Paper AD18

This is the Modern World: Simple, Overlooked SAS ® Enhancements

Bruce Gilsen, Federal Reserve Board, Washington, DC

INTRODUCTION

At my job as a SAS ® consultant at the Federal Reserve Board, reading the SAS-L internet newsgroup, and at SAS conferences, I've noticed that some smaller, less dramatic SAS enhancements seem to have fallen through the cracks. Users continue to use older, more cumbersome methods when simpler solutions are available. Some of these enhancements were introduced in Version 9.2, but others were introduced in Version 9, Version 8, or even Version 6! This paper reviews underutilized enhancements that allow you to more easily do the following.

1. Write date values in the form yyyymmdd.

2. Increment date values with the INTNX function.

3. Create transport files: PROC CPORT/CIMPORT versus PROC COPY with the XPORT engine.

4. Count the number of times a character or substring occurs in a character string or the number of words in a character string.

5. Concatenate character strings.

6. Check if any of a list of variables contains a value.

7. Sort by the numeric portion of character values.

8. Retrieve DB2 data on z/OS mainframes.

1. Write date values in the form yyyymmdd.

SAS provides a variety of informats and formats to read and write SAS dates. The *SAS 9.1.3 Language Reference: Dictionary* lists 22 date informats and 46 date formats. In my experience, date data being read into SAS are most often in the form yyyymmdd, and are overwhelmingly written in the form yyyymmdd. Writing dates in the yyyymmdd format has not always been easy.

On input, the YYMMDD8. informat reads dates in the form yyyymmdd. On output, however, the YYMMDD8. format writes dates in the form yy-mm-dd and the YYMMDD10. format writes dates in the form yyyy-mm-dd, as in the following code.

```
data one;
  date1 = "18oct2008"d;     /* SAS date value for 10/18/2008 */
  put date1 yymmdd8.;       /* writes 08-10-18 */
  put date1 yymmdd10.;      /* writes 2008-10-18 */
```

Prior to Version 8, the easiest way to write data in the form yyyymmdd was to use the PUT function to create a character string of the form yyyy-mm-dd and use the COMPRESS function to remove the dashes, as in the following statements.

```
datetemp = compress(put(date1,yymmdd10.),'-');
put datetemp $8. ;        /* writes 20081018 */
```

In Version 8, SAS added a new format, YYMMDDxw, where x is an optional separator and w is the number of bytes. The optional separator x can be one of the following.

B blank

C colon

D dash

N none

P period

S slash

The N separator allows you to print dates in the form yyyymmdd much more easily than the method shown above.

```
date1 = "18oct2008"d;     /* SAS date value for 10/18/2008 */
put date1 yymmddn8.;      /* writes 20081018 */
```

Here are some examples of the other separators.

```
data one ;
  date1 = "18oct2008"d;     /* SAS date value for 10/18/2008 */
  put date1 yymmddb8.;      /* writes 08 10 18 */
  put date1 yymmddc8.;      /* writes 08:10:18 */
  put date1 yymmddd10.;     /* writes 2008-10-18 */
  put date1 yymmddn8.;      /* writes 20081018 */
  put date1 yymmddp8.;      /* writes 08.10.18 */
  put date1 yymmdds10.;     /* writes 2008/10/18 */
  put date1 yymmddn6.;      /* writes 081018 */
```

2. Increment date values with the INTNX function.

The data step function INTNX returns a SAS date value incremented by a specified number of intervals (days, weeks, months, quarters, years, etc.).

INTNX has three required arguments and one optional argument, commonly used as follows for SAS date values.

INTNX(*interval*, *start-from*, *increment* <,*alignment*>);

- *interval* is the unit of measure (days, weeks, months, quarters, years, etc.) by which *start-from* is incremented.
- *start-from* is a SAS date value to be incremented.
- *increment* is the integer number of intervals by which *start-from* is incremented (negative values = earlier dates).
- *alignment* is where *start-from* is aligned within *interval* after being incremented. Possible values are BEGINNING, MIDDLE, END, and (new in Version 9) SAMEDAY. This argument is optional, and defaults to BEGINNING.

One issue with INTNX is that the default *alignment* is BEGINNING, so by default *start-from* is aligned to the beginning of the period after being incremented. This leads to unexpected results for intervals other than DAY, as in the following code.

```
date1 = '18oct2008'd;
dateplus2month = intnx('month',date1,2);
```

SAS first increments by two months, then aligns to December 1, 2008 (the beginning of the interval, MONTH). The result is 17,867, the SAS date value for December 1, 2008. Similarly, intnx('year',date1,2) aligns to the beginning of the interval, YEAR, and returns the SAS date value for January 1, 2010, not October 18, 2010.

Users might infrequently want to align to the start or end of an interval, but overwhelmingly want to keep the same alignment when they increment a date. That is, they want intnx('month', '18oct2008'd,2) to return the SAS date value for December 18, 2008, not December 1, 2008 (BEGINNING), December 16, 2008 (MIDDLE), or December 31, 2008 (END).

A second INTNX issue prior to Version 9 was the extra coding needed to account for leap years. The following code accounts for leap years when shifting a date by years; more generalized code is needed to handle all frequencies.

```
offset = 2;      /* number of years to increment */
if day(date1) = 29 and month(date1) = 2 and mod(offset,4) ne 0
  and mod(offset,400) ne 0
  then date1 = mdy(2, 28, year(date1)+offset) ;
  else date1 = mdy(month(date1), day(date1), year(date1)+offset) ;
```

Happily, SAS added a new *alignment* value, SAMEDAY, in Version 9. SAMEDAY preserves the SAS date value's alignment within the interval after it is incremented, generating the expected results. It also handles leap years correctly. To prevent the problems shown above, always set *alignment* to SAMEDAY for intervals other than DAY (when interval is DAY, SAMEDAY is not necessary). Here are some examples; note that 2000 and 2004 but not 2003 are leap years. As before, DATE1 is the SAS date value for October 18, 2008.

SAS Statement	Description	Value	SAS date
date2=intnx('day',date1,2,"sameday");	2 days after 10/18/2008	17825	10/20/2008
date3=intnx('month',date1,2,"sameday");	2 months after 10/18/2008	17884	12/18/2008
date4=intnx('year',date1,2,"sameday");	2 years after 10/18/2008	18553	10/18/2010
date5=intnx('year','29feb2000'd,1,"sameday");	1 year after 2/29/2000	15034	2/28/2001
date6=intnx('year','29feb2000'd,4,"sameday");	4 years after 2/29/2000	16130	2/29/2004
date7=intnx('month','31mar2003'd,-1,"sameday");	1 month before 3/31/2003	15764	2/28/2003
date8=intnx('month','31mar2004'd,-1,"sameday");	1 month before 3/31/2004	16130	2/29/2004

Note that until SAS Version 9.2, SAMEDAY should only be used with single, non-shifted date intervals (DAY, WEEK, WEEKDAY, TENDAY, SEMIMONTH, MONTH, QTR, SEMIYEAR, YEAR), because as described in SAS Note 016184, the following intervals might return the wrong answer with no error or warning.

- multiple (e.g., month2 = two-month interval)
- shifted (e.g., month.4 = month interval starting on April 1)
- time
- datetime

3. Create transport files: PROC CPORT/CIMPORT versus PROC COPY with the XPORT engine.

A transport file is a file organized in a machine-independent format that the SAS system can read on any operating system. A transport file can be used to copy SAS data sets from one operating system to another. Starting with Version 6, there are two types of transport files. An example of how to use each type of transport file follows.

Type 1: XPORT-style transport files.

Step 1. On the original host, use PROC COPY to create a transport file containing the data set(s) to be copied. Specify the EXPORT engine on the output library's LIBNAME statement.

```
libname  inn  'directory-with-SAS-data-set(s)' ;
libname  trans  xport  'transport-file-to-create';
proc copy  in=inn  out=trans ;
run ;
```

Step 2. Copy the transport file to the destination host with the ftp command or another method.

Step 3. On the destination host, use PROC COPY to convert the transport file into native SAS data set(s). Specify the XPORT engine on the input library's LIBNAME statement.

```
libname  trans  xport  'transport-file' ;
libname  sasdata1  'SAS-library-write-data-set(s)-here' ;
proc copy  in=trans  out=sasdata1 ;
run ;
```

Type 2: CPORT-style transport files.

Step 1. On the original host, use PROC CPORT to create a transport file containing the data set(s) to be copied.

```
libname  sasdata1  'directory-with-SAS-data-set(s)' ;
filename  trans  'transport-file-to-create';
proc cport  library=sasdata1  file=trans ;
run ;
```

Step 2. Copy the transport file to the destination host with the ftp command or another method.

Step 3. On the destination host, use PROC CIMPORT to convert the transport file into native SAS data set(s).

```
filename  trans  'transport-file' ;
libname  sasdata1  'SAS-library-write-data-set(s)-here' ;
proc cimport  infile=trans  library=sasdata1 ;
run ;
```

Starting with Version 7, changes to SAS data sets included the following.

- The maximum length of a variable name increased from 8 to 32 characters.
- The maximum length of a character variable increased from 200 to 32768.

- SAS stores and displays variable names by case, based on "first reference" - the first time it encounters the variable. SAS does not distinguish by case (i.e., if you create variable Abcd, SAS displays it that way, but considers it the same variable as abcd).

Some users still create XPORT-style transport files, but since Version 7, PROC CPORT and CIMPORT should be used in all cases. XPORT-style transport files are created in the Version 6 format. Version 6 does not understand the features introduced in Version 7 (long variable names, long character variable values, and mixed case variable names), so XPORT-style transport will generate undesirable results in some cases.

If you set OPTIONS VALIDVARNAME=V6; before creating the transport file, undesirable results might occur as follows.

- SAS data sets with any character variables longer than 200 characters are "copied" to the transport file as empty data sets and error messages are written to the SAS log.
- All other SAS data sets are correctly copied to the transport file, with the following changes.
    - Mixed case variable names are converted to upper case.
    - Variable names longer than 8 characters are truncated to 8 character names, using an algorithm that ensures that all variable names are unique. If a truncated variable does not have a label, the old (long) variable name becomes the label of the new (8 character) variable name.

If you do not set OPTIONS VALIDVARNAME=V6; before creating the transport file, undesirable results might occur as follows.

- SAS data sets with any variable names longer than 8 characters or character variables longer than 200 characters are "copied" to the transport file as empty data sets, and error messages are written to the SAS log.
- All other SAS data sets are correctly copied to the transport file, with the following change: mixed case variable names are converted to upper case.

4. Count the number of times a character or substring occurs in a character string or the number of words in a character string.

We might want to do one of the following for the character string "ab1 ab a344444abb555 ab 4".

- Count the number of occurrences of the substring "ab".
- Count the number of occurrences of the characters "a" or “b".
- Count the number of space-separated words.

Prior to Version 9, users typically used iterative methods (loops) to count occurrences of these types. To simplify the code, use COUNT (new in Version 9), COUNTC (new in Version 9), or COUNTW (new in Version 9.2) instead.

Here is typical pre-Version 9 code to count the number of times that the substring "ab" occurs in "ab1 ab a344444abb555 ab 4". It uses the INDEX function to search for a substring of characters. The result, NUM_AB, is 4.

```
data one;
  char1 = "ab1 ab a344444abb555  ab 4";   /* character variable we want to check */
```

```
  num_ab = 0;                          /* counter for number of occurrences */
  loc=index(char1,"ab");               /* find first occurrence */
  do while (loc ne 0);
    num_ab+1;                          /* count occurrences of "ab" */
    char1 = substr(char1,loc+2);       /* resume search after "ab" */
    loc = index(char1, "ab");          /* find next occurrence */
  end;
run;
```

To simplify the code in Version 9, use the COUNT function, which counts the number of times that a substring occurs in a character string. The result, NUM_AB, is 4, as in the previous code.

```
data one;
  char1 = "ab1 ab a344444abb555  ab 4";   /* character variable we want to check */
  num_ab = count(char1,"ab");
run;
```

Here is typical pre-Version 9 code to count the number of times the characters "a" or “b" occur in "ab1 ab a344444abb555 ab 4". It uses the INDEXC function to search for any of one or more characters. The result, NUM_A_OR_B, is 10.

```
data one;
  char1 = "ab1 ab a344444abb555  ab 4";   /* character variable we want to check */

  num_a_or_b = 0;                      /* counter for number of occurrences */
  loc=indexc(char1,"ab");              /* find first occurrence */
  do while (loc ne 0);
    num_a_or_b+1;                      /* count occurrences of "a" or "b" */
    char1 = substr(char1,loc+1);       /* resume search after "a" or "b" */
    loc = indexc(char1, "ab");         /* find next occurrence */
  end;
run;
```

To simplify the code in Version 9, use the COUNTC function, which counts the number of times one or more characters occur in a character string. The result, NUM_A_OR_B, is 10, as in the previous code.

```
data one;
  char1 = "ab1 ab a344444abb555  ab 4";   /* character variable we want to check */
  num_a_or_b = countc(char1,"ab");
run;
```

Here is typical pre-Version 9.2 code to count the number of space-separated words in "ab1 ab a344444abb555 ab 4". It uses the INDEXC function to search for spaces. The LEFT function accounts for multiple spaces. The result, NUM_WORDS, is 5.

```
data one;
  char1 = "ab1 ab a344444abb555  ab 4";   /* character variable we want to check */

  num_words = 0;                          /* counter for number of words */
  loc=indexc(char1," ");                  /* find first occurrence */
  do while (loc ne 0 and char1 ne "");
    num_words+1;                          /* count number of words */
    char1 = left(substr(char1,loc+1));    /* resume search after space */
    loc = indexc(char1, " ");             /* find next occurrence */
  end;
run;
```

To simplify the code starting in Version 9.2, use the COUNTW function, which counts the number of words in a character string. The result, NUM_WORDS, is 5, as in the previous code.

```
data one;
  char1 = "ab1 ab a344444abb555  ab 4";   /* character variable we want to check */
  num_words = countw(char1," ");
run;
```

COUNT, COUNTC, and COUNTW have arguments that provide features such as allowing you to trim trailing blanks and make the search case-insensitive. See the documentation for details.

5. Concatenate character strings.

Traditionally, the concatenation operator, ||, was used to concatenate character strings. If the strings contained leading or trailing blanks, they were included in the result unless the user removed them with additional functions, such as TRIM or TRIM(LEFT)), as in the examples below.

Here is a typical DATA step with some concatenation operations.

```
data one;
  length char1 char2 char3 $8.;
  char1="xxxxxxxx";
  char2=" yyy";     /* 1 leading blank */
  char3="zzzz";

    /* Concatenate: leading and trailing blanks are included */
  cat1 = char1 || char2 || char3;

    /* Concatenate: trailing blanks are removed */
  cat2 = trim(char1) || trim(char2) || trim(char2);

    /* Concatenate: leading and trailing blanks are removed */
```

```
  cat3 = trim(left(char1)) || trim(left(char2)) || trim(left(char3));

  put cat1= cat2= cat3=;
run;
```

Since CHAR1, CHAR2, and CHAR3 were defined with a length of 8, the values of CAT1, CAT2, and CAT3 are as follows.

- CAT1 includes the leading blank and the 4 trailing blanks from CHAR2, and the 4 trailing blanks from CHAR3.
- CAT2 includes the leading blank but not the trailing blanks from CHAR2.
- CAT1, CAT2, and CAT3 all have a length of 24, which is the sum of the lengths of the three contributing variables. They have enough trailing blanks to fill the 24 character length.

```
cat1=xxxxxxxx yyy    zzzz
cat2=xxxxxxxx yyyzzzz
cat3=xxxxxxxxyyyzzzz
```

In Version 9, the program can be re-coded more simply using the following new functions.

- CAT concatenates character strings but does not remove leading or trailing blanks.
- CATT concatenates character strings and removes trailing blanks.
- CATS concatenates character strings and removes leading and trailing blanks.
- CATX concatenates character strings, removes leading and trailing blanks, and inserts a delimiter (separator) between each argument.

CAT, CATT, and CATS can be used to re-code the three concatenation statements shown above. Here, the concatenation statements from above are compared to comparable statements using the new functions.

```
  /* Concatenate: leading and trailing blanks are included */
cat1 = char1 || char2 || char3;
cat1_cat = cat(char1, char2, char3);

  /* Concatenate: trailing blanks are removed */
cat2 = trim(char1) || trim(char2) || trim(char2);
cat2_catt = catt(char1, char2, char3);

  /* Concatenate: leading and trailing blanks are removed */
cat3 = trim(left(char1)) || trim(left(char2)) || trim(left(char3));
cat3_cats = cats(char1, char2, char3);
```

CAT1 is the same as CAT1_CAT, CAT2 is the same as CAT2_CATT, and CAT3 is the same as CAT3_CATS, with an important exception.

- If CAT1, CAT2, or CAT3 are not previously defined, their length is the sum of the length of the contributing variables: 8+8+8=24 in this case.

- If CAT1_CAT, CAT2_CATT, or CAT3_CATS are not previously defined, their length is 200.

CATX is similar to CATS, except that the first argument specifies a delimiter (separator) to insert between each subsequent argument. If the first argument is a variable, the delimiter could differ from observation to observation. Here are a few examples.

```
data one;
  length char1 char2 char3 $8.;
  char1="xxxxxxxx";
  char2=" yyy";     /* 1 leading blank */
  char3="zzzz";
  aa = '123';

  char_catx1 = catx(aa, char1, char2);              /* value of AA as delimiter */

  char_catx2 = catx(',', char1, char2, char3);      /* comma as delimiter */

  char_catx3 = catx('123', char1, char2, char3);    /* '123' as delimiter */

  char_catx4 = catx(' ', char1, char2, char3);      /* single space as delimiter */
run;
```

CHAR_CATX1, CHAR_CATX2, CHAR_CATX3, and CHAR_CATX4 have lengths of 200, and values as follows.

```
char_catx1=xxxxxxxx123yyy
char_catx2=xxxxxxxx,yyy,zzzz
char_catx3=xxxxxxxx123yyy123zzzz
char_catx4=xxxxxxxx yyy zzzz
```

6. Check if any of a list of variables contains a value.

Suppose we want to check if any of a list of variables, X1 - X5, have the value 1, and set the variable ISONE to 1 in observations where this is true and 0 otherwise. There are many ways to code this problem.

One method is to test each variable one at a time. This method quickly becomes unwieldy for large groups of variables. Also, extraneous variables are tested. For example, if X2 is 1, then it is unnecessary and inefficient to test the remaining variables for that observation.

```
data new;
  set old;
  if x1 = 1 or x2 = 1 or x3 = 1 or x4 = 1 or x5 = 1
```

```
    then isone=1;
    else isone=0;
```

A second method is to put the variables in an array and loop through the array. As with the first method, all variables in all observations are tested, including the remaining (extraneous) variables after the value is found in an observation.

```
data new;
  set old;
  array xall (*) x1-x5;
  drop i;
  is_one=0;
  do i = 1 to dim(xall);
    if xall(i) = 1 then is_one=1;
  end;
```

A third method is to put the variables in an array and loop through the array, stopping if the value is found to prevent extraneous tests. If not all variables are equally likely to have the value, order the variables left to right in the array from most likely to have the value to least likely to have the value to reduce the number of tests.

```
data new;
  set old;
  array xall (*) x1-x5;
  drop i;
  is_one=0;
  do i = 1 to dim(xall) while (is_one=0);
    if xall(i) = 1 then is_one=1;
  end;
```

Starting in Version 9, a simpler method is to use the IN operator with the array name, as follows. Variables are checked left to right in the array and testing stops if the value is found. As in the third method, order the variables left to right in the array from most likely to have the value to least likely to have the value (if known) to reduce the number of tests.

```
data new;
  set old;
  array xall (*) x1-x5;
  if 1 in xall
    then is_one=1;
    else is_one=0;
```

7. Sort by the numeric portion of character values.

Data set ONE contains the following values of the variable BANK.

```
bank2
bank100
bank1
bank10
```

A conventional PROC SORT does a left to right character-by-character comparison, and "1" in "bank100" is less than "2" in "bank2". We want to order the integer values within the text by their numeric value, so that 2 is less than 10 and 100.

Conventional sort order	Desired sort order
bank1	bank1
bank10	bank2
bank100	bank10
bank2	bank100

Prior to Version 9.2, one solution was to create a new variable, BANKTEMP, with the numeric part of BANK, and then sort by the new variable, as in the following code. This code assumes that the numeric part of the value always starts at the same location (the fifth character) and is at most three digits; a more general case would be slightly more complicated.

```
data two;
  set one;
  banktemp = input(substr(bank,5),3.);
run;

proc sort data=two out=three;
  by banktemp;
run;
```

Starting in Version 9.2, SORTSEQ=LINGUISTIC with the NUMERIC_COLLATION option sorts integer values by their numeric value instead of character by character. Since 2 is less than 10 and 100, the desired sort order is returned.

```
proc sort data=one out=two
  sortseq=linguistic (numeric_collation=on);
  by bank;
run;
```

Numeric collation does not apply only to the end of character values. Suppose variable ADDRESS includes the following values.

```
East 14th Street
East 2nd Street
```

A conventional sort orders East 14th Street before East 2nd Street because “1" is less than “2", but SORTSEQ=LINGUISTIC (NUMERIC_COLLATION=ON) orders East 2nd Street before East 14th Street because 2 is less than 14.

8. Retrieve DB2 data on z/OS mainframes.

There are several ways to retrieve DB2 data on z/OS. I still see users using PROC DB2EXT or PROC ACCESS when they could use a SAS/ACCESS LIBNAME statement or the Pass-Through Facility. Each of these methods is reviewed below. Starting in Version 7, the maximum length of a variable name increased from 8 to 32 characters, and it is important to understand how each method handles long variable names.

1. PROC DB2EXT.

PROC DB2EXT is the oldest way to retrieve DB2 data. Since Version 6 (in 1990), PROC DB2EXT has been an undocumented feature that continues to work. In Version 9, PROC DB2EXT displays the following message:

```
NOTE: Beginning with SAS version 10, DB2EXT will no longer be available.
```

Because PROC DB2EXT is a Version 6-compatible procedure, it cannot process long DB2 column names. It truncates them to 8 characters using an algorithm that ensures unique variable names.

2. PROC ACCESS.

PROC ACCESS allows you to retrieve DB2 data using access and view descriptors. It was introduced in Version 6 and is still supported but not recommended; SAS Institute recommends using a SAS/ACCESS LIBNAME statement or the Pass-Through Facility. SAS/ACCESS access and view descriptors can be converted to SQL views with PROC CV2VIEW, which is documented in the *SAS/ACCESS 9.1.3 Interface to Relational Databases*.

Because PROC ACCESS is a Version 6-compatible procedure, it cannot process long DB2 column names. It truncates them to 8 characters using an algorithm that ensures unique variable names.

3. SAS/ACCESS LIBNAME statement.

The SAS/ACCESS LIBNAME statement allows you to assign a libref directly to a DB2 database. It was introduced in Version 7 and is the easiest way to retrieve DB2 data. Long DB2 column names are processed directly.

4. Pass-Through Facility.

The Pass-Through Facility allows you to retrieve DB2 data by sending DBMS (Database Management System) - specific SQL statements to DB2. It was introduced in Version 6. It processes long DB2 column names directly unless you set the system option VALIDVARNAME to V6, in which case it truncates to 8 characters as per PROC DB2EXT and PROC ACCESS. The value of V6 for VALIDVARNAME is documented in Version 8, and undocumented but supported in Version 9.

In some cases, the Pass-Through Facility can be more efficient than the SAS/ACCESS LIBNAME statement, particularly for complex queries. The SAS/ACCESS LIBNAME statement and the Pass-Through Facility are compared in the *SAS/ACCESS 9.1.3 Interface to Relational Databases* (*Overview of the SAS/ACCESS to Relational Databases* chapter, *Selecting a SAS/ACCESS method* section).

Example.

The following program retrieves the same DB2 data on the Federal Reserve Board's z/OS mainframe. The only difference in the results is how long column names, if any, are handled, as described above.

```
  /* 1. PROC DB2EXT */

options db2ssid=dsn;
run;
proc db2ext out=new1;
 select cntry_nm
 from nicua.cuv_country_nm;
run;

  /* 2. PROC ACCESS */

proc access dbms=db2;
  create work.new1.access;
  table = nicua.cuv_country_nm;
  ssid = dsn;
  create work.new1.view;
  select cntry_nm;
run;

  /* 3. SAS/ACCESS to DB2 LIBNAME statement */

libname in1 db2 ssid=dsn authid=nicua;

data new1;
 set in1.cuv_country_nm
   (keep = cntry_nm);
run;

  /* 4. Pass-Through Facility */

proc sql;
  connect to db2 (ssid=dsn);
  create table new1
  as select * from connection to db2
    (select cntry_nm
    from nicua.cuv_country_nm);
disconnect from db2;
quit;
```

Note that if you convert from PROC DB2EXT or PROC ACCESS to a SAS/ACCESS LIBNAME statement or the Pass-Through Facility, and your DB2 data has column names longer than 8 characters, the resulting variable names will not match the (shorter) variable names generated by PROC DB2EXT or PROC ACCESS. If subsequent SAS code uses the shorter variable names, you can use a RENAME statement in a subsequent step or modify your SAS code to use the longer names.

CONCLUSION

This paper reviewed some smaller, less dramatic SAS enhancements that seem to have fallen through the cracks. Hopefully, users will embrace these enhancements and simplify their SAS applications.

REFERENCES

Gilsen, Bruce (2003), "Deja-vu All Over Again: Common Mistakes by New SAS Users," Proceedings of the Sixteenth Annual NorthEast SAS Users Group Conference. http://www.nesug.org/html/Proceedings/nesug03/bt/bt010.pdf

Gilsen, Bruce (2006), "Improve Your Dating: The INTNX Function Alignment Value SAMEDAY," Proceedings of the Thirty-first Annual SAS Users Group International Conference. http://www2.sas.com/proceedings/sugi31/027-31.pdf

Gilsen, Bruce (2004), "More Tales from the Help Desk: Solutions for Simple SAS Mistakes," Proceedings of the Seventeenth Annual NorthEast SAS Users Group Conference.
http://www.nesug.org/html/Proceedings/nesug04/pm/pm11.pdf

Olson, Diane (2008), "New in SAS® 9.2: It's the Little Things That Count," Proceedings of the SAS Global Forum 2008 Conference. http://www2.sas.com/proceedings/forum2008/176-2008.pdf

SAS Institute Inc. (2004), "Base SAS 9.1.3 Procedures Guide," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "SAS/ACCESS 9.1.3 Interface to Relational Databases," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "SAS 9.1.3 Language Reference: Concepts," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "SAS 9.1.3 Language Reference: Dictionary, Volumes 1, 2, and 3," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2006), SAS Note 016184, "INTNX function with SAMEDAY alignment does not support multiple, shifted, time, or datetime intervals." http://support.sas.com/kb/16/184.html

SAS Institute Inc. (2010), SAS Sample 41182: Use the IN operator with arrays to check for existence of a value." http://support.sas.com/kb/41/182.html

Secosky, Jason (2008), "A Sampler of What's New in Base SAS® 9.2," Proceedings of the Twenty-first Annual NorthEast SAS Users Group Conference. <http://www.nesug.org/Proceedings/nesug08/ff/ff15.pdf>

ACKNOWLEDGMENTS

The following people contributed extensively to the development of this paper: Heidi Markovitz and Donna Hill at the Federal Reserve Board, Howard Schreier at Howles Informatics, and Diane Olson at SAS Institute. Their support is greatly appreciated.

CONTACT INFORMATION

For more information, contact the author, Bruce Gilsen, by mail at Federal Reserve Board, Mail Stop N-122, Washington, DC 20551; by e-mail at bruce.gilsen@frb.gov; or by phone at 202-452-2494.

TRADEMARK INFORMATION