PharmaSUG 2017 - Paper TT08

# Clinical Trials Data: It's a Scary World Out There!

# (or "Code that Helps You Sleep at Night")

Scott Horton, United BioSource Corporation

## ABSTRACT

Clinical trials data can be tough to manipulate and summarize.

Sometimes the problems that data can cause are not obvious when SAS® code runs clean and perusal of large data sets does not reveal any anomalies.

Text truncation, Merging data, Coded data, Unexpected dates, Duplicate data, Data set naming, Macro parameters, Select statement, Array statement will be covered.

Most of these tricks are a few lines of code. Very little effort that continues to pay dividends without too much up front cost.

All of the tricks are straightforward as well; even beginning programmers can implement them.

## INTRODUCTION

Clinical trials data can be tough to manipulate and summarize. Sometimes the problems that data can cause are not obvious when code runs clean and perusal of the large data sets does not reveal any anomalies. Problems can occur due to:

1) DCT/CRF design may have been poorly designed,
2) DCT/CRF design may have changed during study execution (e.g., due to protocol amendment),
3) Sites closed prior to data being fully cleaned,
4) Data could be illogical but no update can be done (e.g., source data does not support changing it),
5) Data are not final (e.g., Interim Analyses, DMCs/DSMBs, Annual Reports).

These are among the many issues that can result in "dirty data" and produce outputs that are unacceptable but not easily detectable. Programs may have been running without issue for prior deliveries, but now may need updates that you are unaware of.

Areas that will be covered in this presentation:

- Text truncation
- Merging data
- Coded data
- Unexpected dates
- Duplicate data
- Data set naming
- Macro parameters
- Select statement
- Array statement

Most of these tricks are a few lines of code. Very little effort that continues to pay dividends without too much up front cost. All of the tricks are straightforward as well; even beginning programmers can implement them.

## TEXT TRUNCATION

SAS® provides some simple functions that perform concatenations:

- CAT - concatenates entire text value of variable or constant
- CATS - concatenates after removing leading and trailing blanks
- CATT - concatenates after removing trailing blanks
- CATX - concatenates with a specified delimeter (1+ characters) removing leading and trailing blanks

During concatenation, all four of these functions automatically detect any truncations that occur *(i.e., code that breaks… …and that is good!).* These functions assume the variable receiving the concatenated result has already been assigned a variable length. The example presented will just use CATX function.

Some clinical data may be stored in variables that have a length greater than 200 characters when being exported from a clinical database. However, given the CDISC standards that we operate within, storing such data in variables of length $200 is required. We want to create data set variables of length $200 where:

1) The actual data values fit into a single variable of length $200
2) The actual data values require two or more variables of length $200.

### DETECTION OF TEXT TRUNCATION

We will use CATX function and detect if single variable of length $200 is sufficient.

```
LENGTH C200Variable $200;
C200Variable = CATX(' ',BigTextVariable);
```

If data (ignoring leading or trailing blanks) that is stored in BigTextVariable is greater than 200 characters, a warning will be written to LOG file that indicates number of characters being truncated. Let's look at an example.

Variable BIG has a length $500 and is populated with 'x' in each of the first 188 positions and has blanks for rest of value. When processing data via the DATA step below, we find that the values stored in VAR1, VAR2, VAR3, and VAR4 are identical.

```
DATA two;
   SET one;
   LENGTH var1 var2 var3 var4 $200;
   var1=big;
   var2=STRIP(big);
   var3=CATX(' ',big);
   var4=CATX(' ',STRIP(big));
run;
```

But what if variable BIG contained 211 characters of 'x' in the first 211 positions of BIG? Running same code above results in the following in the LOG file:

*WARNING: In a call to the CATX function, the buffer allocated for the result was not long enough to contain the concatenation of all the arguments. The correct result would contain 211 characters, but the actual result might either be truncated to 200 character(s) or be completely blank, depending on the calling environment. The following note indicates the left-most argument that caused truncation.*

LOG file indicates exact length needed due to only first value that is too long.

It is important to note that if variable being assigned the concatenated result is also included in the items being concatenated then a WARNING will not be written to the log file. An example of this is provided in the DATA step below.

```
DATA two;
   SET one;
   var3=CATX(' ','no warning written',var3);
run;
```

## MERGING DATA

Merging is one of the most common activities in the manipulation of clinical trials data. Are there little bits of code that will help navigate this avenue of data manipulation? Yes! A couple examples of approaches to detecting problems with the data are:

- Detect when expected extra records per subject are found
- Detect when important data are missing

### CODE TO IDENTIFIY MERGING ISSUES

Below are three DATA steps that address both of these issues. The PUT statement below (and throughout this paper) that splits WARNING into two pieces is used to have WARNING written to the LOG file as one word only if a problem in the data is detected. This is done so that tools that parse the log file do not detect WARNING when just writing the actual code to the LOG file.

```
DATA final dupsubs no_dm_random;
   MERGE dm(IN=indm) random(IN=inrandom) baseheight baseweight;
   BY subjid;

   *** DETECT SUBJECTS WITH 2+ RECORDS ***;
   IF not (first.subjid and last.subjid) THEN output dupsubs;

   *** DETECT SUBJECTS MISSING DEMOGRAPHIC AND/OR RANDOMIZATION DATA ***;
   IF sum(indm,inrandom) lt 2 THEN output no_dm_random;
   OUTPUT final;
run;

DATA _null_;
   SET dupsubs END=last;
   PUT 'WARN' 'ING - ' subjid ' HAS 2+ RECORDS';

   *** STOP EXECUTION ***;
   IF last THEN abort;
run;

DATA _null_;
   SET no_dm_random;
   PUT 'WARN' 'ING - ' subjid
    ' MISSING EITHER DEMO OR RANDOM DATA';
run;
```

## CODED DATA

For analyses that are conducted during a clinical trial (e.g., interim analyses, DMCs, annual reports), not all terms (e.g., adverse events, concomitant medications) may be coded at the time of a delivery. Flagging any uncoded terms without searching for them manually will pay dividends through repeated analyses. This detection of uncoded terms can be done via a PUT statement that writes a WARNING to the LOG file.

### CODE TO IDENTIFIY UNCODED TERMS

The key line of code in the DATA step below is the IF-THEN statement. This does not have to be a "stand alone" DATA step; the IF-THEN statement can be included in a DATA step with many other statements.

```
DATA adae1;
   SET adae;
   IF aebodsys eq ' ' THEN
      PUT 'WARN' 'ING - ' subjid  ` WITH UNCODED AE (' aeterm ') ';
run;
```

The result of detected terms that are not coded is a WARNING being written to the LOG file.

### PRESENTATION OF UNCODED TERMS IN TABLES AND LISTINGS

Presentation of uncoded terms in tables and listings can be handled with different approaches; some approaches may be sponsor specific or even project specific in nature. The example given below is for adverse events but can be extrapolated to other types of data that is coded.

The text of "Uncoded" can be assigned into the variable AEBODSYS (which contains the coded term for the MedDRA System Organ Class of an adverse event) for display in a table. It is common in the case of an uncoded adverse event for the text recorded by the clinical site to be displayed in AEDECOD (which contains the MedDRA Preferred Term of an adverse event). Another example of what could be assigned to AEBODSYS for uncoded adverse events is "Uncoded (verbatim term listed below)".

Here is some example code.

```
DATA aelist1;
   SET aelist;
   IF aebodsys eq ' ' THEN do;
      aebodsys='Uncoded (verbatim term listed below)';
      aedecod=strip(aeterm);
   END;
run;
```

If you want to have the uncoded terms appear at the top of tables that include this type of data, assignment of an underscore (e.g., '_Uncoded' or '_Not coded') will cause this data to sort to top of table when the table is sorted alphabetically. If the presentation order is based upon the frequencies of coded terms then the creation of a sorting variable will give the desired presentation.

## UNEXPECTED DATES

Clinical trials data often contains dates which are restricted in what is a legal value—that is, a date may be impossible based on its definition though the value of the date may be a legal value otherwise.

### DATES RESTRICTED BY RELATIONSHIP TO THE START OF TREATMENT DATE

An example of date values that cannot occur based on its definition is the date of cancer recurrence after the start of study treatment. A PUT statement can alert a programmer to inappropriate date values that otherwise would be legal date values.

```
DATA two;
   MERGE dm recurr;
   BY subjid;
   IF n(recurrdt) and recurrdt lt trtsdt THEN
      PUT 'WARN' 'ING - ' subjid  ' RECURRENCE DATE(' recurrdt
         ') IS PRIOR TO START OF STUDY DRUG('  trtsdt ') ';
run;
```

The result of recurrence dates being detected that are prior to the start of study treatment is a WARNING being written to the LOG file.

## DUPLICATE DATA

Clinical databases may allow clinical sites to enter the same data twice. This duplication can be due to entry screens designed for multiple records to be entered where the same exact data is entered a second time. A second way duplication can occur is when a clinical site enters the same data under two different protocol visits. Below we will present two ways that such duplicate data can be handled.

### DISCARD DUPLICATE DATA

Discarding of duplicate data may be desires so that resulting summary data makes sense (e.g., the n for a mean is not greater than the population N).

```
PROC SORT DATA=one OUT=two NODUPKEY DUPOUT=dups;
   BY _all_;  *** OR BY JUST CERTAIN UNIQUE KEYS ***;
run;

DATA _null_;
   SET dups;
   PUT 'WARN' 'ING - ' subjid ' DUPLICATE RECORDS IN DATA ONE';
run;
```

### KEEP DUPLICATE DATA

You may want to keep all records in the output for a reviewer to see (instead of just sending duplicates to data management personnel for querying).

```
PROC SORT DATA=one;
   BY subjid keyvar2 keyvar3;
run;

DATA two;
   SET one;
   BY subjid keyvar2 keyvar3;
   IF not (first.keyvar3 and last.keyvar3) THEN
      PUT 'WARN' 'ING - ' subjid
         ' HAS DUPLICATE RECORDS IN DATA ONE for keyvar2('
         keyvar2 ') and keyvar3(' keyvar3 ') ';
run;
```

## DATA SET NAMING

Using the same name for a DATA set is syntactically correct. However, when code reuses a DATA set name review of a LST file can be confusing. More important, in interactive SAS® the earlier versions of the data are not available for review if running the entire program. A good approach to eliminate such potential confusion entails always using unique DATA set names in the same program.

### NON-UNIQUE DATA SET NAMING

The ellipses (…) below just represent statements which manipulate data values that might be included in a DATA step. In the example below, it is not immediately obvious which version of DATA set LAB is being reviewed in the LST file.

```
DATA lab;
   MERGE lab demo;
   BY subjid;
   …
run;

title 'DATA LAB';
PROC PRINT;
run;

PROC SORT;
   BY subjid randdt;
run;

DATA lab;
   SET lab;
   BY subjid randdt;
   …
run;

title 'DATA LAB';
PROC PRINT;
run;
```

### UNIQUE DATA SET NAMING

Reviewing the LST file is always clear with the code below.

```
TITLE 'DATA LAB';
PROC PRINT DATA=lab;
run;

DATA lab1;
   SET lab;
   …
run;

TITLE 'DATA LAB1';
PROC PRINT DATA=lab1;
run;
```

### WHEN UNIQUE DATA SET NAMING MAY BE A CONCERN

When manipulating large datasets repeatedly in a program, you may need to consider using the same DATA set name due to computer memory constraints; or, to maintain unique naming of DATA sets, you can consider using the DATASETS PROCEDURE to delete work DATA sets that are not needed later in the program. In cases where you do reuse the same DATA set name, you can add some descriptive text (e.g. a unique algorithm performed in that particular DATA step) in the title statement used in conjunction with a PROC PRINT.

A rare case that you might encounter is that extremely large DATA sets may require using the same DATA set name. An example of this using a PROC SORT where renaming the resulting sorted DATA set name must be the same DATA set name.

## MACRO PARAMETERS

How much information can be communicated to a programmer who takes over the maintenance of a program (or even to the original author when a significant amount of time has elapsed) by just looking at

individual macro calls? Below are three calls where a review might indicate what is being performed, but may not give a programmer full confidence on initial review.

```
%dolisting(1,Adverse Events,saffl eq 'Y')

%dolisting(2,Adverse Events Leading to Permanent Discontinuation,
   saffl eq 'Y' and index(upcase(aeacn),'WITHDRAW'))

%dolisting(3,Serious Adverse Events,
   saffl eq 'Y' and upcase(aeser) eq 'YES')
```

%dolisting? The purpose of the macro seems to be obvious: creating a listing. But what about the first positional parameter in each of the calls above? Does if reflect the precision of display for some data? Is it part of some text? Is it part of the listing title? Or maybe it is used to identify the title of the listing?

The use of keyword parameters instead of positional parameters allow more information about a particular macro call to be more readily identified.

```
%dolisting(listnum=1,listttl=Adverse Events,subset=saffl eq 'Y')
```

LISTNUM? The digit identifies which listing is being created. This might be the actual number of the listing or a reference used to obtain the listing number. LISTTTL? This is the title of the listing. SUBSET? This is an expression used to obtain the desired subset of the data that will be included in the listing.

Of course, commenting macro calls adds to the visual processing of a program. Keyword macro parameters also allow a default value to be specified as well. In the first example below, having default value of 1 for the SUBSET macro parameter would result in no subsetting of the data being processed (e.g., based on macro code that uses the value of SUBSET in a subsetting IF statement or in a WHERE statement).

```
%macro dolisting(listnum=,listttl=,subset=1)

*** ADVERSE EVENTS LISTING ***;
%dolisting(listnum=1,
   listttl=Adverse Events)

*** SERIOUS ADVERSE EVENTS LISTING ***;
%dolisting(listnum=3,
   listttl=Serious Adverse Events,
   subset=saffl eq 'Y' and upcase(aeser) eq 'YES')
```

## SELECT STATEMENT

An IF-THEN/ELSE statement might be the most used statement in DATA steps that create or manipulate data; so why might you consider using a SELECT statement instead. One result of an IF-THEN/ELSE statement can be somewhat hidden to the casual review of code. We will compare the results of such code versus a similar code using a SELECT statement.

### CODE THAT ASSUMES TWO VALUES FOR A VARIABLE

Below is an IF-THEN/ELSE statement that seems to do exact what we want: convert a numeric value to a corresponding text value.

```
IF colorn eq 1 THEN color='Red';
ELSE color='Blue';
```

This code assumes that only values of 1 and one other value (e.g., 2) exist in the data for COLORN. That seems reasonable based on what should be in the data—but maybe that is not specific enough.

```
IF colorn eq 1 THEN color='Red';
ELSE IF colorn eq 2 THEN color='Blue';
```

Instead of just assuming a value of 2 for Blue, we want to check for that. This code results in missing values when COLORN is either missing or contains a value other than 1 or 2—and the LOG file doesn't complain about this. This may not be exactly how you want to process the data.

### SELECT STATEMENT ('BREAKS' BY DESIGN)

The SELECT statement fails when encountering unexpected values—that is, the code break by design… …and that is what triggers you to address potentially needed updates.

```
SELECT (colorn);
   WHEN (1) color='Red';
   WHEN (2) color='Blue';
END;
```

The code above will write an ERROR to LOG file for first value of COLORN other than 1 or 2 (including missing values). If an ERROR is written to the LOG file, then you can determine whether a potential update should be made or information should be relayed back to data management personnel.

If you want your code to not encounter an ERROR but still handle values of 1 and 2, ignore missing values, and still keep you posted on other data values by writing the text WARNING to the LOG file, you might consider this:

```
SELECT (colorn);
   WHEN (1) color='Red';
   WHEN (2) color='Blue';
   WHEN (.) ;
   OTHERWISE PUT 'WARN' 'ING - UNEXPECTED VALUE ' colorn=;
END;
```

You can accomplish this same algorithm with multiple IF-THEN/ELSE statements:

```
IF colorn eq 1 THEN color='Red';
ELSE IF colorn eq 2 THEN color='Blue';
ELSE IF colorn ne . THEN PUT 'WARN' 'ING - UNEXPECTED VALUE ' colorn=;
```

This is a simple example. But once you have more than a couple of different values, the SELECT statement is visually easier to decipher in regards to what values are actually being handled. If no OTHERWISE is used, then code automatically trips over new values not handled by code. The SELECT statement is not limited to testing the values of a single variable (per above), but can also test compound expressions as well.

```
SELECT;
   WHEN (colorn eq 1) color='Red';
   WHEN (colorn eq 2 and day eq 'Friday') color='Blue';
   WHEN (thispresentationistoolong eq 'TRUE') ;
   WHEN (nmiss(colorn)) ;
   OTHERWISE
      PUT 'WARN' 'ING - UNEXPECTED COMBINATION OF VALUES '
         colorn= day= thispresentationistoolong=;
END;
```

## ARRAY STATEMENT

By design, the DATA step allows by-observation processing to occur as you process statements in a DATA step for each observation of the data (i.e. implicit looping). ARRAY statements allow multiple variables to be similarly processed on the same observation. In our example below we will use a

temporary array instead of one that uses actual variables. The goal of the code is to slot subjects into time-to-event time slots; and these slots are cumulative slots. The resulting data will show how many subjects reached an event with the passage of time.

```
ARRAY slots{5} _temporary_ (1 3 6 12 18);
DO i=1 TO dim(slots);
   IF n(tte) and tte le slots[i] THEN do;
      slotn=slots[i];
      output;
   END;
END;
```

But what if the final number of slots to consider was unknown at the original creation of the code? Or the number of slots changed after initial code creation? We could create a large array that would handle a lot of possibilities (*ellipses in the code below, of course, are not syntactically correct—they are just used to indicate exact values would replace them*); the rest of the code would remain the same.

```
ARRAY slots{…} _temporary_ (1 3 6 12 18 24 30 36 …);
```

A more robust solution that would trigger you when the code is inadequate due to new data values. It would be placed outside of the DO-LOOP.

```
IF tte gt slots[dim(slots)] THEN
   PUT 'WARN' 'ING – TIME TO EVENT TOO LARGE FOR ARRAY SLOTS ' tte=;
```

## CONCLUSION

Though clinical trials data can be tough to manipulate and summarize due to unexpected values, implementation of some very simple lines of code will pay ongoing dividends. Very little effort is required to implement such code since they are straightforward and can be used by programmers of any experience level.

Implementing code that assumes some data anomalies will occur (without going overboard to cover all imaginable problems) will:

- *Ensure quality* in future repeat deliveries for the same study
- *Ensure smoother transitions*
    - New programmer joining the study team
    - New programming lead for the study
- Make you look good as others view or use your code
    - Alerts them (or you) to problems in a timely fashion
- Help you sleep at night!

## ACKNOWLEDGMENTS

Benjamin Young, Experis

## CONTACT INFORMATION

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

Scott Horton
scott.horton@ubc.com