested, and then move on. By focusing on one and only one task at a time, I find I do a much better job of testing and almost always find my errors on the spot as I am developing the code. I find I also end up with a much better understanding of, and confidence in, the underlying data.

SIMPLE/REPEATABLE CODING TECHNIQUES

It is hard to imagine that any programming system offers a richer set of tools than SAS. Not only can SAS help solve just about any coding/analytical problem, it often provides many different ways to solve the same problem. As a SAS user, the question becomes, just how many different ways do you need to know to skin a cat? If your goal is to support/teach SAS, than you need to know most, if not all of them. If your goal is to skin the cat as quickly and efficiently as possible, you only need to know one. The best way to avoid making errors when writing code is to use familiar structures so you can keep your eye on the code logic, not the code syntax. The theme of this section is KISS, "**K**eep-**I**ng **S**AS **S**imple".

REUSE OLD CODE AS A PATTERN

The most obvious technique I use to get the code right as quickly as possible is to start with an old program which performs a similar task. I suspect this is a fairly universal approach; but don't underestimate its power in getting both the logic and syntax correct.

BE FAITHFUL TO A SMALL SET OF SAS PROCEDURES

Over the years, many SAS procedures have grown closer and closer to each other (e.g. PROC SUMMARY, MEANS, and to some extent UNIVARIATE). Today, I almost exclusively use PROC SUMMARY unless I need a specific format, feature, or statistic that is not available in Summary. It's not that I think Summary is better than the other two; I just know the syntax inside and out and for the most part don't have to give its usage much thought. I do the same thing with Regressions. Since I seldom need any heavy lifting in this area, I almost exclusively use PROC REG. This concept can be extended to techniques like joining data sets. I use MERGE statements because I find the IN= Option invaluable for checking how many observations are being contributed/ignored from each data set. Others who are fluent SQL users can probably make similar claims for it and should stick to it. My main point is that sticking to one or the other yields a smaller chance of error/lost time than switching back and forth.

BE A SAS DINOSAUR (OR WHY I NEED TO APOLOGIZE TO WARREN REPOLE [4])

Despite peer pressure or the coolness factor to try the latest and greatest new features, stick to what you know if you want to write fast, consistent, working code. There is a reason some methods are considered "tried and true" – they work. Let the other guy experiment with (and debug) new features from the vendor[5]. Very rarely have I ever gotten back the time spent exploring and developing code using new techniques over existing, established ones.

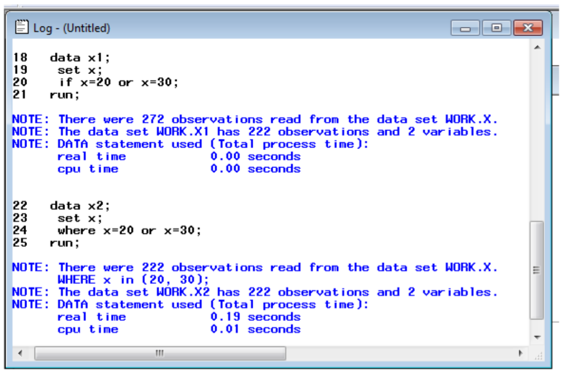
At a previous MWSUG, I learned about "hash tables" [5]. I was so enthused that I immediately recoded several existing data joins (merge) in our production code to use the hash table technique. I had the perfect application: an extremely large table with a plant code and a second smaller table that included the plant code and supporting fields (name, location, etc.). Although I was able to remove a large sort from the code, the overall time difference in a set of jobs that ran only once a month did not even save the time spent in development and test.

My criteria for using new features are simple:

- Does it solve a problem I don't already know how to solve?
- Does it help me eliminate a *meaningful* bottleneck in the throughput of my job?
- Does it give me better feedback on how my processes are running (movement of observation in/out)?

USE "SUB-SETTING IF" STATEMENTS RATHER THAN "WHERE CLAUSES" IN A DATA STEP

To me, this is a no-brainer. I want as much feedback from SAS as possible on how the data is being processed. Take a look at the two simple DATA steps in Figure 8. The difference is subtle, but provides a potentially invaluable clue that you are on/off the right track. It is very clear in the first that I dropped 50 observations. In the second you are clueless. I will discuss this topic in more detail in the section on data filtering, but I believe strongly in monitoring the comings and goings of observations.



```
data x1;
 set x;
 if x=20 or x=30;
run;

NOTE: There were 272 observations read from the data set WORK.X.
NOTE: The data set WORK.X1 has 222 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


data x2;
 set x;
 where x=20 or x=30;
run;

NOTE: There were 222 observations read from the data set WORK.X.
      WHERE x in (20, 30);
NOTE: The data set WORK.X2 has 222 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.19 seconds
      cpu time            0.01 seconds
```

Figure 8: Sub-setting IF in a DATA Step

[5] It took me almost a complete day to write my first SAS program, despite its simplicity (read 10 numbers and perform a PROC FREQ). It turns out that someone had given my JCL to run a test version of SAS rather than the production code. PROC FREQ had a bug. Is it an extreme case? Yes. Did I lose a day? Yes

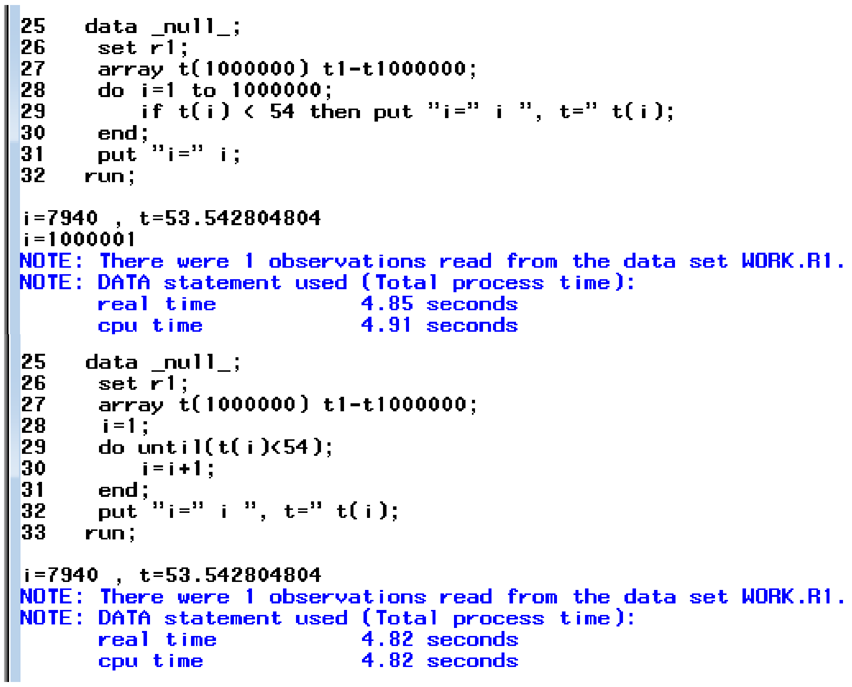
USE SIMPLE CODE WHENEVER POSSIBLE

A number of years ago, I needed to write SAS code to generate throughput and reliability metrics for a large supply chain organization. We designed a dozen or so report templates that needed to be called multiple times for various metrics and geographic/organizational levels. Unfortunately I decided to write a very complicated set of nested macros that could recursively call themselves as needed based on the data. They worked great and, except for maybe when I programmed the graphics chip inside my old Atari 800 directly with machine language, was probably some of the most enjoyable code I have ever written. The rub came 6 months later when another member of my group needed to familiarize himself with the code as I was to be leaving on a short term overseas assignment. Even I had a hard time unraveling the macro nesting after 6 months. We used the nested code for the rest of that year, but I eventually split it into 12 separate programs, each of which was responsible for generating a specific type of template. Run time was longer, but the code was much, much cleaner and easier both for me and others to follow in the future. Elegance and style points do not really count for much in the coding world. The more complicated the code, or even an expression, the more likely you are to get it wrong.

DON'T BE AFRAID TO WASTE THE COMPUTER'S TIME.

Unless the application has been shown to be a bottleneck or timing is mission critical, don't be afraid to waste the computer's time to get simpler code. Modern compilers and processors are amazing. Let them do the work. Even something as simple as using a straight DO LOOP with start and stop values that executes unnecessary loops may be preferable to coding a DO Until/DO While loop for which you, the coder, are responsible for book-keeping the loop increments. The difference in execution time is not nearly as great as you might think and the chance of error, either now or later, are much less with a standard DO LOOP.

Below is an extreme example. I generated 1,000,000 normal random variates (mean=100, standard deviation=10) and stored them in a SAS data set. By inspection, I knew that the break between the smallest and second smallest values was 54 and was the 7,940th generated number in the sequence. I then wrote two different DATA steps to search an array to find and then print any observation less than 54. This first DATA step uses a straight DO LOOP with no branch out; hence it continues to test every array member and technically would print out more than one if it existed. The second DATA step utilizes a DO UNTIL LOOP and stops immediately after finding the first (and in this case only) value less than 54. In an attempt to keep the contest fair, I completely exited SAS between executing the DATA Steps. Figure 9 shows the execution times for each DATA Step. They took essentially the same amount of time despite the huge difference in loop traversals. Is this a function of a fast processor or a smart complier? I don't know, but I also don't care. At the end of the day, the first code block is easier to write and less likely to cause an issue down the road. I've never had an infinite loop with a straight Do Loop.



```
data _null_;
 set r1;
 array t(1000000) t1-t1000000;
 do i=1 to 1000000;
    if t(i) < 54 then put "i=" i ", t=" t(i);
 end;
 put "i=" i;
run;

i=7940 , t=53.542804804
i=1000001
NOTE: There were 1 observations read from the data set WORK.R1.
NOTE: DATA statement used (Total process time):
      real time           4.85 seconds
      cpu time            4.91 seconds

data _null_;
 set r1;
 array t(1000000) t1-t1000000;
  i=1;
 do until(t(i)<54);
    i=i+1;
 end;
 put "i=" i ", t=" t(i);
run;

i=7940 , t=53.542804804
NOTE: There were 1 observations read from the data set WORK.R1.
NOTE: DATA statement used (Total process time):
      real time           4.82 seconds
      cpu time            4.82 seconds
```

Figure 9: DO LOOP versus DO UNTIL

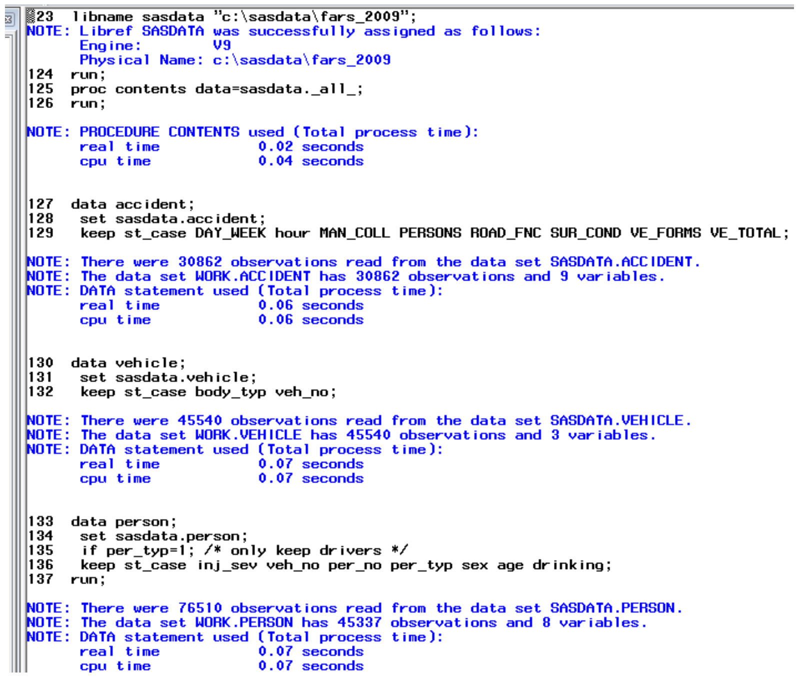
DATA FILTERING

Data filtering occurs in two flavors. One can sub-set an existing table via a “sub-setting if” or “where clause” or one can drop observations from one or more tables as they are being joined. I feel very strongly that one needs to track the comings and goings of observations much like the police track their chain of evidence. Over the years, I have adopted a tree diagram approach for documenting the evidence chain (simplified data flow diagram). In this section, I will again utilize data from the 2009 Fatal Accident Reporting System (FARS) to demonstrate the technique.

FARS is managed by the National Highway Traffic Safety Administration and contains detailed records on every fatal motor vehicle accident that has occurred in the United States since 1975. The FARS data is available to the public through NHTSA’s web-site in a variety of formats, including SAS data sets. FARS records for a given calendar year are stored in three primary tables: (1) Accident File, (2) Vehicle File, and (3) Person File. Each fatal accident has its own unique identifier which allows one to pick and choose variables of interest amongst the three different tables for a given analysis.

For this example, let us assume we would like to build an analysis data set that contains teen-age drivers of passenger cars where the crash occurred between 11:00 pm and 3:00 am. As we are interested in the type of highway, weather conditions, vehicle type, single versus multi vehicle accident, etc.; we will need to pull key fields from each of the three main tables and then join them into one table for the analysis. What makes life interesting is that the accident file has just one record per accident, the vehicle file has one record per reported vehicle, and the person file has one record per person involved in the accident.

Figure 10 shows DATA steps to pull each of the three main tables and restrict the variables to those of interest. Since we are only interested in drivers, I further restricted the PERSON file to drivers only. As we have seen earlier, NHTSA does a great job of publishing traffic facts so the expected totals from each of these data sets is easily validated. Notice that I used a “sub-setting if” to filter out non-drivers. This allows me to validate both the total number of persons and the total number of drivers. A “where clause” would only allow me to check the number of drivers.



```
libname sasdata "c:\sasdata\fars_2009";
NOTE: Libref SASDATA was successfully assigned as follows:
      Engine:        V9
      Physical Name: c:\sasdata\fars_2009
run;
proc contents data=sasdata._all_;
run;

NOTE: PROCEDURE CONTENTS used (Total process time):
      real time           0.02 seconds
      cpu time            0.04 seconds


data accident;
 set sasdata.accident;
 keep st_case DAY_WEEK hour MAN_COLL PERSONS ROAD_FNC SUR_COND VE_FORMS VE_TOTAL;

NOTE: There were 30862 observations read from the data set SASDATA.ACCIDENT.
NOTE: The data set WORK.ACCIDENT has 30862 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.06 seconds


data vehicle;
 set sasdata.vehicle;
 keep st_case body_typ veh_no;

NOTE: There were 45540 observations read from the data set SASDATA.VEHICLE.
NOTE: The data set WORK.VEHICLE has 45540 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.07 seconds


data person;
 set sasdata.person;
 if per_typ=1; /* only keep drivers */
 keep st_case inj_sev veh_no per_no per_typ sex age drinking;
run;

NOTE: There were 76510 observations read from the data set SASDATA.PERSON.
NOTE: The data set WORK.PERSON has 45337 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.07 seconds
```

Figure 10: DATA Steps to Load FARS Tables

Figure 11 shows the start of the data tree diagram which will depict how observations enter and leave our study data. There is a bubble for each of our three main tables, each with a label and starting number of observations. In addition, it captures the results of the first filtering operation – sub-setting the person file to drivers only.

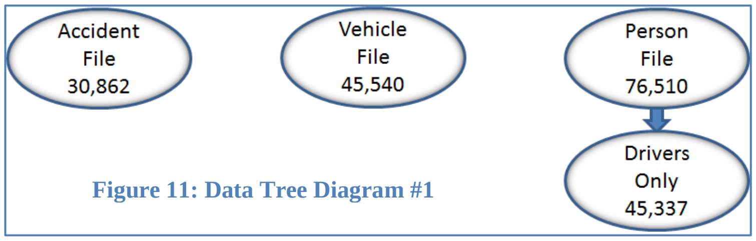


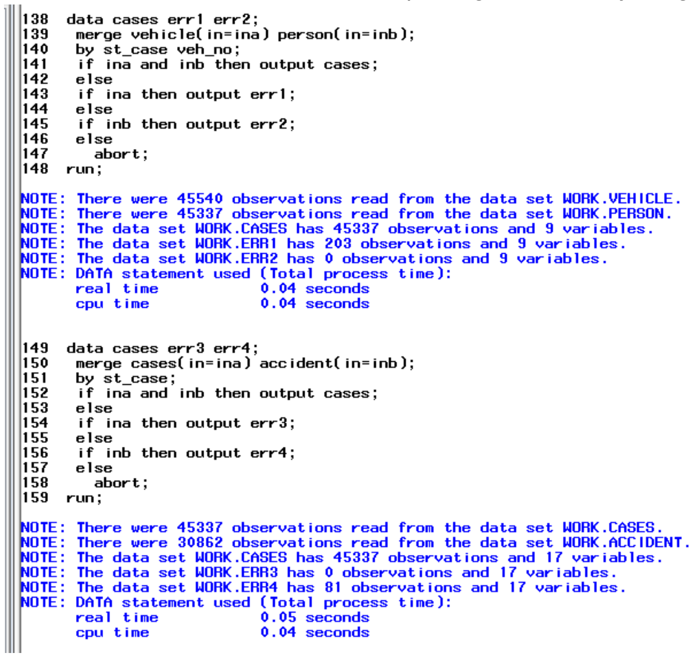
Figure 11: Data Tree Diagram #1

Notice that we have about 200 more vehicles than we have drivers. This is not unusual with this data. Drivers do head for the hills before the police arrive. On average, we have about 1.5 vehicles per accident. So, what is the best way to join these three tables? In addition to the CASE ID, both the vehicle file and the person file have a VEH NO ID which will allow us to match up a driver to his vehicle. In order to keep an accurate record of observations comings and goings, I will first join the vehicle and driver tables and then join the resulting table with the accident table. Figure 12 on the next page shows both the SAS code for joining the tables and the updated tree diagram.

When joining tables in SAS, I always use the MERGE statement and the IN= option. This allows me to keep track of which data sets are contributing to the new combined data set. In the first DATA step in Figure 12, I am joining the vehicle and person table. Only records with both a vehicle and matching driver make it into my case file. You can see from the notes that there were 203 cases where we had a vehicle with no driver. All driver records had a corresponding vehicle record.

Now we are at a cross-road. I can either let the 203 vehicles drop on the floor; or, I can re-write the join to keep them (in which case I would add an indicator variable to the output data set reflecting that the person record was missing). Since the volume is so low, I will let them drop on the floor for this exercise.

The second Data Step in Figure 12 shows the results of joining the combined file from the vehicle and person file with the accident file. Every vehicle-driver combination has a corresponding accident record; however, there were 81 accident records that did not have a vehicle-driver combination. If I were doing a real study, I would bump the 81 accident records up against the 203 vehicle records which did not have a driver record for completeness.



```
data cases err1 err2;
  merge vehicle( in=ina) person( in=inb);
  by st_case veh_no;
  if ina and inb then output cases;
  else
  if ina then output err1;
  else
  if inb then output err2;
  else
    abort;
run;

NOTE: There were 45540 observations read from the data set WORK.VEHICLE.
NOTE: There were 45337 observations read from the data set WORK.PERSON.
NOTE: The data set WORK.CASES has 45337 observations and 9 variables.
NOTE: The data set WORK.ERR1 has 203 observations and 9 variables.
NOTE: The data set WORK.ERR2 has 0 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds


data cases err3 err4;
  merge cases( in=ina) accident( in=inb);
  by st_case;
  if ina and inb then output cases;
  else
  if ina then output err3;
  else
  if inb then output err4;
  else
    abort;
run;

NOTE: There were 45337 observations read from the data set WORK.CASES.
NOTE: There were 30862 observations read from the data set WORK.ACCIDENT.
NOTE: The data set WORK.CASES has 45337 observations and 17 variables.
NOTE: The data set WORK.ERR3 has 0 observations and 17 variables.
NOTE: The data set WORK.ERR4 has 81 observations and 17 variables.
NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      cpu time            0.04 seconds
```

Figure 12: Joining the Primary SAS Tables

Figure 13 shows an updated tree diagram with the results from the two joins described in Figure 12. Included is a new bubble showing the results from joining the vehicle and person files and one showing the result of bringing the accident file into the mix. In addition, I capture the number of observations lost at each stage with a brief description as to why they were dropped. Note that the counts in the tree diagram merely summarize the same information found in the SAS notes; however, I find it more convenient as all of the information is pulled together in one location.

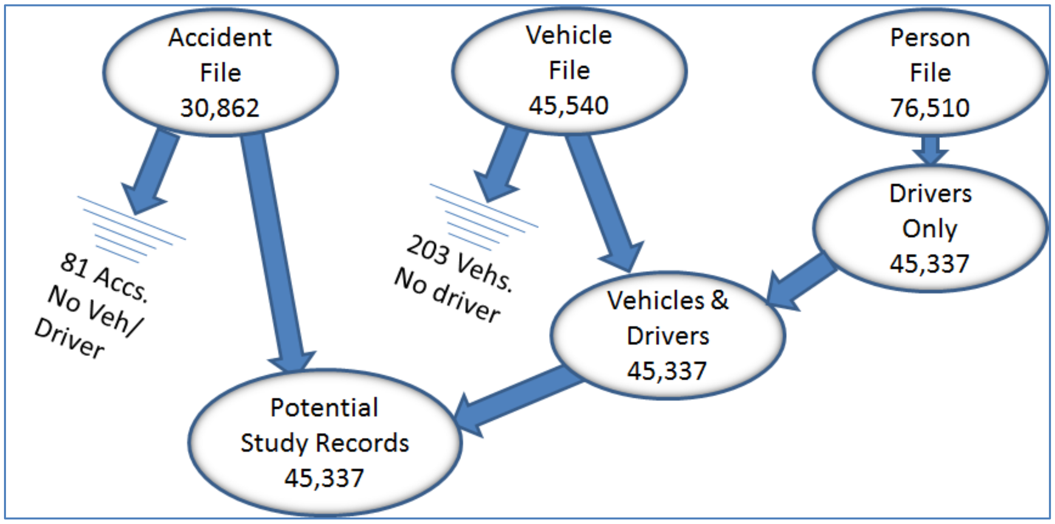
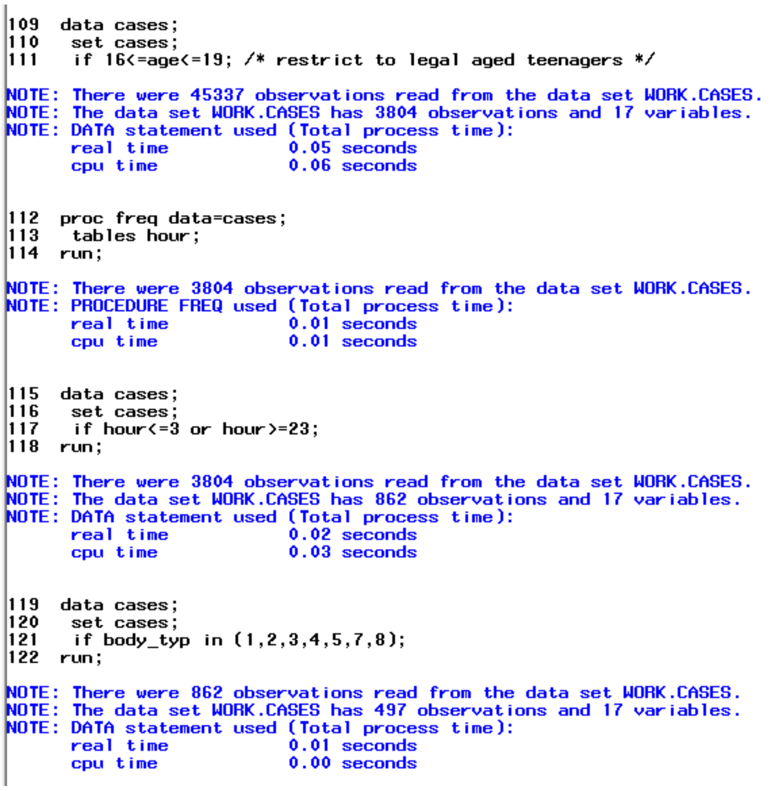


Figure 13: Data Tree Diagram #2

In the previous step, we joined the accident, vehicle, and driver records to form a potential study set. However, as is, it includes all accidents, vehicle types (car, trucks, motorcycles, etc.) and drivers. Our requirements were to study teenagers involved in a fatal crash that occurred between 11:00 p.m. and 3:00 a.m. who were driving a passenger car. Hence we need to filter our data based on these variables. In keeping with the philosophy of maximum visibility to dropped observations, I applied each filter in a separate DATA Step as shown in Figure 14. Although not shown, I checked the number of missing values for each variable. As the volume was relatively small for each, I let them drop on the floor for this exercise. The PROC FREQs between the DATA Steps allow me to double check that I am dropping the correct number of records. (I removed the PROC FREQ for "body type" to save space.)



```
data cases;
 set cases;
 if 16<=age<=19; /* restrict to legal aged teenagers */

NOTE: There were 45337 observations read from the data set WORK.CASES.
NOTE: The data set WORK.CASES has 3804 observations and 17 variables.
NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      cpu time            0.06 seconds


proc freq data=cases;
 tables hour;
run;

NOTE: There were 3804 observations read from the data set WORK.CASES.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


data cases;
 set cases;
 if hour<=3 or hour>=23;
run;

NOTE: There were 3804 observations read from the data set WORK.CASES.
NOTE: The data set WORK.CASES has 862 observations and 17 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.03 seconds


data cases;
 set cases;
 if body_typ in (1,2,3,4,5,7,8);
run;

NOTE: There were 862 observations read from the data set WORK.CASES.
NOTE: The data set WORK.CASES has 497 observations and 17 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

Figure 14: DATA Step Filtering

Figure 15 shows the final tree diagram that results from our 3 data filter steps. As always, it keeps track of the number of records moving forward after each step as well as the number dropped along the way.

Although there is some extra work in separating the filters into multiple joins and data steps and documenting the data flow; it is a powerful tool in determining if you have solid data for an analysis. In the Introduction Section, I described a life-changing experience I had early in my career. As previously stated, the error actually occurred in the other study, not ours. The interesting part of the story is that we were able to determine the issue with the other folk's SAS code for them without ever seeing their code.

By sheer coincidence, we had just recently published a paper on motorcycle helmet effectiveness; hence, I had a tree diagram for the corresponding data filters from that study. It turns out that when we compared frequency counts between the two seat belt studies, they had a much larger number of deaths for unbelted passengers in the rear-left seat of the cars. The difference struck a chord with me and turned out to be very close to the number of dead motorcycle passengers we had observed – which coincidentally end up being coded in FARS as rear-left seat, unbelted passenger (helmet use is a different value than seat belt use.) Ultimately, it was shown that motorcycle occupants had inadvertently not been filtered out of the other seat belt study.

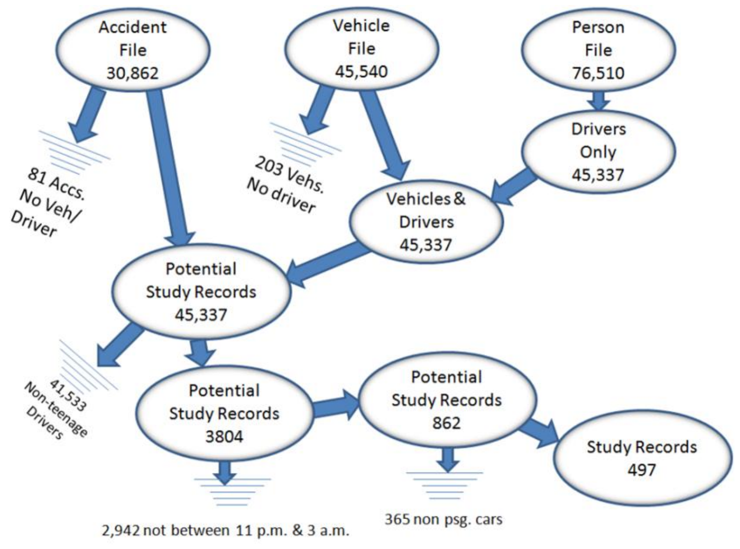

Accident File 30,862
Vehicle File 45,540
Person File 76,510
Drivers Only 45,337
81 Accs. No Veh/ Driver
203 Vehs. No driver
Vehicles & Drivers 45,337
Potential Study Records 45,337
41,533 Non-teenage Drivers
Potential Study Records 3804
Potential Study Records 862
Study Records 497
2,942 not between 11 p.m. & 3 a.m.
365 non psg. cars

Figure 15: Data Tree Diagram #3

SAS AS AN ITERATIVE PLATFORM

I will show my age a second time here. At last count, I have written code in 25 different languages (not counting database access software, graphics libraries, etc.). They range from old fashioned assembled code (I even got to write and enter machine code with binary switches on the front panel of the machine) to newer compilers with interactive debuggers. I can truthfully say that I have yet to code in another piece of software that is as useful as I found SAS the first time I used it in 1982.

As you have seen, my programming model is to write small chunks of code, test them, and then move on. Certainly, this iterative technique can be implemented in other environments; but it is the combination of on demand runs with easily obtained frequency counts and invaluable "notes" that makes SAS stand out in my mind. And despite what many of my IT colleagues have told me over the years, I have yet to run into an IT problem that I cannot solve in SAS. Not saying they aren't out there, but the types of problems I have worked on over the years have all been solvable in SAS. I have used other software over the years to be sure, but that is primarily due to a lack of SAS software at customer locations.

I once developed a prototype for a manufacturing company that analyzed recent sales history for a market and recommended stocking levels to local distributors. There were 10 modules – all SAS. Prior to the system being placed into production, the IT department re-coded all 10 modules in PL/I (yes – this was a few years ago). Despite having a working prototype and a staff of programmers, it took them longer to recode the system then it did for me to write it in the first place. Why? Not skill level. I had worked with this team before and had a high opinion of them. I believe the difference was SAS versus PL/I. Since SAS and PL/I syntax are very similar, I submit the difference is the programming environment.

At the end of the day, access to a full-featured programming language and built in functions with superb interface (PROCS) coupled with the ability to write software iteratively make the SAS software system an ideal choice to prepare data for subsequent statistical analysis. Oh yeah, you get great statistical functions in the same package.

SUMMARY

In this paper, I have demonstrated techniques that I have used over the last 30 years aimed at "Getting the Code Right the First Time." In summary, I would like to leave you with the following thoughts:

1. Do not write more code at one time prior to testing than you can visualize in your "mind's eye." It will save time in the long run.
2. Keep your code as simple as possible. Style points do not count in production code.
3. Be ferocious in tracking down why observations enter/leave your analysis data set. More times than not, coding errors will manifest themselves as erroneous data gain/loss.

REFERENCES

[1] Glantz Stanton A (1980), Biostatistics: how to detect, correct and prevent errors in the medical literature. *Circulation*. 1980; 61:1-7.

[2] Excel (*Part of Microsoft Office Professional Edition)* [computer program]. Microsoft; 2010.

[3] *Traffic Safety Facts – Research Notes, 2010 Motor Vehicle Crashes: Overview*. National Highway Traffic Safety Administration, revised Feb 2012

[4] Repole. (2007), Modernizing Your SAS Code, or How to Avoid Becoming a SAS Dinosaur, *Proceedings of the Eighteenth*

Midwest SAS Users Group Conference, paper SAS-08.

[5] Secosky Jason, Bloom Janice (2007). Getting Started with the DATA Step Hash Object. *Proceedings of the Eighteenth Midwest SAS Users Group Conference.* paper SAS08.

CONTACT INFORMATION

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

Name: Michael C. Frick
Enterprise: General Motors, Retired
Address: 30238 Underwood Drive
City, State ZIP: Warren, MI 48092
Work Phone: 586-573-0977
Fax: N/A
E-mail: mcfdaf001@yahoo.com
Web: N/A