Using Format Concatenation in SASÒ Software to Decode Data in Longitudinal Studies

Perry Watts, IMS Health, Plymouth Meeting, PA

Abstract

Formatting data becomes a complex process when codes change their meaning over time. While new codes can be easily accommodated, deletions and modifications require multiple time-dependent versions of a format for accurate decoding. All these versions must be available to the programmer who works with longitudinal data.

This paper uses the concatenation feature in the FORMAT procedure to eliminate redundancy from versioned formats. The author shows how to create the formats and then how to use them in a sequentially linked manner so that time is incorporated as a dimension into the decoding event. Efficiency issues are also considered in the presentation. For a description of format concatenation, the reader is referred to Jack Shoemaker's paper *Advanced Techniques to Build and Manage User-Defined SASÒ FORMAT Catalogs* that appeared in the NESUG 1998 Conference Proceedings.

Problem Definition

Coded values in the health care industry are constantly changing. The development of new drugs generates new codes. A zip code for a hospital may change from one year to the next. A given illness may be subsumed under a different diagnostic classification such that the original code ceases to exist. In short all types of edits - add, modify, and delete are required for the accurate decoding of longitudinal data.

The problems only get worse when the number of formats used for decoding increases or when they are updated on a quarterly instead of an annual basis. In addition, if formats interact with each other, they must be updated in synchrony.

Formats can interact with each other in several ways. First, a single code may yield multiple decoded values. For example, the code *111222333* might represent the **drug** *XYZZY* and a **dosage** of *twice per day*. Secondly, the output of one format can be used as input to another. In this instance, one needs to nest function calls to obtain a correctly formatted value. The value for *x* below depends on both *afmt* and *bfmt* being accurate for the time period in which the variable *a* is created.

```
    x = put(put(a,afmt.),$bfmt.);
```

If format interaction exists in a programming environment, then all formats should have access to their time-dependent antecedents. Unfortunately, such open access requires a lot disk space. Format concatenation addresses this problem by removing all redundancy from versioned formats.

Description of the Formats

Diagnostic codes updated annually from 1990 through 1999 are diagrammed prior to concatenation in Figure 1.

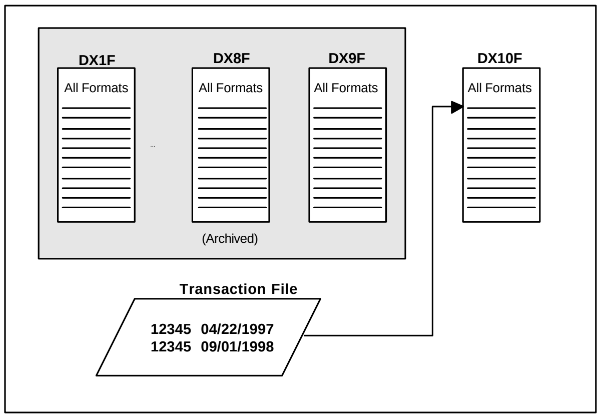

Figure 1. Ten DX formats for the years 1990 to 1999.

Only the current version of the format is used for decoding diagnoses in a transaction file. There is no way to link format name to the year of the update. Earlier versions of such a format are typically archived to save disk space.

Figure 2 shows a diagram of the same formats after concatenation.

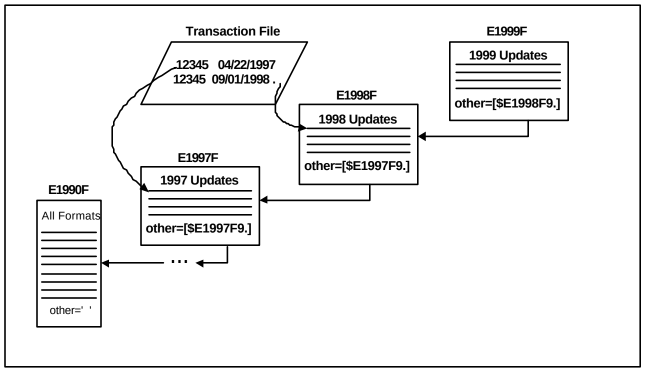

Figure 2. Concatenated formats communicate with each other by using their antecedents as labels. Such formats on the right side of an equation are known as *embedded* formats.

The dated records in the transaction file depicted in Figure 2 can be matched to the appropriate format with the *putc* function that links the data-year to the year sub-string in the format name. A code fragment later in the paper shows exactly how the linkage occurs.

Once data-year is linked to format name, the most recent updates are searched first for a match. If no match occurs, then the processing moves back in time by invoking the embedded format. Eventually, in the absence of an update, the *E1990F* format is applied to the diagnostic code in the transaction file. *E1990F* is simply a copy of the *DX1F* format in Figure 1.

Overlapping range errors do not occur when embedded formats are used, so the same code on the left side of a format equation can appear in multiple linked formats (Shoemaker). Because of this feature, updates can be restricted to a defined range in time. For example, if code *12345* references Measles in 1998, Chicken Pox in 1997 and doesn't exist prior to 1997, a diagnosis of *12345* recorded in 1999 will be decoded correctly as Measles, one in 1997 will return Chicken Pox, and an entry in 1992 will be coded as missing. From Figure 1 for the original formats, code *12345* recorded in 1997 would be erroneously decoded as Measles.

Implementing Format Concatenation

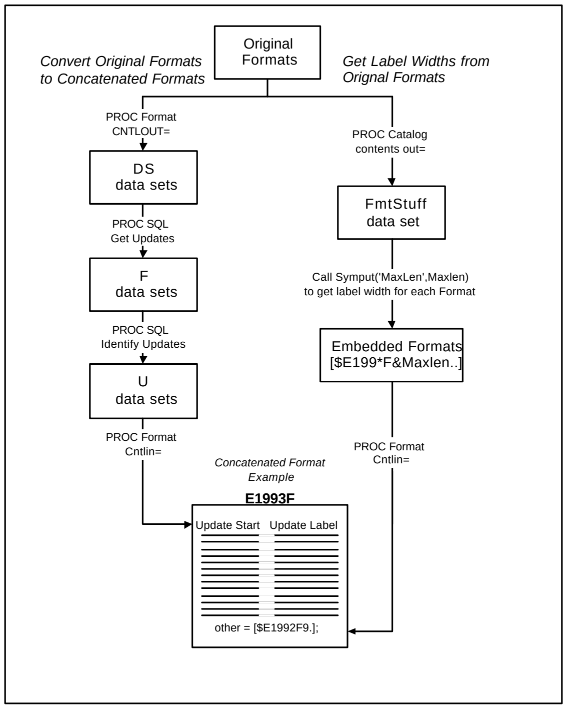
Original Formats
Convert Original Formats to Concatenated Formats
Get Label Widths from Orignal Formats
PROC Format CNTLOUT=
PROC Catalog contents out=
DS data sets
FmtStuff data set
PROC SQL Get Updates
F data sets
Call Symput('MaxLen',Maxlen) to get label width for each Format
PROC SQL Identify Updates
Embedded Formats [$E199*F&Maxlen..]
U data sets
PROC Format Cntlin=
PROC Format Cntlin=
Concatenated Format Example
E1993F
Update Start Update Label
other = [$E1992F9.];

Figure 3. Flowchart for creating concatenated formats.

Creating a series of concatenated formats involves the completion of two major tasks. The tasks depicted as two separate paths are flowcharted in Figure 3. They are described in separate sections below.

· Converting the Original Formats

From the flowchart one can see that all the original formats must be available to the developer for processing. These formats are converted to SAS data sets with the *cntlout* option on the FORMAT procedure. For example, DX1F populates variables *start* and *label* in the newly created SAS data set DS1 in the code example below:

```
proc format library=library.formats
cntlout=DS1 (keep=start label);
   select DX1F;
run;
```

Actually ten cntlout data sets, *DS1, DS2,…,DS10*, are needed for generating the new formats. A macro *%do-loop* creates the data sets efficiently. For simplicity, however, macros in the original programs will typically be resolved before appearing as code examples in the paper.

Next, annual updates need to be retrieved from the newly created data sets. The retrieval process is shown in Figure 4.

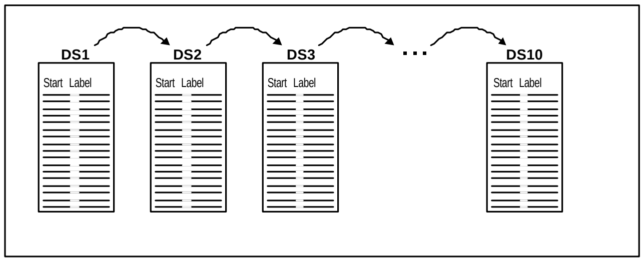

Figure 4. Yearly updates are obtained by processing the Cntlout data sets.

Note from Figure 4 that the second data set in the initial two-by-two comparison becomes the first one in the next comparison. PROC SQL is used in the actual processing. The SQL code below culls updates from the first two cntlout data sets and stores the results in F2:

```
create table F2 as
  select Early.Start as EStart,
         Early.Label as ELabel,
         Late.Start as LStart,
         Late.Label as LLabel
  from DS1 as Early full join DS2 as Late
  on Early.Start = Late.Start
  where Early.Label ^= Late.Label
;
```

The full outer join in the SQL command retrieves both deleted and new codes whereas the selection criteria listed in the *on* and *where* clauses returns updated labels. A left join alone would return the deleted codes whereas a right join would capture the new codes. Repeating the SQL command generates tables F3 to F10. The numeric extension on the name references the latter of the two tables being compared - mirroring how the update process actually works. For example, F10 for 1999 would capture changes that have occurred between 1998 and 1999.

At this point the four variables in the F* files need to be compressed into variables *start* and *label* used in the concatenated formats. The *union* set operator is used to differentiate processing for the three types of update:

```
create table U2 as /*U for update*/
  /*additions*/
    select LStart as Start, LLabel as Label
    from F2
    where LStart ^= ' ' and Estart = ' '

  union
    /*deletions*/
    select Estart as Start,
           "deleted" as Label
    from F2
      where ESTart ^= ' ' and Lstart = ' '

  union
    /*modifications*/
    select LStart as Start, LLabel as Label
    from F2
    where LStart=EStart and
    LLabel ^=ELabel;
```

Deletions must be labeled with a constant such as *deleted* so that the prior undeleted label for a code is not erroneously retrieved in a given transaction.

· Obtaining Label Widths from the Original Formats

Embedded formats should always be written with a specific numeric label width appended to the format name. Otherwise SAS issues a warning message and assigns a value of 40 to the label width. Labels with more than 40 characters will be truncated. Here is an example of an embedded format from Shoemaker's article:

```
        other=[$LOB12.];
```

Gerlach and Rinkus also append a width to their embedded date format, and they show that embedded formats are not restricted to *other* assignment statements:

```
  proc format;
    value titratf
    '01jan95'd - '31dec95'd = 'Screening'
    '01jan96'd - '30-jun97'd = [date7.]
    '01jul97'd - '31dec97'd = 'Post Study'
    other= 'Error';
  run;
```

Note from the examples above that square brackets rather than quotes surround embedded formats. They are not text strings.

In the application for longitudinal data, label widths are assigned dynamically to each of the 10 embedded formats with the CATALOG procedure:

```
  proc catalog c=library.FORMATS;
    contents out=fmtstuff;
  run;
  data AddLen(Keep=Name MaxLen);
    length MaxLen $3;
    set fmtstuff;
      MaxLen=scan(DESC,3,',');
    run;
```

A value for DESC in *fmtstuff* could be `FORMAT:MAXLEN=5,5,9` with the numbers referencing maximum lengths for *start*, *end* and *label* fields. The *scan* function parses the third word in the string and assigns a value of *9* to *MaxLen*, the length of the embedded format.

· Final Assembly with Cntlin

Now that the updates are stored in SAS data sets and the exact width of *label* is known, concatenated formats can be built with control-in data sets. Even though the value assigned to *label* below is a character string, the following works as intended:

```
  data Cntlin
  (keep=fmtname type hlo start label);
    retain fmtname "$E1992F" type "C";
    set U2 (keep=start label) end=last;
    output;
    if last then do;
      hlo="OF"; label="E1991F9."; output;
    end;
  run;
  proc format cntlin=Cntlin
              library=Library.EmbCat;
  run;
```

The reason this code works is that multiple-character values can be assigned to *HLO*. *O* means that the range is *other*, and *F* references *format* or *informat*. The argument for *Label* is a string, and the square brackets have been removed. *HLO* is also flexible enough to handle the embedded format in Gerlach's example above. Just set *HLO* to *F* and *label* to *date7.* when *start* equals '*01jan96'd*.

New features were added to *HLO* in release v6.07 meaning that additional letters can be assigned to this variable besides the original *H*, *L*, or *O* (*SAS Technical Report P-222: Changes and Enhancements to Base SAS® Software*, 210). In fact the variable would be better named as *HLONRFIS* to capture all of its features.

As the final step in the process, the concatenated formats just created are stored in a separate catalog so that they can be easily differentiated from their original counterparts. Also a final format, *CkTimeF*, is created with a macro so that the correct format for decoding can be selected at runtime. *CkTimeF* is stored in the same catalog as the concatenated formats:

```
  %macro CkTime(lowtime,hightime);
      proc format library=Library.EmbCat;
      value $CkTimeF
        %do time = &lowtime %to &hightime;
         "&time" = "$E&time.F"
        %end;
        other="Time ERROR"
      ;
```

```
    run;
%mend CkTime;
%CkTime(1990,1999);
```

How the *CkTimeF* format is used is described in the next section.

Using the Concatenated Formats

Now that the concatenated formats have been built, one needs to learn how to use them. A hard-coded simplified version of the *XX* family of formats below is adapted from the *LOB* formats in Shoemaker's article.

```
proc format library=Library.EmbCat;
  value $xx1990F     /*the base format*/
   'CO'='Commerical'
   'MC'='Medicare'
   'SF'='Self-Funded'
    other='Unknown'
  ;
  value $xx1991F
   'MD'='Medicaid'        /*add*/
   'CO'='Deleted'         /*delete*/
   'SF'='Single Family'   /*modify*/
    other=[$xx1990F11.]
  ;
  value $xx1992F
   'CP'='Compliant'            /*add*/
   'SF'='Deleted'              /*delete*/
   'MC'='Medicare or Medicaid' /*modify*/
    other=[$xx1991F13.]
  ;
run;
```

Unlike the DX formats described earlier, every format in *XX1990F* is updated in a subsequent year. Note that the *delete* type of edit is simply a special instance of the *modify* edit. Also a width equivalent to the longest label of the previous year's format has been appended to the embedded format.

The *XX* formats are invoked in the *decode* data set below. Highlighted lines of code will be discussed:

```
options fmtsearch = (library.EmbCat);
data decode;
   length label $20;
   input key $ keydate : mmddyy10.;
   yr=put(year(keydate),4.);
   thisfmt=put(yr,$CkTimeF.);
   if(index(thisfmt,'ERROR')) gt 0 then
     label=thisfmt;
   else
     label=putc(key,thisfmt);
 cards;
 MC 01/01/1991
 MC 01/01/1992
 SF 01/01/1990
 SF 01/01/1991
 SF 01/01/1992
 CP 01/01/1990
 CP 01/01/1992
 XX 01/01/1992
 XX 01/01/1999
 run;
```

Since the *XX* formats are stored in a catalog other than *work.formats* or *library.formats*, the *fmtsearch* option must be invoked for the *put* and *putc* functions to decode data properly. Next the *CKTimeF* format is invoked with a four digit character variable *yr* that has been derived from the data. *ChkTimeF* matches format to year or returns an error if the value for *yr* is out of range. Finally, the *putc* function formats *key* by referencing the variable, *thisfmt*. Because the *putc* function references a variable instead of a format, awkward *if-then-else* statements do not have to be inserted into the code. Instead, format selection is easily determined at runtime by the contents of *thisfmt*.

The reader is invited to check the output for accuracy:

```
               Data Set DECODE
KEY    KEYDATE       LABEL
MC     01JAN1991     Medicare
MC     01JAN1992     Medicare or Medicaid
SF     01JAN1990     Self-Funded
SF     01JAN1991     Single Family
SF     01JAN1992     Delete
CP     01JAN1990     Unknown
CP     01JAN1992     Compliant
XX     01JAN1992     Unknown
XX     01JAN1999     Time ERROR
```

Size Issues

A claim was made earlier in the paper that format concatenation would significantly reduce the amount of space required for storing formats. Table 1 below lists sizes of *cntlout* data sets for both the original and concatenated (updated) diagnostic formats:

Original	#Obs	Updated	#Obs	%Change
DS1	14,284	U1	14,284	
DS2	14,303	U2	53	0.371
DS3	14,410	U3	159	1.103
DS4	14,550	U4	202	1.388
DS5	14,703	U5	211	1.435
DS6	14,885	U6	267	1.794
DS7	14,960	U7	99	0.662
DS8	15,034	U8	96	0.639
DS9	15,119	U9	100	0.661
DS10	15,186	U10	87	0.573
Column Sums	147,434		15,558	8.6257

Table 1. A listing of the number of observations in the cntlout data sets for the original and concatenated (updated) formats.

The size of the concatenated (updated) data sets is one tenth that of their original counterparts. The relative size of the corresponding catalogs where the formats are stored is equivalent:

1,017KB v. 10,453KB. Note that every entry is represented only once in the concatenated formats. In fact, there is so much redundancy in the original formats that the total size of the concatenated formats is only slightly larger than DS10 (15,558 v. 15,186 observations).

An examination of Table 1 also shows that a relatively small change in percent for an annual update can add up to a sizeable 8.62 percent over a span of ten years. Nevertheless, it should be noted that a breakout by type of update was not performed in this exercise. Therefore, if the majority of the updates were additions, they could be satisfactorily tracked by employing conventional decoding methods. Correspondingly, if there are a lot of deletes and modifications in the updates, then format concatenation should be considered as an alternative.

Speed Comparisons between Conventional and Concatenated Formats

While disk space is reduced by a factor of ten for concatenated formats, a file of codes must be systematically processed to assess the relative efficiency of formatting data with concatenated and conventional methods.

The (**U**[pdate]) data sets depicted earlier in Figure 3 as the data source for the concatenated formats are also used as input for comparing speeds. All records from a given U[pdate] data set are assigned a date that is derived from the data set name. For example, ‘01Jan1999’d is assigned to *FDate* when all observations from n99.u10 are being processed. Values for both *DXCode* and *FDate* are written to the **V** datasets that are later concatenated to form *TestDat* containing 15,558 observations – the same number listed in Table 1 for the concatenated formats.

```
 libname n99 ‘c:\nesug\n99’;
%macro AffixYr;
 %do vsn= 1 %to 10;
   %let year=%eval(1989 + &vsn);
   %let chdate=01jan&year;
    data V&vsn (keep=start FDate
                rename=(start=DXCode));
      retain FDate "&chdate"d;
      set n99.U&vsn;
    run;
    proc append base=TestDat data=V&vsn;
 %end;
%mend AffixYr;
%AffixYr;
```

The time it takes to format *DXCode* in *TestDat* is captured during testing, and *FDate* after some manipulation points to specific format in a library. During testing records in *TestDat* are also permuted to eliminate bias from the order in which codes are formatted. The underlying algorithm for this process is displayed pictorially in Figure 5. The open triangular arrows show how the data are generated.

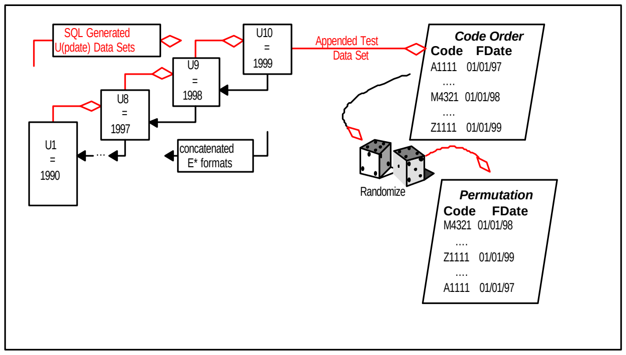

Figure 5. Generating the *TestDat* data set.

Because the test data and the concatenated formats originate from the same source, the chance that a given code is formatted by a more readily accessible updated entry reflects the original ratio of updated to unaltered codes in the format library. The underlying assumption here is that new codes are equally as likely to be formatted as old ones. In other words, all diagnoses and date combinations are equally represented in a transaction file.

There is, however, a major limitation to deriving the value for *FDate* from a record’s associated format name. If, for example, format *12345* is modified during 1998, then the date assigned to *FDate* in *TestDat* is 01Jan1998. This assignment unduly shortens processing time, because prior formats in a series never have to be searched for a matching code. To compensate for this bias, a second data set identical to *TestDat* is processed with the value for *FDate* fixed at 01Jan1999. In this second data set a backward search for matches is required for all format updates except those that occur in 1999. Figure 6 is a diagram of the four decoding methods that are used for comparing speed.

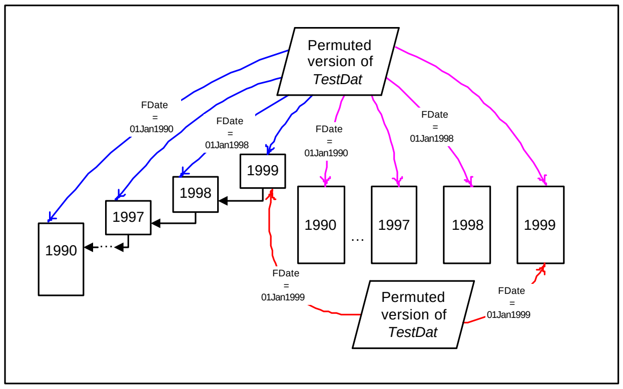

Figure 6. Formatting codes in *TestDat* with both Concatenated and original versions of the DX formats.

Concatenated formats are shown on the left side of Figure 6, and the conventional formats on the right side are diagrammed as larger rectangles. Table 2 below ranks speeds and shows how four methods for calculating run times are derived from Figure 6. Recall that the 1990 concatenated and conventional formats are duplicates of each other.

	Concatenated	Original
All Years	2	4 (slowest)
1999 (Current Year)	3	1 (fastest)

Table 2. 200 permutations of *TestDat* are programmed for each format method. While the current configuration is the fastest, concatenated methods outperform the conventional method where it is possible to access historically correct formats.

Besides permuting the records in *TestDat*, the order in which the four run types are submitted is also permuted. A data set, *pdata*, containing decoded values for approximately 15,000 records, is generated for each of the 800 executions of the data step. Since results from *pdata*, are not used in subsequent processing, the data set is simply overwritten each time the data step is executed. On the other hand, a record of the time it takes to create *pdata* needs to be preserved. The time is captured at the beginning of the data step, and the elapsed time in seconds is calculated at the end. Here is the code fragment where codes from *TestDat* are formatted:

```
/*two data sets are created:*/
data pdata(keep=dxcode yr thisfmt label)
/* T&i.&&order&j reflects the randomization
   of the runtype*/
   T&i.&&order&j (keep=RunType Elapsed);
   length label $31;
   retain time1;
/*SPROCESS is sorted in permuted order*/
   set sprocess end=last;
   if _n_ = 1 then do;
     current=time();
     hr1=hour(current);
     min1=minute(current)
     sec1=second(current);
     time1=3600*hr1+60*min1+sec1;
   end;
/*yrstring is set to "1999" or
           put(year(datadate),4.)*/
   yr=&yrstring;
/*Depending On &Ltr either original or em-
bedded fmt called*/
   thisfmt=put(yr,$CkTime&Ltr..);
   if(index(thisfmt,'ERR')) gt 0 then
     label=thisfmt;
   else
     label=putc(DXCode,thisfmt);
   output pdata;
   if last then do;
     current=time();
     hr2=hour(current);
     min2=minute(current);
     sec2=second(current);
     time2=3600*hr2+60*min2+sec2;
     elapsed = time2-time1;
     RunNum=&i;
     DataType = &&order&j;
     output t&i.&&order&j.;
   end;
run;
```

Highlighted lines for *thisfmt* and *label* demonstrate that the same process that was used to format the *XX* codes earlier is used with the *DX* codes here. Elapsed time is calculated by subtracting the initial time from the ending time. Single values for *Elapsed* and *RunType* are saved from each run. Results from the concatenated 800 one-line data sets are summarized graphically in Figure 7.

All possible pair-wise comparisons of Wilcoxon Scores in the NPAR1WAY procedure are significant (Prob> |Z| = 0.0001) for the data graphed in Figure 7. While the current configuration of *Orig:Fast* probably resembles the current version of a conventional format library, it achieves speed at the cost of accuracy, since there is no way to retrieve a historically valid label for an updated format.

What is surprising is how well the embedded formats perform. According to Figure 7, the worst case scenario requiring the maximum number of backward searches retrieves formatted labels faster than searching an individual complete format. The concatenation method, therefore, appears to confer significant advantages of speed, accuracy and disk space over its conventional counterpart.

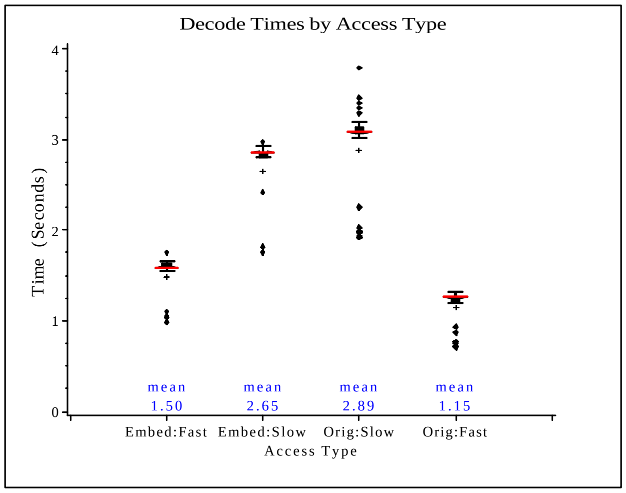

Figure 7. Box plots are used to display the distribution of speeds for 200 iterations of the four types of runs. The mean is denoted by a plus sign (+), and the median is marked with a thin wide horizontal line within a plot's *whiskers*. The plot is adapted from Ruzsa and Kalt's program *annoboxn.sas* Copyright © by SAS Institute.

Summary and Conclusions

Format concatenation has been described as a way to eliminate redundancy from multiple versioned formats that are applied selectively to longitudinal data. While the flowchart describing their construction appears to be complex, the actual code for creating a set of concatenated formats involves the application of just two SQL statements and one invocation of the CATALOG procedure. Also decoding data is pretty straightforward with the output of a format of formats being used as input to the *putc* function. As with conventional formats, concatenated formats can be constructed with control-in data sets.

As mentioned previously, longitudinal data are decoded accurately with concatenated formats. In addition, concatenated formats are economical in terms of disk storage space requirements and efficient in the time it takes to format a transaction file.

While there are many advantages to having concatenated formats available for decoding time-dependent data, the initial conversion process for multiple formats requires an investment of time and full access to versioned predecessors. Also if the formats interact with each other, additional problems may arise if they are updated at different time intervals.

References

Gerlach, John and Alan Rinkus. *Formats as a Programming Tool*. Proceedings of the 6th Annual SouthEast SAS®Users Group Conference. Norfolk, VA, September 13-15, 1998. 180-186.

Ruza, Peter and Mike Kalt. *annoboxn.sas* http:/ftp.sas.com/techsup/download/sample/graph/gplot-boxplot.txt. Copyright © 1998 by SAS Institute Inc., Cary, NC.

SAS Technical Report P-222: Changes and Enhancements to Base SASÒ Software: Release 6.07. Cary, NC: SAS Institute Inc., 1991. 207-217.

Shoemaker, Jack. *Advanced Techniques to Build and Manage User-Defined SASÒ FORMAT Catalogs*. Proceedings of the 11th Annual NorthEast SAS® Users Group Conference. Pittsburgh, PA. October 4-6, 1998. 102-107.

Author Information

Perry Watts
IMS Health
660 W. Germantown Pike
Plymouth Meeting, PA 19462
610-834-5084
pwatts@us.imshealth.com