A Sampler of What's New in Base SAS® 9.2

Jason Secosky, SAS Institute Inc., Cary, NC

ABSTRACT

Coding with SAS is easier than ever with SAS 9.2. This paper highlights the top new features and performance improvements in DATA step, PROC SQL, and PROC SORT. Included are writing functions with DATA step syntax, improved performance when accessing an external database from PROC SQL, more intuitive and culturally acceptable sorting with PROC SORT, and several "Top 10" SASware Ballot items.

INTRODUCTION

SAS is a wonderful language to code in. Common data manipulations and analyses take few statements without having to specify as many details as with general purpose programming languages. However, there are times when coding in SAS requires more code than one would think, distracting from the intent of the code. Each new version of SAS, like SAS 9.2, adds new features to ease programming in SAS. This paper describes user requested features that have been added to Base SAS 9.2 for the DATA step, the SQL procedure, and the SORT procedure.

We start by describing how to write functions using DATA step syntax. This makes writing modular programs easier without macros. The second section describes a features that automatically transforms PUT functions in a WHERE clause into external database SQL syntax. This enables more of a query to be passed to a database for faster subsetting operations. The next section presents linguistic collation. It is a culturally intuitive ordering with PROC SORT that can provide a dictionary-like ordering for words. The last section looks at new DATA step features like data set lists and string delimiters when reading and writing text files.

USER-WRITTEN DATA STEP FUNCTIONS

Solving difficult programming problems are made easier by combining smaller program units. SAS has many built-in functions, like SUBSTR and SUM, that allow DATA step programs to be built from smaller components. Before SAS 9.2, DATA step programmers would create macros to group multiple DATA step statements to ease programming.

One must be cautious when using macros in a DATA step. For instance, %DO loops in a DATA step can generate huge amounts of DATA step code that impact performance, forgetting an ampersand on a variable name may inadvertently modify a DATA step variable, and undisciplined use of macros can result in difficult to understand programs.

In SAS 9.2, SAS programmers can write their own functions, using DATA step syntax and the FCMP procedure. This eases programming, provides DATA step logic in SQL queries and WHERE clauses, and includes safety mechanisms like variable scope. Also, functions are stored in data sets, making it easy to share them between many SAS programmers in an organization regarless of the operating system they use.

PROC FCMP is used to create functions and CALL routines. FCMP stands for "function compiler." Functions and CALL routines created with PROC FCMP are called FCMP routines. In SAS 9.2, FCMP routines can be called from the DATA step, PROC SQL queries, and WHERE clauses like any other SAS function. This enables programmers to more easily read, write, and maintain complex code with independent and reusable subroutines.

In the following sections, we summarize the syntax for creating a function, while providing guidelines for when one would want to write a function versus a macro. Some of the text and concepts of this section are taken from Secosky (2007).

COMPARING MACRO AND FUNCTIONS

To compare and contrast Macro and PROC FCMP, we code a simple routine to compute a "study day." In clinical trials, computing a study day is a common task. A study day is the number of days a subject has participated in a trial. A study day computation has two inputs, the date a subject starts a trial and the date an event occurs.

If the event date is the same as the start date, this is study day 1. Days after the start date increase from 1. That is, the day after the start date is study day 2, two days after the start date is study day 3, and so on. Days before the start date decrease from -1. That is, the day before the start date is study day -1, two days before the start date is study day -2, and so on.

To compute a study day, we could merely subtract the event and start dates. Unfortunately, the computation isn't that simple. When the event date is the same as the start date, subtracting the two results in a study day of zero, which is not valid. We need IF..THEN..ELSE logic along with the subtract. Next, we examine a macro to compute study day.

A macro to compute study day within a DATA step takes three parameters. The start date, the event date, and a place to put the result. Here is a macro for study day that can be used within a DATA step.

```
%macro study_day(start, event, study_day);
  n = &event - &start;
  if n >= 0 then
    &study_day = n + 1;
  else
    &study_day = n;
%mend;
/* There is a bug in this macro that we describe next. */
```

First we subtract the event and start dates. If the value is greater than or equal to zero, we add one to the difference to avoid the zero day. If the value is less than zero, we do not need to make any change to the difference.

The macro is simple and gets the job done. Also, it encapsulates and names the study day logic. To share the macro, we can put it on a shared drive and a co-worker can place this path on their AUTOCALL path to find the macro.

Even with this simple macro, *there is a subtle bug in it*. When the macro is used in an existing DATA step, the program may not compute the same values as it did before using the %STUDY_DAY macro. The problem is with not putting an ampersand before the variable N. Let's look at the problematic use of this macro.

We use the %STUDY_DAY macro in a program that also computes the number of visits to a clinic. The number of visits is stored in the variable N. Then, the program determines the study day for an event. If there were more than five clinic visits, a particular action is taken. Here is a portion of the program:

```
data results;
  set trial_data;
  n = sum(of visits_jan--visits_dec);
  %study_day(start, event, study_day)
  if n > 5 then
    ...
```

The %STUDY_DAY macro inadvertently changes the value of the variable N. This changes the number of visits the subject made to the clinic. A macro generates SAS source code. Let us look at the source code the macro generates, in bold, to better see how the macro changes the variable N.

```
data results;
  set trial_data;
  n = sum(of visits_jan--visits_dec);
  n = event - start;
  if n >= 0 then
    study_day = n + 1;
  else
    study_day = n;
  if n > 5 then
    ...
```

N is set to the sum of the of the number of clinic visits. Then, the macro generates DATA step code that assigns the difference between EVENT and START to N. When using Macro, one has to be disciplined to avoid forgetting to put an ampersand before a macro variable name. If the macro were more complicated or if the macro used other macros, hunting down how a macro changes DATA step variables is tedious and time consuming.

In addition, if we wanted to compute a study day from PROC SQL or a WHERE clause, we would need to code a different version since, in general, DATA step syntax cannot be used in PROC SQL or a WHERE clause.

Instead of writing a macro, writing a function with PROC FCMP alleviates the problem of inadvertently changing DATA step variables. And, the same function can be called from DATA step, PROC SQL, and WHERE clauses. PROC FCMP routines have the following advantages:

- Use DATA step syntax, making the code more clear.
- Have local variables to store intermediate results.
- Can be called from DATA step, PROC SQL, WHERE clauses, and other locations like built-in SAS functions.
- Are faster by operating on numeric values as numbers, instead of converting a character value to a number, then converting the result from a number back into a character value as Macro does.

FCMP STUDY_DAY FUNCTION

A STUDY_DAY function is coded with PROC FCMP with this code:

```
proc fcmp outlib=sasuser.funcs.trial;
  function study_day(start, event);
    n = event - start;
    if n >= 0 then
      return(n + 1);
    else
      return(n);
  endsub;
```

This code creates a function named STUDY_DAY in a package named Trial. The package is stored in the data set Sasuser.Funcs. A *package* is a collection of routines that have unique names. STUDY_DAY takes two numeric parameters, START and EVENT. The body of the routine uses DATA step syntax to compute a study day, as described earlier in this paper.

The variable N in the STUDY_DAY function is a local variable. That is, the variable N only exists for the call to the STUDY_DAY function. N is used to hold intermediate values when the function is called. The variable N in the function and a variable named N in a DATA step are different variables that share the same name. Changing one does not change the other. This is a key protection of functions. Functions cannot modify their parameters and cannot change DATA step variables in a DATA step that calls the function.

The code to call the STUDY_DAY function from a DATA step is here:

```
options cmplib=sasuser.funcs;
data _null_;
  start = '15Feb2006'd;
  today = '27Mar2006'd;
  sd = study_day(start, today);
  put sd=;
run;
```

STUDY_DAY is called from DATA step code like any built-in SAS function. When the DATA step encounters a call to STUDY_DAY, it does not find this function in its built-in library of functions. When this occurs, it searches each of the data sets specified in the CMPLIB system option for a package that contains STUDY_DAY. In this case, it finds STUDY_DAY in Sasuser.Funcs.Trial and calls the function, passing in the variable values for START and TODAY. The result of STUDY_DAY is assigned to the variable SD.

Writing a function protects programmers from common mistakes like omitting an ampersand on a variable name or generating huge DATA steps by using %DO loops within a DATA step. Functions make reading, writing, and maintaining programs easier with variable scope and the ability to reuse code without replicating it throughout a DATA step.

FUNCTIONS VERSUS MACROS

Writing a function or CALL routine with PROC FCMP should be considered when:

- A computation that returns one or more values needs to be performed in more than one location.
- Mixing DATA step and Macro syntax make the code illegible.
- A computation requires several intermediate values. Using local variables to store intermediate values protect values outside the function from being changed.
- DATA step logic would be useful in a PROC SQL computed column or WHERE clause.
- Consolidating several functions and expressions into one function to simplify SAS coding.
- Performing a computation that requires a DO loop. For performance, a DO loop should be used instead of a %DO loop.

Writing a Macro should be considered when:

- Grouping several SAS DATA steps and procedures into one program unit.
- There is a need to parameterize a group of several SAS steps.

OTHER PROC FCMP FEATURES

This section summarizes what an FCMP routine is and why someone would want to use them. There are several items that we don't mention that are included in Secosky (2007) and SAS Institute (2008a). Here is a short list of PROC FCMP features that we've not shown.

- Encrypted program storage
- A message that writes a SAS log message indicating the data set a function is found in.
- FCMP supports resizable, dynamic arrays.
- FCMP can easily call C and C++ routines.
- FCMP routines can be called from DATA step, PROC SQL, WHERE clauses, and the Graph Template Language. Calling from %SYSFUNC is coming in the "Phase 2" release of SAS 9.2.
- FCMP includes a RUN_MACRO function that immediately executes a macro from an FCMP routine. This is important because it allows a DATA step to execute a PROC or DATA step, then continue executing the step.

UNPUT

When data is stored in an external database, it can be costly to pull a large table into SAS to execute a SQL query. Executing as much of the query as possible in the database usually makes the query execute faster. The theme for many of the SAS 9.2 PROC SQL optimizations is improved SAS client and solution performance with external databases. The primary way to improve performance to external databases is to have more or the entire query executed in the database. PROC SQL implicit pass-though (IP) technology is used to pass SQL queries to the database (via SAS/ACCESS® products). Because performance with external databases is tied to IP, IP has evolved to play a critical role in the success of SAS clients and solutions.

When SAS clients and solutions work with external databases, they depend on SQL queries being passed down and executed within those databases. Typically, this is accomplished only if the SQL query is free of SAS specific syntax that the external databases cannot understand. PROC SQL already contains technology that allows the procedure to implicitly pass the SQL query to the database if it is free of such syntax and meets other criteria.

SAS Business Intelligence software, such as SAS Web Report Studio and SAS Enterprise Guide®, generate SQL queries to filter and return formatted data. The formatted data values are used to populate tables and graphs in reports SAS software presents to users. The PUT function is used in SQL queries to describe how PROC SQL is to format the resulting values that are used in WHERE or HAVING clauses.

Although the PUT function is specific to SAS software, it is useful when querying data from databases via SAS/ACCESS engines, as well as from SAS data sets. When accessing database tables, the SAS/ACCESS engine fetches the data from external databases into the PROC SQL process space where the formatting work for the PUT function is performed. Any SQL query that contains a reference to the PUT function must first fetch the data into SAS in order to perform the query. If table sizes are large, say millions or hundreds of millions of observations, the demands on the system (computers and network) to perform such a query will be correspondingly large. Not only does the increased response time required to fetch the data translate into greater expense for the customer, the additional disk and memory space to store the data adds to the expense as well.

As the integration of SAS software with external databases expands and the popularity of using formatted data increases, there is an increased probability that queries will not pass to the database. In light of these performance problems, SAS customers need a new technique to reduce the size and time constraints of performing such queries.

The unPUT technology described in this paper allows programmatic transformation of many PUT functions into an alternate SQL syntax that can be passed to a third-party database. This new technique will reduce the following:

- resource requirements by helping to limit the amount of data that must be fetched
- response time for the query

In summary, unPUT technology solves the performance problems noted by transforming the PUT function, when possible, into a different expression that can be passed to external databases. For the initial implementation of this technology, developers targeted PUT functions in the SQL WHERE and HAVING clauses. Much of the text and concepts of this section are taken from Whitcher (2008).

The following are examples of targeted queries that are accompanied by the transformed equivalent.

QUERY THAT INCLUDES A USER-DEFINED FORMAT

This first example employs a user-defined format to query for certain size categories.

```
proc format;
   value udfmt 1-3='small' 4-6='medium' 7-9='large';
run;

proc sql;
   select style as SmallStyles from db.clothes
      where put(size, udfmt.) = 'small';
quit;
```

As written, the query cannot be passed to the database for execution because it uses a PUT function that contains a user-defined format. During the PROC SQL planning process, unPUT technology recognizes the expression **put(size, udfmt.) = 'small'**. The unPUT optimization looks up the definition of the udfmt format to find the allowed values for the label 'small' (1-3 in this example) and then constructs a new WHERE clause:

```
(1 <= size and size <= 3)
```

unPUT automatically parses the new expression and reinserts it back into the in-memory structure (known as the SQL tree) that is used to process the query. Query processing continues and, with the PUT function removed, the query can now be passed to the database. As a result, it is likely that fewer records are returned, resulting in increased performance and processing of fewer records by PROC SQL. The final query that unPUT passes to the database looks similar to this example:

```
select style as SmallStyles from db.clothes
   where (1 <= size and size <= 3);
```

If the FEEDBACK option is specified on the PROC SQL statement, the transformed SQL query is output to the SAS log. This enables programmers to see how unPUT transforms their queries. While the FEEDBACK option shows the transformed query, it does not show the query that is passed to a database. The SASTRACE and SASTRACELOC system options show queries that are passed to a database.

QUERY THAT COMPARES FORMATTED DATES

Comparing formatted dates is a very common operation. This query shows how unPUT simplifies the comparison of formatted dates. For example, this query helps examine a population sample of individuals who were born on New Year's Day. By typing the date into a GUI, the query that is generated looks like this.

```
proc sql;
   select name from db.employees where
   (put(birthday, date5.) = '01JAN');
quit;
```

There is no user-defined list of values to substitute for the label '01JAN' in this example. Instead, the unPUT optimization makes use of other functions, like DAY and MONTH, to perform the transformation. The expression, (put(birthday, date5.) = '01JAN'), is changed to the following:

```
(MONTH(birthday)=1 AND DAY(birthday)=1)
```

The SAS SQL IP technology maps the MONTH and DAY SAS functions to their equivalent database functions, like EXTRACT in Oracle®, in order to perform the proper operation. Again, unPUT has transformed a query with formatted data into a query that can be passed to the external databases.

Note: Each database has its own set of functions. The SAS/ACCESS engines map SAS functions to their equivalent database function. Check SAS Institute (2008b) to see which functions are supported in this mapping process.

QUERY THAT INCLUDES USER-DEFINED FORMATS AND INEQUALITIES

This final query illustrates a more complex transformation that involves FORMAT procedure style formats and inequalities. The following example comes from a field request to pass a query which involves inequalities to the database:

```
proc format;
   value nudfmt
      0='RED'
      1='REDHEAD'
      2='NOTRED'
      3='GREEN'
      other='BLACK';
   run;

proc sql;
   select * from db.data_i
      where (index(put(color, nudfmt.), "RED") > 0);
quit;
```

UnPUT generates the following query transformation:

```
select * from db.data_i
   where color IN(0,1,2);
```

This example highlights the necessity of using PROC FORMAT's OTHER= option. In order to render expressions that are mathematically sound, unPUT must know the full range of possible outcomes for a user-defined format. If the OTHER= option is not present, a call to the PUT function for a column value that does not match results is a character string of that column value, whether the input column is numeric or character.

In the following example, if color is 5, and there is no OTHER= for nudfmt, the PUT function below returns the character string '5'. That is, column value 5 does not match one of the defined values in the nudfmt format (1, 2, or 3), so the number is converted to its character-string equivalent.

```
put(color, nudfmt.)
```

Without the OTHER= option, it is impossible to use unPUT for a PUT expression that uses PROC FORMAT style formats. By adding the OTHER= option to the PROC FORMAT definition, the format now has a default label to which it can refer for non-matching values.

Experience with internal testing suggests that it is atypical for SAS programmers to include the OTHER= option when defining PROC FORMAT style formats. For customers to utilize this technology, it may be necessary to retrofit the OTHER= option to their PROC FORMAT style definitions.

There is a new message that appears in the SAS log when the unPUT optimization fails because the PROC FORMAT- style format does not include the OTHER= option.

```
NOTE: Optimization for the PUT function was skipped because
the referenced format, NUDFMT, does not have an OTHER= range
defined.
```

Customers who encounter this new message and want to perform unPUT optimization, should add the OTHER= option to their PROC FORMAT format definition. For examples of using the OTHER= option, see SAS Institute (2008c).

OTHER UNPUT FEATURES

UnPUT supports many types of PUT function expressions, including those in the following list:

- Equals, put(birthday, date9.) = '01JAN2000';
- Not equals, put(birthday, date9.) ^= '01JAN2000';
- Variable expressions, put(anniv+30, date9.) = '11Feb2000'
- Expressions using inequalities, index(put(color, nudfmt.), "RED") > 0
- IN, NOT IN clauses, put(size, udfmt.) in ('small', 'large')
- LIKE, NOT LIKE clauses, put(size, udfmt.) like "small"
- CONTAINS, NOT CONTAINS clauses, put(type, udfmt.) contains "s"
- BETWEEN, NOT BETWEEN clauses, put(size, udfmt.) between 'small' and 'large'
- STRIP or TRIM(LEFT()) functions, strip(put(x, udfmt.)) in ('small', 'medium')
- UPCASE function, upcase(put(bday, date5.)) = '01JAN'

SUMMARY

Use of the unPUT optimization is intended to ensure that external databases perform a greater portion of the sub-setting for queries that SAS clients and solutions generate. For example, running a query with the SAS Web Report Studio query cache turned on should yield a configuration whereby the initial sub-setting query is sent to the database, and follow-up queries, for individual tables and graphs within a report, can execute queries against cached data. These circumstances result in better response times for the user.

LINGUISTIC COLLATION

Computer programmers are somewhat accustomed to the binary or TRANTAB collation sequence of PROC SORT. A binary collation sequence is one that orders characters based on the code point order of an encoding. A TRANTAB collation sequence orders characters based on the order in a SAS translation table. For non-programmers, these methods are not intuitive. That is, these methods do not produce sequences that we associate with printed materials like dictionaries, phone books, and book indexes. The collating sequences used in print were developed and established hundreds of years before the advent of the computer and have become culturally ingrained. We find that locating a word in the dictionary, a name in the phonebook, or an entry in an index is a relatively simple task because we know, or at least have a good idea, of how the items are arranged.

Locating items within a computer-generated report, on the other hand, might be difficult if we do not know the collating sequence used for items in the report. For instance, we may not immediately realize that words beginning with uppercase letters are listed separately from words beginning with lowercase letters or that words beginning with accented characters are listed after words beginning with unaccented characters. Globally, different cultural expectations have developed with respect to collating sequences in printed materials for the multitude of languages and writing systems that exist. Linguistic collation addresses these cultural expectations and produces sequences that are deemed reasonable and acceptable throughout the various countries and regions of the world.

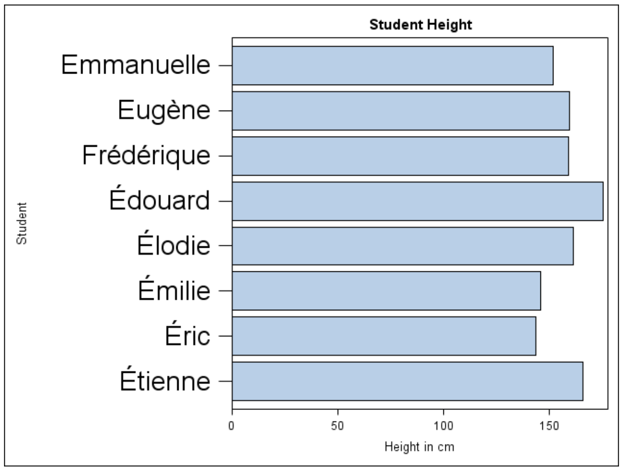
In SAS 9.2, PROC SORT is enhanced to provide linguistic collation. Setting the SORT procedure's SORTSEQ= option to LINGUISTIC causes PROC SORT to collate linguistically in accordance with the SAS System LOCALE setting. Options for modifying properties of the linguistic collating sequence may be specified within parentheses following the LINGUISTIC option value.

Linguistic collation is useful when a more intuitive or culturally correct sequence is desired for report generation or other data presentation, as well as for compatibility between systems. Linguistic collation is available for the 100+ locales and 50+ languages supported within SAS 9.2. Use of linguistic collation requires additional computational time, memory, and possibly storage space.

This section of the paper looks at what linguistic collation can do with accented characters, with words that contain blanks or punctuation, and how numbers in character columns can be sorted numerically. Text and concepts for this section are taken from Bridgers and Mebust (2007).

ORDERING ACCENTED CHARACTERS

In many countries, accented characters are prevalent. When performing a binary collation of accented characters in a Windows Latin-1 encoding with PROC SORT, the accented characters come after all the unaccented characters. In most languages, this is unintuitive and incorrect. Our next example sorts French names that contain accented characters. Figure 1 uses a binary collation, where "`Frédérique`" sorts before "`Édouard`", because "`F`" occurs before "`É`" in ASCII-based encodings. This is incorrect. The correct ordering is in Figure 2, where "`Édouard`" comes before "`Frédérique`".



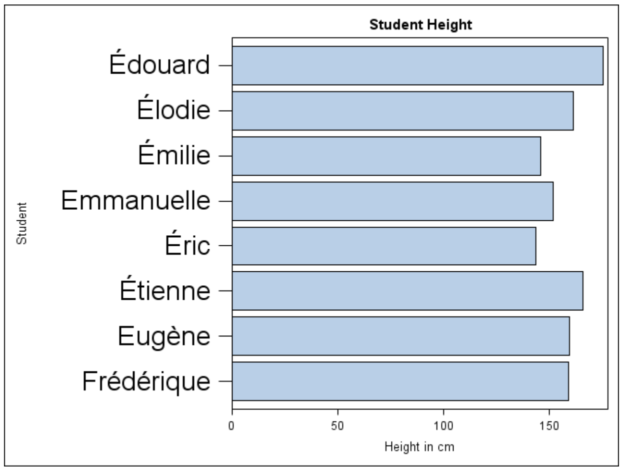


Figure 1: Binary Collation **Figure 2:** Linguistic Collation

Figure 2 uses linguistic collation with PROC SORT. The code to correctly sort French names is here:

```
proc sort data=class
          out=sorted
          sortseq=linguistic(locale=fr_FR);
  by name;
run;
```

We set the PROC SORT SORTSEQ= option to LINGUISTIC to use linguistic collation. If the SAS session locale isn't fr_FR, for France French, then we need to set the locale for lnguistic collation to use. This is done with the LOCALE= option in parentheses after the LINGUISTIC keyword. The point is linguistic collation is powerful and simple to code.

How does linguistic collation sort values in a culturally correct way? Linguistic collation uses a five step process when comparing two character variable values. First, all characters are compared based on their alphabetic base. That is, accent marks are removed and case is not considered when comparing the character strings. For values that compare the same after this first step, accent marks are considered according to the rules of the locale. If values still compare the same after the second step, case is used to distinguish between them. The fourth step considers punctuation characters and the fifth step compares characters by their Unicode code point values. More information about how linguistic collation occurs can be found in SAS Institute (2008d).

From speaking with customers in the United States, many do not need to program applications that handle accented characters. For these applications, what does linguistic collation have to offer? The next two sections show how linguistic collation handles character case, spaces and hyphens, and numbers coded in character values.

CONSIDERING CASE WHEN SORTING

To demonstrate how linguistic collation sorts values differently than the default binary collation or collation with a TRANTAB, consider the table of words below. The table contains three columns, each demonstrating a different type of collation.

The first column shows the sequence of words that is obtained using a binary collating sequence on words that are represented in an ASCII based encoding. The second column shows the same list of words in a sequence obtained with a TRANTAB collation using a translation table named LOWFIRST. This translation table was designed to alternate the ordering of lowercase and uppercase letter such that '`a`' < '`A`' < '`b`' < '`B`' < … < '`z`' < '`Z`'. The third column shows the results of linguistic collation of the same list of words.

In the first column, there is an obvious lack of alphabetic ordering due to the separate grouping of words beginning with uppercase and lowercase letters. In this column, the word "`Zeus`" incorrectly appears before "`aardvark`". This ordering is due to the code points assigned to the characters within the ASCII-based encoding.

In the second column, we have tried to correct this problem by using a translation table that assigns weights that are very similar to the lowercase and uppercase character variants. Using this table, by specifying SORTSEQ=LOWFIRST, we can cause "`aardvark`" to appear before "`Zeus`". However, this TRANTAB collation causes the string "`azimuth`" to appear before "`Aaron`" because the lowercase character '`a`' is assigned a weight that is less than the weight of the uppercase character '`A`'. Regardless of the assignment of weights in the translation table, it is not possible to achieve a true alphabetic ordering that takes the character case into account. This problem is inherent in the use of a single-level weighting algorithm like TRANTAB collation.

In the third column, we see that a proper alphabetic ordering is achieved, in accordance with the en_US, United States English, locale, using a multi-level linguistic collation algorithm. This particular collating sequence specifies that, after the alphabetic order is established, words beginning with lowercase letters appear before words beginning with uppercase letters.

Obs	Binary	Lowfirst	Linguistic
1	Aaron	aardvark	aardvark
2	Aztec	azimuth	Aaron
3	Zeus	Aaron	azimuth
4	aardvark	Aztec	Aztec
5	azimuth	zebra	zebra
6	zebra	Zeus	Zeus

SPECIAL HANDLING OF SPACES AND PUNCTUATION

When words contain spaces or punctuation, we continue to see unintuitive sorting with binary collation. Words that contain spaces or punctuation are common with city names and hyphenated last names. Using linguistic collation, we can achieve intuitive sorting. Setting the linguistic collation option ALTERNATE_HANDLING= to SHIFTED, spaces and punctuation are of less importance when comparing two character strings.

Consider the two graphs below. Figure 3 uses binary collation. Figure 4 uses linguistic collation with ALTERNATE_HANDLING=SHIFTED.

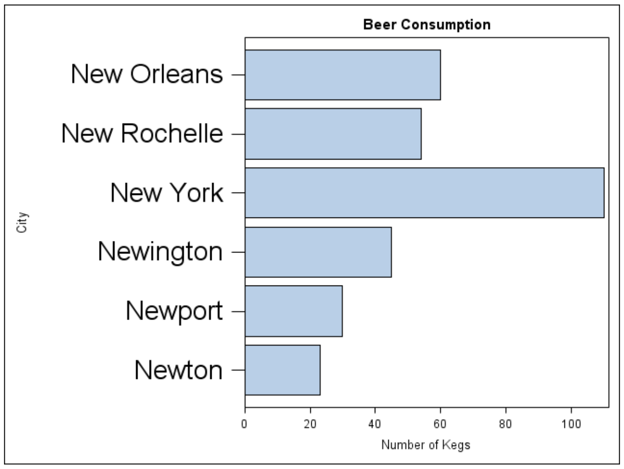


Figure 3: Binary Collation

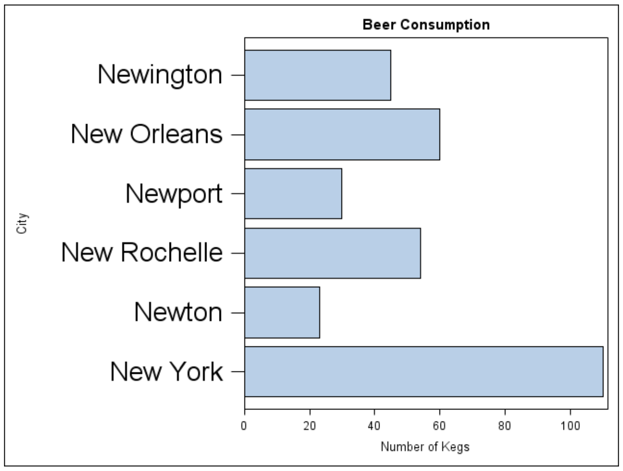


Figure 4: Linguistic Collation with ALTERNATE_HANDLING=SHIFTED

With the binary collation, all names that start with "New" and a space are sorted into one group, resulting in an ordering that is not the same as found in an atlas index. Setting ALTERNATE_HANDLING=SHIFTED causes spaces and punctuation to be essentially ignored when ordering the values. The result is a sort that is the same as an atlas index. A similar operation would occur if the terms were last names that could be hyphenated. The code to perform the linguistic collation is here:

```
proc sort data=consumption
          out=sorted
          sortseq=linguistic(alternate_handling=shifted);
  by city;
run;
```

SORTING NUMBERS IN CHARACTER VALUES

When performing a binary collation on numbers that appear in a character variable, unintuitive results occur. For instance, the values "1", "2", "10", "20" are sorted as "1", "10", "2", "20". The reason is that "1" occurs before "2" in ASCII and EBCDIC collating sequences, so all numbers beginning with a "1" come before all numbers beginning with a "2". Linguistic collation can scan the values being sorted for a series of digit characters and treat the digit characters as numbers, resulting in a more intuitive sort.

The following example shows the ordering of addresses with binary collation and linguistic collation. To treat a series of digit characters as a number, the linguistic collation option NUMERIC_COLLATION is set to ON.

Obs	Binary	Linguistic
1	0123 Main St. Apt #12	123 Main St. Apt #1
2	123 Main St. Apt #1	123 Main St. Apt #2
3	123 Main St. Apt #103	0123 Main St. Apt #12
4	123 Main St. Apt #2	123 Main St. Apt #24
5	123 Main St. Apt #24	123 Main St. Apt #103

In all the addresses, the street number is the same, even though one of the addresses has a typo with a street number that begins with a zero. In each case "123" or "0123" is treated as the number 123. The apartment numbers are also compared as numbers. With binary collation, "Apt #103" comes before "Apt #2" because the code point for "1" comes before the code point for "2". With linguistic collation, "103" is treated as the number 103 and "2" is treated as the number 2, numerically, 2 comes before 103, so Apt #2 comes before Apt #103 when using linguistic collation.

SUMMARY

When producing reports with SAS, binary or TRANTAB collation does not always result in an intuitive or culturally correct order. Linguistic collation, provided by PROC SORT in SAS 9.2, results in a more intuitive ordering. Currently, linguistic collation is only explicitly supported in PROC SORT although the new SORTKEY function allows some of linguistic ordering functionality to be used elsewhere, such as in PROC SQL or the DATA step. There is an option with linguistic collation to make spaces and punctuation less important, which is important when sorting names. In addition, numbers that appear in character values can be sorted as numbers, avoiding orderings where larger numbers appear before smaller numbers.

NEW DATA STEP FEATURES

SAS publishes the SASware Ballot to gauge popularity for new feature requests. The SASware Ballot is an annual survey that SAS users can take at support.sas.com. Users vote on the features they would like to see in SAS. This section summarizes DATA step feature requests that were top 10 vote getters on the SASware Ballot.

DATA SET LISTS

Reading a list of like-named data sets in the DATA step is tedious to program. For instance, to sequentially read the data sets A1 to A50, one would have to type 50 data set names on the SET statement, like this:

```
data out;
  set a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18
      a19 a20 a21 a22 a23 a24 a25 a26 a27 a28 a29 a30 a31 a32 a33 a34
      a35 a36 a37 a38 a39 a40 a41 a42 a43 a44 a45 a46 a47 a48 a49 a50;
```

Most programmers wouldn't type all of these names in. Instead, they'd resort to a macro, further cluttering their DATA step program. Having the ability to read from a list of data sets was the #1 vote getter on the 2006 SASware Ballot. Data set lists are implemented as dash lists and colon, or prefix, lists in SAS 9.2. We first look at dash lists, then colon lists.

The code to sequentially read the data sets A1 to A50 with a dash lists is here:

```
data out;
  set a1-a50;
```

The dash list expands to all of the data sets from A1 to A50. If one of the data sets does not exist, an error occurs. Data set options, like WHERE=, KEEP=, and PW=, are applied to all of the data sets in the list by placing them in parentheses after the second data set name of the dash list.

In the prior example data sets are read from the WORK library. To read from another library, the libref is specified on both data set names. For example, to read the data sets A1 to A50 from the PERM library and apply a WHERE= data set option to each data set, one would code:

```
data out;
  set perm.a1-perm.a50(where=(speed > 100));
```

Colon lists assist when reading from data sets that start with the same prefix or when all of the data sets to be read are not known. For instance, to read a group of eight fuel related data sets, a SET statement would be coded as:

```
data results;
  set gas_price_option
      gas_rbid_option
      gas_price_forward
      gas_rbid_forward
      coal_price_option
      coal_rbid_option
      coal_price_forward
      coal_rbid_forward;
```

Using a colon list, a common prefix is typed and a colon is placed after the prefix. Equivalent syntax using colon lists is here:

```
data results;
  set gas: coal:;
```

This DATA step reads all the data sets in the WORK library that begin with GAS and all those that begin with COAL. Data set options can be placed in parentheses after the colon and a libname can be specified before the prefix to read from a library other than the WORK library. Also, when the programmer coded this step, he/she did not need to know how many data sets are read, only that he/she wants to read all of the data sets with a particular prefix. Both colon and dash lists also work with the MERGE statement.

Both dash and colon lists make programming easier in SAS by requiring less typing and use of macro to implement these syntactic features.

RETRIEVING INPUT DATA SET NAME

When using a SET statement to read from a series of data sets, it is difficult to determine which data set contributed the observation that was last read. This information may be needed if processing the observation depends on data coded in the data set name. In the last example where we read from fuel data sets, the fuel type may cause different processing to occur in the DATA step.

In SAS 9.1, to determine the data set that contributed the current observation, the IN= data set option is used in concert with a series of IF..THEN..ELSE statements. This is tedious. In SAS 9.2, the INDSNAME= option on the SET statement fills a DATA step variable with the name of the data set that contributed the current observation. INDSNAME is an acronym for INput Data Set NAME. Using the code from the prior example that reads all the data sets that begin with GAS and COAL, here is how the INDSNAME= option is used to get the name of the data set that contributed the current observation.

```
data results;
  set gas: coal: indsname=current_dataset;
```

This program defines a character variable named CURRENT_DATASET and fills it with the name of the data set that contributed the current observation. For instance, if the first row is read from GAS_PRICE_OPTION, then the value of CURRENT_DATASET is set to "WORK.GAS_PRICE_OPTION". The value in CURRENT_DATASET can be parsed with the SCAN function to determine the fuel type or to pull any other information from the data set name.

The INDSNAME= option is a user requested feature that eases determining the data set that contributed the current observation.

STRING DELIMITER

One of the most powerful aspects of the DATA step is reading data from text files. Complex formatting can usually be read with a minimal amount of code. The DELIMITER= option on the INFILE statement specifies one or more distinct characters to use as a delimiter when reading values from a text file. If multiple characters are specified on the DELIMITER= option, for example DELMITER=",/+", the DATA step treats a comma, a slash, or a plus as a delimiter. It does not treat the string ",/+" and as a multi-character delimiter.

In SAS 9.2, the DLMSTR= option on the INFILE statement enables programmers to specify a multi-character string as a delimiter. DLMSTR stands for "delimiter string." For example, DLMSTR="---" will treat three dashes as a delimiter. Without the DLMSTR= option, one usually reads an entire line into a variable and calls functions to pick the line apart to correctly parse string delimiters. The next paragraph steps through an example using DLMSTR=.

A customer wanted to parse a log file produced by a network firewall. The firewall placed quotes around the values and delimited them with two spaces. Some of the values could contain one or more consecutive spaces. Upon first glance, one might start to code by setting the delimiter to a space and use the DSD option to remove the quotes from the values and handle values that could contain the delimiter. Here is the code:

```
data bad_guys;
  length date time ip $ 16;
  infile datalines dlm=' ' dsd;
  input date time ip;
datalines;
"20APR2007"  "12:56:46"  "146.16.1.23"
;
```

The program works, except when DSD is used, two consecutive delimiters are treated as a missing value. The variable DATE is filled with the date, TIME is set to missing (all blanks) because there are two consecutive spaces, and IP is set to the time value. The way that DSD treats consecutive delimiters as a missing value makes it difficult to parse this log.

The DLMSTR= option allows us to set the delimiter to two blanks and use DSD to parse the data. Here is the program that correctly parses the log:

```
data bad_guys;
  length date time ip $ 16;
  infile datalines dlmstr='  ' dsd;
  input date time ip;
datalines;
"20APR2007"  "12:56:46"  "146.16.1.23"
;
```

In this case, DATE is filled with the date value, the delimiter is two blanks, so a single delimiter is encountered and TIME is filled with the time value. Another delimiter is seen and IP is set to the IP address.

The DLMSTR= option supports specifying a character variable after the equal sign. The value in the variable is used as the delimiter. The DLMSOPT= option on the INFILE statement is another new option that works in connection with DLMSTR=. DLMSOPT= allows programmers to trim trailing blanks from the value specified on the DLMSTR= option and/or specify that a case-independent comparison occur when scanning the input record for a delimiter.

DLMSTR= is also supported on the FILE statement, enabling users to write data with multi-character string delimiters. We refer the reader to SAS Institute (2008e) to learn about more details of DLMSTR= and DLMSOPT=.

The DATA step has powerful text parsing abilities. In SAS 9.2, we extend those capabilities by enabling string delimiters to be used with the DLMSTR= option. DLMSTR= simplifies code when reading string delimiters, making it easier to write SAS code with fewer statements.

ADDITIONAL FEATURES

This paper outlines some of the new features in Base SAS for the DATA step, PROC SQL, and PROC SORT. SAS Institute (2008f) is a more complete list of new features. Some of the new features not presented in the paper are:

- **DATA Step Javaobj**
  The DATA Step Javaobj enables SAS programmers to instantiate a Java class in a DATA step and call methods and access attributes of the class. The Javaobj is helpful when interfacing SAS with a third-party library of Java methods. The Javaobj is available as an experimental feature in SAS 9.1.3. See SAS Institute (2008g) for more information about the Javaobj.

- **Hash Object Additions**
  The DATA step hash object is able to store duplicate keys, which enables SQL-like joins to occur among DATA step statements all in one step. Other new additions to the hash object are the abilities to count the number of times a key is found and to clear a hash object. Being able to clear a hash object makes processing BY groups in a hash object faster and easier. See Ray and Secosky (2008) for more information about hash object additions.

- **Faster SELECT DISTINCT and COUNT DISTINCT.**
  When possible, PROC SQL uses an in-memory hash table to compute SELECT DISTINCT and COUNT DISTINCT operations. Using in-memory storage enables faster distincting operations by not having to use a disk-based storage scheme. See Whitcher (2008) for more information.

- **Encrypted Stored Macros**
  The SECURE option on the %MACRO statement causes macros to be encrypted before storing them in a catalog entry. Encrypted storage makes it less likely for someone to see the SAS code that could be generated by a macro. See SAS Institute (2008h) for more information about encrypted macro storage.

CONCLUSION

SAS is a wonderful language to program in. In SAS 9.2, we've added many user requested features to ease coding and improve the performance of SAS. This paper outlines many of the new features in the DATA step, PROC SQL, and PROC SORT. Please try these new features and let us know how they work for you. You are going to find SAS is easier to program than ever.

REFERENCES

Bridgers, Michael and Mebust, Scott. 2007. "Creating Order Out of Character Chaos: Collation Capabilities of the SAS® System." *Proceedings of the First SAS Global Forum*. Cary, NC: SAS Institute Inc. Available at http://www2.sas.com/proceedings/forum2007/297-2007.pdf.

Ray, Robert and Secosky, Jason. 2008. "Better Hashing in SAS 9.2." *Proceedings of the Second SAS Global Forum.* Cary, NC: SAS Institute Inc. Available at http://support.sas.com/rnd/base/datastep/dot/better-hashing-sas92.pdf.

SAS Institute Inc. 2008a. Base SAS, Base SAS Procedures Guide, Procedures, The FCMP Procedure. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/proc/59565/HTML/default/a002890483.htm.

SAS Institute Inc. 2008b. SAS/ACCESS Software. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/onlinedoc/access/index.html.

SAS Institute Inc. 2008c. Base SAS, Base SAS Procedures Guide, Procedures, The FORMAT Procedure. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/proc/59565/HTML/default/a000063536.htm.

SAS Institute Inc. 2008d. SAS 9.2 National Language Support, Reference Guide, Collating Sequence Option. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/nlsref/59554/HTML/default/a002613572.htm.

SAS Institute Inc. 2008e. Base SAS, Base SAS Language Reference Dictionary, Statements, INFILE Statement. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/lrdict/59540/HTML/default/a000146932.htm.

SAS Institute Inc. 2008f. What's New in SAS 9.2, Base SAS, SAS Language Features. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/whatsnew/61982/HTML/default/lrdictwhatsnew902.htm.

SAS Institute Inc. 2008g. The Java Object and the DATA Step Component Interface. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/rnd/base/datastep/dot/javaobj.html.

SAS Institute Inc. 2008h. SAS 9.2 Macro Language: Reference, Macro Language Dictionary, Macro Statements, %MACRO Statement. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/mcrolref/59526/HTML/default/macro-stmt.htm.

SAS Institute Inc. 2008i. SAS/GRAPH(R) 9.2: Graph Template Language Reference. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/grstatgraph/60787/HTML/default/main_contents.htm.

Secosky, Jason. 2007. "User-Written DATA Step Functions." *Proceedings of the First SAS Global Forum.* Cary, NC: SAS Institute Inc. Available at http://www2.sas.com/proceedings/forum2007/008-2007.pdf.

Whitcher, Mike. 2008. "New SAS® Performance Optimizations to Enhance Your SAS® Client and Solution Access to the Database." *Proceedings of the Second SAS Global Forum.* Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/sgf2008/optimization.pdf.

ACKNOWLEDGMENTS

The author would like to thank Mike Whitcher, Scott Mebust, and Michael Bridgers for letting him borrow text, examples, and concepts from their papers on SAS 9.2 new features. The bar graphs in the paper were produced with the SGRENDER procedure using a TEMPLATE procedure template written in the Graph Template Language. PROC SGRENDER and the Graph Template Language are part of SAS/GRAPH and are new in SAS 9.2. See SAS Institute (2008i) for more information about PROC SGRENDER and the Graph Template Language.

CONTACT INFORMATION

Your comments, questions, and success stories are valued and encouraged. Contact the authors at:

Jason Secosky
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-677-8000
Jason.Secosky@sas.com