

A SAS[®] Primer for Healthcare Data Analysts

Christopher W. Schacherer, Clinical Data Management Systems, LLC
Brent D. Westra, Mayo Clinic Health Solutions

ABSTRACT

As in other fields, analysts in healthcare come to their vocation from a variety of paths—statisticians with formal training in predictive modeling, IT professionals with experience in programming and report writing, billing and claims professionals who possess a wealth of subject-matter expertise, and accounting and finance professionals who understand the methodologies for valid reporting of financial results. Analytics and Business Intelligence groups within healthcare organizations are strengthened by this diversity of expertise, but individuals coming into a business intelligence or healthcare analytics department from these other areas often struggle to acquire the skills that will help them leverage their particular backgrounds against the analytic challenges they face. The current work focuses on helping these analysts acquire the SAS programming skills necessary to turn raw data into analysis-ready data sets. The intention is not to train analysts in the finer points of each PROC and variation on DATA STEP and MACRO programming, but to provide a basic understanding of SAS concepts using specific techniques that will be useful to them in their attempts to create analytic datasets from their source data.

INTRODUCTION

Having worked with both experienced and inexperienced analysts in the healthcare analytics field, we have identified a number of SAS concepts that every healthcare analyst should learn and, more importantly, have identified a number of concepts and methodologies with which analysts new to SAS struggle when being introduced to SAS for the first time. We have compiled these concepts here as a framework for helping train analysts new to SAS and have attempted to provide sufficient depth to the examples to make them directly relevant to the reader, but without going into great depth about the background of the data sources. It is not our intention to provide in-depth training in the course of this brief manuscript, but to provide some focus points for further study and training. What you will find in the ensuing pages is an overview of the SAS development environment, a description of the basic layout of a SAS program, and a discussion of the procedures and programming methods that will enable a new SAS user to get started using SAS.

THE SAS DEVELOPMENT ENVIRONMENT

In our training for healthcare analysts, we begin by making sure that new users of BASE SAS first understand the environment in which they are working. The SAS development environment is made up of four main components: the EDITOR, LOG, and OUTPUT windows and the EXPLORER.

The EDITOR (or, Program Editor) window is where you will write the programs that perform your data management, analytic, and reporting tasks. The commands written here will import data files, extract data from relational database management systems, manipulate data to merge and reshape datasets, and execute analytic functions. As programs are run, SAS generates information about the execution of your code in the LOG window—identifying syntax errors in your code and describing the datasets being created. As analytic and reporting procedures are encountered during the execution of your code, the default behavior of SAS is to print the results of those procedures in the OUTPUT window. There are alternative output locations for the results of these procedures, but the EDITOR, LOG, and OUTPUT window make up the main three interactive components of the SAS programming environment. Finally the EXPLORER allows you to keep track of the data objects with which you are working, and, more importantly, allows a means of interacting with those datasets and variables by opening them in a data grid view and right-clicking on them to study their properties.

PARTS OF A SAS PROGRAM

Once analysts new to SAS are comfortable with these components, they are ready to start developing their first simple programs. We introduce them to the parts of a SAS program by discussing the Header, Library Assignments, Options and Program Body.

HEADER INFORMATION. Arguably the most overlooked topic in teaching someone to write SAS programs is that of the program header—which serves as a guide to help understand why a program was written, who wrote it, when, and what the program does. Your organization may have a standard for this information, so it is important to work together within your group to maintain consistent information that is clearly understood by everyone.

Header information is written at the top of the program as a comment (or series of comments)—using "/*" to signify the beginning of the comment and "*/" to signify the end of the comment. Text lying between the comment identifiers is ignored by SAS at the time your program is executed. You can also "comment out" individual lines in the body of your SAS program with an asterisk (*) which comments out all text until the next semicolon (;) is encountered.

```
/*
29MAY2008A - [CWS] This program maps pharmacy benefits data from PharmCo
              Health Associates source file v1.2.0 to data staging table
              "UHEALTH.PBM.RAW.PHARMCO.

15AUG2009A - [CWS] Updated mapping of pharmacy benefits source data to
              accommodate source file v1.3.0. (see Project DM00112 for
              specification.

*/
```

Some headers are more formalized than the previous example:

```
/*
AUTHOR:   Chris Schacherer

ORGANIZATION: XYZ Hospital, Somewhere, USA

DATE:    05/28/2009

DESCRIPTION: This program extracts data from the professional billing system and
              creates a dimensional data mart for use by analysts in the Decision Support
              Department.

DEVELOPMENT OS:  Windows XP, SP3
SAS VERSION:    9.1.2
*/
```

Whether using a more or less formal method of documenting the purpose of your programs, the key to writing effective header information is recording sufficient information to enable future users of the program to understand the original purpose of the program and any major revisions it has undergone. Combined with comments in the body of the program that explain the program logic, programmers that inherit the code will be able to successfully maintain, debug, and modify it throughout its lifecycle. After completing or updating the header section of a program, you will likely also add one or more LIBNAME statements--that define the locations where SAS will find and store the data with which you are working.

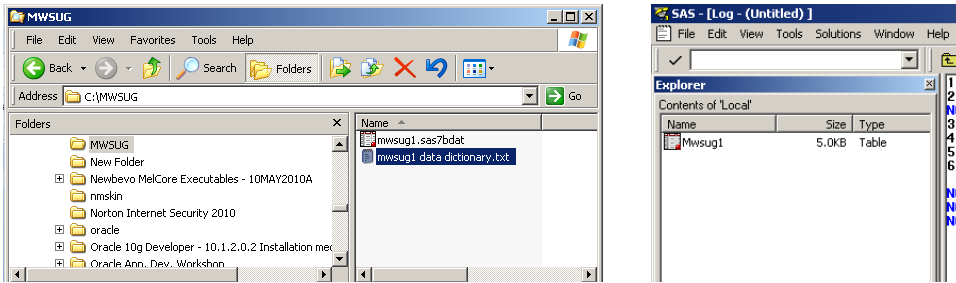
LIBRARY ASSIGNMENT. Libraries in SAS are pointers (or virtual locations) that SAS uses to know the location of the SAS objects with which you are working. These pointers can direct SAS to either a location on a hard drive or network or to a database, and are assigned in the SAS program using a LIBNAME statement.

The following LIBNAME statement will define the hard drive location "C:\MWSUG" as the library "local".

```
LIBNAME LOCAL 'C:\MWSUG';
```

Throughout the remainder of your SAS session (or until you execute another LIBNAME statement that reassigns the location of "LOCAL") SAS will recognize the location "C:\MWSUG" as the directory to which "LOCAL" refers. Whenever a reference to the library LOCAL is made, "C:\MWSUG" will be searched for the specified object. When you double-click on "LOCAL" in the Explorer, you will see the SAS data objects that are stored in "C:\MWSUG".

One characteristic of libraries that is sometimes confusing to new SAS programmers is that only recognized SAS data objects will appear in the Explorer window when a library is opened.



For example, in the Microsoft Explorer window on the left, we see that there are two files in "C:\MWSUG", but only one of them appears in the SAS Explorer window showing the contents of "Local". This is because the text file is not recognized as a native SAS data object with which it can work directly. Non-SAS datasets (e.g., Microsoft Excel files, text files, etc.) can be imported into SAS and converted to SAS datasets, but these file types are not recognized as data sources that can be directly accessed without some additional code.

As noted previously, SAS libraries are not limited to local directories. They can also be specified using mapped drive locations on your local system:

```
LIBNAME LUMEN = 'M:\Datasets';
```

However, for the sake of portability of your code from one computer to another, we recommend that libraries referencing network locations be defined using their universal naming convention (UNC) name:

```
LIBNAME LUMEN = '\\mfad\rchdept\MMSI\Finance\MMSI\Lumen AIR File\Datasets';
```

More frequently, however, source data are not just shared as files on a local drive or network directory, but reside in relational database management systems. In order to facilitate access to these systems, SAS allows users (via SAS/ACCESS) to connect directly to the databases to which they have access and read the associated database tables and views much as they would SAS datasets that reside on local and network drives.

In the following example, we define our SAS Library "hca" as a connection to the Microsoft SQL Server Database "UHHCA" maintained by "University Health's Health Care Analytics department. This library definition specifies that the database type is an Object Linking and Embedding (OLE) connection to the local system interface SQLOLEDB.1. The library specification further defines the "initial catalog" as the database "UHHCA" that is running on the SQL Server named "HOSP-HCA". The bulkcopy processor option (BCP=Yes) is being utilized (which will allow loads to the database tables to proceed without committing every <x> rows) and the database schema to which the SAS library "HCA" will attach is "DBO".

```
LIBNAME hca OLEDB PROVIDER=SQLOLEDB.1 REQUIRED=Yes
      USER = cschacherer
      DATASOURCE = "HOSP-HCA"
      PROPERTIES = ('initial catalog'=UHHCA 'Persist Security Info'=True)
      BCP = Yes
      SCHEMA = 'DBO';
```

In the next example, the library "billing" is defined as a connection to the "hospital_billing" schema on an Oracle database called "financials".

```
LIBNAME billing ORACLE
      USER = 'cschacherer'
      DPPROMPT = YES
      PATH = "financials"
      SCHEMA = hospital_billing;
```

It should be noted that the type of database to which one is connecting will dictate the parameters of the library definition. For example, in the Oracle connection there is a parameter "DBPROMPT" which determines whether the user will be presented with an Oracle login screen during the initial access to that library. In Microsoft SQL Server, one can specify whether to enable the bulk-copy processor for the connection or not. However, in all SAS/ACCESS connections to database management systems, the type of database to which one is attempting a connection (e.g., OLEDB, ORACLE, DB2, etc.) is specified following the library name and is followed by the parameters required by

SAS/ACCESS to make a connection to that particular database system. In addition, many database vendors have their own proprietary networking and client software that must be installed on the system running SAS in order to enable a connection to the database. For more detailed information on defining libraries as connections to databases, the reader is referred to Riley (2008) and SAS Institute, Inc. (2004a; 2010a) for discussions of SAS Access connections to Microsoft SQL Server and OLE DB Datasources, Rhodes (2007), Levin (2004), SAS Institute, Inc. (2004b), and Schacherer (2008) for SAS/ACCESS connectivity to Oracle, and Gona & Van Dyk (1998) for an earlier, but still very relevant, description of SAS/ACCESS and the LIBNAME statement. For more advanced treatments of SAS/Access programming efficiencies, see Levine (2001) and Helf (2002).

It should be noted that LIBNAME statements can be executed anywhere in your code, but it is best to assign all libraries that are used throughout your program at the beginning of the program so that all the libraries you are referencing can be found in one location, easily reviewed for accuracy, and updated with required changes.

Finally, in addition to libraries that you define, there is one library that is always defined for you by default—WORK. The WORK library is defined automatically every time SAS is launched. This temporary work space is created (as the default option during installation) in the Windows directory "C:\Documents and Settings\

OPTIONS. Once library names are specified, we are almost ready to dive into the programming that will result in our analytic datasets, but there is one more (optional) section that may appear before the data manipulation portion of our program—an OPTIONS statement. Because SAS is highly configurable by the end user, there are a number of configuration options that are set with defaults but can be overridden by the user. As a simple example, "LINESIZE" (the length of a line in characters) and "ORIENTATION" (of the output that appears in the OUTPUT window) can be changed with an OPTIONS statement:

```
OPTIONS LINESIZE=120 ORIENTATION=LANDSCAPE;
```

Although this is a simple example, there are many different options that can be used to change how SAS operates during execution of your program and your entire SAS session. In the following example, the option MSTORED directs SAS to search for compiled SAS macros (reusable units of code that can be called from your SAS program) in a catalog located in the library specified by the SASMSTORE option. By using these options, users working in a group environment can use shared libraries of commonly used SAS Macro programs to maintain consistency in their coding and save time by eliminating routine programming tasks. Similarly, FMTSEARCH is an option that specifies the order in which SAS libraries are searched for formats (instructions for applying labels to the values stored in a given variable).

```
LIBNAME formats 'c:\formats';  
LIBNAME macstore 'c:\macros';  
  
OPTIONS MSTORED SASMSTORE=macstore FMTSEARCH=(formats);
```

An OPTIONS statement, though, is "optional". It is intended to alter how SAS runs for the duration of your program or SAS session. Once the options are specified, your SAS program proceeds to the "body" of the program where the executable code is written.

PROGRAM BODY. The body of the program is where you spend the majority of your development effort. In fact, once the LIBRARY and OPTIONS statements that you use regularly are well-developed, you will find yourself copying and pasting those sections over in new programs. The body of the program, however, is where the programming logic that results in each deliverable dataset, analysis, or report will be written. This is where you will start using the wide array of SAS Procedures (or, "PROCS") that perform a variety of programming functions and the DATA STEP—the procedural programming methodology utilized in most SAS programs. The remainder of the paper, therefore, focuses on programming methods that are used in developing your SAS solutions.

INTRODUCTION TO THE DATA STEP

Arguably, the most common method for manipulating datasets in SAS is the DATA STEP. In DATA STEP programming, SAS processes source datasets row-by-row and transforms the data into one or more target datasets—creating new variables, transforming existing variables, and performing conditional operations necessary

to achieve the desired dataset.

In its simplest form, the DATA STEP involves specifying the target dataset you wish to create and using the SET statement to specify the source dataset(s) that you wish to use in creating the target dataset. The following DATA STEP creates a local copy (in the WORK library) of the dataset "encounters" pulled from the "encounters" table in the HCA database.

```
DATA work.encounters;
  SET hca.encounters;
RUN;
```

What happens between the DATA statement and RUN determines how the dataset in the source is transformed to create the target. For example, we might want to create a local dataset that contains only inpatient encounters, and we might wish to create a new variable called "total_charges" that represents the sum of professional and facility charges for each inpatient encounter. Assigning the value for total_charges simply involves setting the new variable equal to the arithmetic expression that defines it (assuming that both professional_charges and facility_charges are numeric variables...more on that issue in the next section). After the assignment of the total_charges value, a conditional "IF" statement is used to determine which records should be OUTPUT to the target dataset (inpatient_encounters). It should be noted that without an explicit OUTPUT statement, it is assumed (as was the case in the previous DATA STEP example that created the "encounters" dataset) that all records, once processed, are to be written to the target dataset. In this conditional case, however, only those records that meet the conditional criteria are output.

```
DATA work.inpatient_encounters;
  SET hca.encounters;
  total_charges = professional_charges + facility_charges;
  IF encounter_type = 'IP' THEN OUTPUT;
RUN;
```

Two other issues with this example should be noted. First, if a record has missing data associated with either or both of the variables "professional_charges" or "facility_charges", the value of "total_charges" will be a missing value. You will receive a warning alerting you to this fact in the SAS LOG:

```
114 DATA work.inpatient_encounters;
115   SET hca.encounters;
116     total_charges = professional_charges + facility_charges;
117     IF encounter_type = 'IP' /*for inpatient*/ THEN OUTPUT;
118 RUN;
```

NOTE: Missing values were generated as a result of performing an operation on missing values.

Each place is given by: (Number of times) at (Line):(Column).
210 at 116:43

NOTE: There were **253179** observations read from the data set HCA.ENCOUNTERS.

NOTE: The data set WORK.INPATIENT_ENCOUNTERS has **31738** observations and 35 variables.

Second, you can see the effect of the subsetting IF statement in the results reported in the log; 253,179 records were read from the source dataset, and 31,738 records were written out to the target dataset—so, assuming that 31,738 represents the number of records with encounter_type = 'IP', the conditional logic worked. One should note, however, that there is an alternative method for performing this record selection; one that hints at the inner workings of how SAS processes data. As shown in the following example, one can use a WHERE clause after the SET statement instead of the subsetting IF statement used in the previous example. This WHERE clause dictates "which" of the records from the source dataset will be passed on to the program data vector (PDV)—the area in memory where data are stored and manipulated during the processing of the current record. If a record from the dataset does not pass the initial test posed in the WHERE clause, it is never passed to the PDV for further processing. Once passed on to the PDV, however, processing continues in the order of the executable procedures within the program.

```
DATA work.inpatient_encounters;
  SET hca.encounters;
  WHERE encounter_type = 'IP';
```

```

LENGTH charge_cat $4.;

IF professional_charges = . THEN professional_charges = 0;
IF facility_charges = . THEN facility_charges = 0;

total_charges = professional_charges + facility_charges;

IF total_charges <= 0 THEN charge_cat = 'N/C';
ELSE IF 0 < total_charges <= 10000 THEN charge_cat = 'LOW';
ELSE IF 10001 <= total_charges <= 50000 THEN charge_cat = 'MED';
ELSE IF 50001 <= total_charges THEN charge_cat = 'HIGH';

KEEP mrn total_charges professional_charges facility_charges charge_cat;

RUN;

NOTE: There were 31738 observations read from the data set HCA.ENCOUNTERS.
      WHERE encounter_type not = 'IP';
NOTE: The data set WORK.INPATIENT_ENCOUNTERS has 31738 observations and 5
      variables.
NOTE: DATA statement used (Total process time):
      real time          0.15 seconds
      cpu time           0.04 seconds

```

You will notice in this LOG entry, that only 31,738 records were read from the 253,179 records in the source dataset—because only those records with `encounter_type = 'IP'` survived the WHERE clause test in the input buffer and made it on to the PDV for further processing. So, as far as the PDV was concerned, there were only 31,738 records ever encountered for processing. In addition to helping demonstrate how SAS processes records in the DATA STEP, using the WHERE clause instead of the subsetting IF can provide potential performance improvements—but see Gupta (2006), Lafler (2000), Howard (1999; 2003) and Gilson (1999) for further, very accessible discussions on performance improvement in the DATA STEP. For the many applications of the DATA STEP (perhaps, even the majority), the performance gains generated by application of this technique will be negligible. However, due to the size of most healthcare datasets we have included it here because you can reasonably expect to see performance gains in charge-detail, encounter, claims, and population health datasets.

Also in this example, you will notice that there is no LOG note alerting the user to the fact that some values of "total_charges" were set to missing (as there was in the preceding example). "Professional_charges" and "facility_charges" have been set to zero for those instances where their values evaluated to '.'. A period is used in SAS to denote a null value for a numeric variable; the character-variable equivalent is ". After evaluating the values of professional and facility charges and setting them to zero (where missing), "total_charges" will now have a calculated value for each record in the dataset. Finally, in this DATA STEP we decided to KEEP only the "mrn" (medical record number), "facility_charges", "professional_charges", "total_charges", and a new variable "charge_cat" (in order to reduce the storage size of the dataset and make it easier to visually review in the SAS VIEWTABLE). This results in only five variables being written to the dataset "inpatient_encounters". The converse of the KEEP statement is the DROP statement, and people consistently ask "which one should I use". The advice we give (a bit tongue-in-cheek) is that if you have a dataset of 100 variables and you only want three of them for your final dataset, use KEEP, if you want to write 97 of them to your dataset (but not the other three) use DROP.

Finally, this example introduces two other key concepts not found in the preceding example. First is the LENGTH statement, which determines the storage space for the variable "charge_cat" in the target dataset. In this case, we are creating a new variable that will have the possible values 'N/C', 'LOW', 'MED', and 'HIGH'. If we did not specify a length for charge_cat, the storage length would be determined by the first value written to "charge_cat"—a three-character string (\$3.) if the first value written was 'N/C', 'LOW', or 'MED' and a four-character string (\$4.) if the first value written to the dataset was 'HIGH'. Without specifying the format, there is a chance that instead of the value 'HIGH', we would end up with records assigned the value 'HIG' just because we encountered a value of 'N/C', 'LOW', or 'MED' before the first occurrence of 'HIGH'.

Second, there is an example of using "ELSE IF" to complete the syntax example of the IF-THEN/ELSE programming logic presented earlier. In this case, the computed variable "total_charges" is evaluated against non-overlapping intervals and each record is ultimately assigned an appropriate value of the variable "charge_cat" (charge category). Note that if the ranges did mistakenly overlap (e.g., 10001 – 50000 and 45000 – 65000) the value of "charge_cat" would be determined by the order of the ELSE IF statements—with the value of "charge_cat" being assigned based on the first ELSE IF that evaluated to "TRUE". For a record with "total_charges=45500", the following IF-THEN/ELSE

code block would assign the value of "MED" to "charge_cat,

```
IF total_charges <= 0 THEN charge_cat = 'N/C';
ELSE IF 0 < total_charges <= 10000 THEN charge_cat = 'LOW';
ELSE IF 10001 <= total_charges <= 50000 THEN charge_cat = 'MED';
ELSE IF 45000 <= total_charges THEN charge_cat = 'HIGH';
```

but this one would assign "charge_cat" the value "HIGH".

```
IF total_charges <= 0 THEN charge_cat = 'N/C';
ELSE IF 0 < total_charges <= 10000 THEN charge_cat = 'LOW';
ELSE IF 45000 <= total_charges THEN charge_cat = 'HIGH';
ELSE IF 10001 <= total_charges <= 50000 THEN charge_cat = 'MED';
```

Alternatively, if the following IF-THEN logic were used, each record would be evaluated against both IF-THEN conditions (in the absence of an ELSE IF) and would be assigned a value based on the last IF-THEN statement for which the record evaluated to 'TRUE'.

In our example of a record with "total_charges=45500", this arrangement of If-THEN logic results in a value of 'HIGH',

```
IF 10001 <= total_charges <= 50000 THEN charge_cat = 'MED';
IF 45000 <= total_charges THEN charge_cat = 'HIGH';
```

whereas this ordering results in a value of 'MED'.

```
IF 45000 <= total_charges THEN charge_cat = 'HIGH';
IF 10001 <= total_charges <= 50000 THEN charge_cat = 'MED';
```

The most important take-away regarding the DATA STEP is that the records in the source dataset are processed sequentially in the order that the PDV receives them from the source dataset and performs the processes designated in the DATA STEP in the order the executable commands are encountered. When processing associated with the last record has completed, the DATA STEP is finished. With a few simple examples like these, analysts can begin to experience the power that SAS brings to bear on some fairly daunting analytic problems, and we have really only scratched the surface of the DATA STEP.

For excellent, expanded explanations of the DATA STEP programming and the PDV, the reader is referred to Buck (2005), Whitlock (2007), Howard (2003), Aster & Seidman (1997), and Cody (2007a).

SAS FUNCTIONS

Once analysts have a feel for the DATA STEP, we introduce them to SAS Functions as a way to further expand the ways in which they can interact with and manipulate their data. For additional resources on the topic of SAS functions, the reader is referred to Cody (2007b, 2010), Carpenter (2005), Cassidy (2004), Harp (1998), Howard (1999), Morgan (2006), but for the purposes of an introduction to SAS functions, it should suffice to understand that a function is a program that takes zero, one, or more inputs and provides one distinct output as a result of those inputs. SAS functions are generally categorized by the types of data on which they act—character, numeric, date.

CHARACTER FUNCTIONS. Character (or string) functions are heavily used in healthcare analytics due to the nature of old billing and patient record systems that are still in use (which often use character fields to store numeric data or store data only in the form of transcribed clinical dictations). In order to use these data it is often necessary to use character functions to transform the data to another data type or parse the text field to derive a coded value embedded in the character string.

If you are lucky, this transformation is as simple as utilizing the **INPUT** function to transform individual data columns from character to numeric data types using the appropriate **INFORMAT**. "An informat is an instruction used to read data values into a SAS variable. In addition, if a variable has not yet been defined, SAS uses the informat to determine whether the variable ought to be numeric or character. The informat is also used to determine the length of character variables." (Droogen 2004, p. 1). In the following example, the "tot_pay" variable is stored as a text string (e.g., \$10,050.75) but we would like to convert it to a numeric variable (e.g., 10050.75) so that we can calculate "outstanding_balance" by subtracting it from the "total_charges" variable we created in the previous example. Using the **INPUT** function, we create the new variable "total_payments" as a numeric version of the "tot_pay" using the comma12. **INFORMAT**. This informat includes in its instruction set the information necessary to

understand that commas and dollar signs are to be ignored in determining the numeric value that is being read.

```
DATA work.encounters;
  SET work.encounters;
    total_charges = INPUT(tot_pay,COMMA12.);
RUN;
```

With a different instruction set, (say) "8." the dollar sign and commas are not understood by SAS and an error is generated to the log:

```
DATA work.inpatient_calculated_vars;
  SET hca.encounters;
    total_charges = INPUT(tot_pay,8.);
RUN;
```

```
NOTE: Invalid argument to function INPUT at line 966 column 17.
acct_num=810960 tot_pay=$10,050.75 total_payments=. _ERROR_=1 _N_=1
NOTE: Mathematical operations could not be performed at the following places. The
results of the operations have been set to missing values.
```

In order to make sure that you are using an informat that will work with your data, you might also want to use the **VERIFY** function to validate that the characters in the text string will be accommodated by the **INFORMAT** you are using. For example, in validating the "tot_pay" field, you should confirm that the only characters found in the text value are "0123456789\$.,". We know that the **COMMA12.** informat can accommodate those characters (when they are in the expected positions), so a validation that the "tot_pay" field for any given record contains **ONLY** those characters would be one way to help confirm that processing tot_pay with comma12. is going to resolve to a valid numeric value.

```
DATA work.inpatient_calculated_vars;
  SET hca.encounters;
    IF VERIFY(tot_pay,'0123456789$.,') = 0 THEN
      total_payments = INPUT(tot_pay,COMMA12.);
RUN;
```

The **VERIFY** function scans the field entered as the first parameter (tot_pay) for occurrences of characters other than those provided in the text string in the second parameter and returns the position of the first character in the first parameter that is not listed in the second parameter. For example, **VERIFY** ('DOG','DEF') would return "2" because "O" in DOG is not found in the verification list 'DEF'. Therefore, if the result of verifying "tot_pay" against our verification string results in 0, we know there are no characters in the current value of "tot_pay" that violate the characters that the **COMMA12.** informat knows how to handle. If the billing system did things like assign values to "tot_pay" such as "COLLECTIONS" to identify accounts that had gone to collections, we could find those with **VERIFY** and deal with them in a manner that fit our business rules (in this case by keeping them out of our "active_accounts" dataset).

```
DATA work.active_accounts;
  SET hca.encounters;
    IF VERIFY(tot_pay,'0123456789$.,') = 0 THEN
      total_payments = INPUT(tot_pay,COMMA12.)
    ELSE IF tot_pay = 'COLLECTIONS' THEN DELETE
RUN;
```

Working with data from older, complex claims and billing systems is rarely so simple that we can get away with simply converting text representations of numeric values. Another challenge that we often face is searching for a code embedded in a complex text string—for example to extract a clinic code from a "location" field. This is where the **SUBSTR** (or, substring) function comes in handy. Substring takes as its arguments (a) the character variable from which we want to extract a portion, (b) the position identifying the start of the desired string, and (c) the length of the string we wish to extract. For example, if a clinic code used to identify a clinic in a hospital's chart of accounts can be found in positions 2 – 6 of a field called "location", we might derive a new variable "clinic_code" from the location field using the **SUBSTR** function:


```

DATA work.encounters3;
  SET work.encounters2;
      clinic_code = SUBSTR(location,2,4);
RUN;

```

In this case a "location" value of "1SKNC-BLUE-09-132" would yield the clinic_code "SKNC" (for the Skin Center). Starting in the second character of the string, substrng will return the next four characters in the string.

Sometimes, however, character strings contain information that is somewhat more dynamic, and we cannot depend on the data we are looking for being in a fixed position. For example, if the clinic code was still embedded in "location", but all we know about its position within the string is that it is always preceded by "*", stating a fixed position and length for our substrng would do us no good:

```

"1BLUE*SKNC-09-132"      SUBSTR(location,7,4) ==> SKNC
"1YELLOW*SKNC-09-132"  SUBSTR(location,7,4) ==> W*SK

```

Instead, we need to use another function to determine the location of "*" and begin our substrng at the next position in the string. We do this using the **INDEX** function—which takes as its inputs, the string being searched and the term for which we are searching:

```

DATA work.encounters4;
  SET work.encounters3;
      clinic_code = SUBSTR(location,INDEX(location,'*')+1,4);
RUN;

```

Note here that the result of one text function [INDEX(location, '*')] is serving as an input to another function (the SUBSTR of location). For the location value of "1BLUE*SKNC-09-132" the INDEX function searching for "*" will resolve to the value "6". We then add one to that number and get the appropriate starting value for our clinic code—7. For the next location "1YELLOW*SKNC-09-132", the same INDEX function resolves to the value "8"; when we add one to that we get the appropriate starting value (9) for the SUBSTR function to use in deriving the clinic code from "location".

The important thing to remember from these examples is not just that there are a number of powerful SAS functions one can use to operate on character data, but that these functions can be combined in ways that allow you to create even more powerful solutions to otherwise difficult problems by using functions as inputs to other functions.

DATE FUNCTIONS. Another area in which healthcare analysts deal with difficult data conversions that can benefit from SAS functions is when dealing with dates. In order to understand date functions, however, it is important to understand how SAS stores date (and datetime) data. This is one of the data management areas in which new analysts struggle most, but once a few simple concepts are presented they quickly overcome this conceptual barrier.

First, SAS stores dates as integers representing the number of days since January 1, 1960. So even though your SAS dataset may look like it contains a value of 05/15/1963, what is really stored is the integer value "1230"—the number of days from January 1, 1960 to May 15, 1963. Likewise, a date value of "January 1, 1960" is stored as "0" and December 31, 1959 as "-1". So when doing date arithmetic, the difference between two dates results is the number of days because you are simply subtracting one integer from another. Similarly, datetime values (e.g., 15MAY2010:00:20:30) are stored as the number of seconds since January 1, 1960 at midnight.

As a result of these two storage methods, variables of data type "date" cannot be directly compared to variables that are stored as "datetime". For example, suppose you have an encounter record with a "discharge_date" (stored with data type "DATE") of 5/13/2008 and an admissions date that is stored as DATETIME and contains a value of 07MAY2010:09:30:25. If you wrote a data quality control program to make sure that there are no inpatient encounters for which the patient was discharged prior to their admit date, a comparison of these two values would be flagged as a potential error.

```

DATA work.potential_errors;
  SET hca.encounters;
  IF discharge_date < admit_date THEN OUTPUT;
RUN;

```

In this example, what we intend to do is generate a set of potential errors consisting of all records that have discharge dates prior to the admit date. But because the discharge date is stored as a date and the admit date is stored as

datetime, the comparison will reveal that bill date of 5/13/2008 (stored as 17,665--the number of days since January 1, 1960) is, less than 07MAY2008:09:30:25 (stored as 1,525,771,825—the number of seconds since Midnight on January 1, 1960), and the record will be output to the "potential_errors" dataset.

This is why the **DATEPART** function is so important for analysts dealing with data from a variety of source systems. DATEPART performs the conversion necessary to make datetime variables directly comparable to date variable. Using DATEPART to create our previous quality-control dataset, results in 05/13/2008 (17,665) being compared to 05/07/2008 (17,659) in order to correctly conclude that this discharge date was not prior to the admit date.

```
DATA work.potential_errors;
  SET hca.encounters;
  IF bill_date < DATEPART(admit_date) THEN OUTPUT;
RUN;
```

As another example of the use of date functions, suppose you are asked to help with a workforce scheduling project to determine the days of the week that might require more staffing in the emergency room. You could look at the cases coming into the ER by triage date, and use the function **WEEKDAY** to create a new variable that represents the day of the week. WEEKDAY takes as its input a date and returns the day of the week on which that date fell. For example, the variable "dow" in the following example would come to take the value of "1" for every charge record with a "procedure_date" of May 30, 2010 because that was a Sunday, the 1st day of the week.

```
DATA work.encounters;
  SET hca.encounters;
      dow = WEEKDAY(procedure_date)
RUN;
```

There are also date functions that perform much more sophisticated computations like determining the number of intervals of a certain type that have passed between two dates. **INTCK** is a function that takes two dates, times, or datetimes as its date parameters and then takes the name of an interval type (e.g., biweekly) to determine the number of that interval type that has begun in the time between two dates. In the following example, the new variable "num_billing_cycles" would resolve to 6—indicating the number of 4-week billing cycles (beginning on Monday) that have begun between a billing date of 5/13/2008 and a payment date of 11/14/2009:

```
DATA work.collections;
  SET billing.charges
      num_billing_cycles=INTCK('WEEK4.1',vara,varb);
RUN;
```

NUMERIC FUNCTIONS. Oddly, there is not nearly as much need to use numeric functions in healthcare analytics as there is for character and date functions. Mainly this is a result of dollar figures usually being stored, for the most part, as numeric values and numeric values being dealt with largely with normal mathematical functions (addition, multiplication, etc.). However, for purposes of tying out to other financial reports, one may find the need to round numbers to the same level of precision as is found in other systems. This can be done using the **ROUND** function, which takes as its inputs the value to be rounded and an indicator of the level of precision that is required. For example, to round the value resulting from the multiplication of total charges for professional services times a capitation proportion, one could use the following to arrive at a value that is rounded to the penny (i.e., .01):

```
DATA work.professional_billing;
  SET billing.charges
      total_capitation_charges = ROUND(total_charges*capitation,.01);
RUN;
```

Similarly, we have found occasion to use **CEIL** and **FLOOR** which provide an easy way to determine the next highest (or lowest) integer from the input of a calculated numeric value. For example if the age of a health plan member on the date of service is calculated using the date of service and member's date of birth, we might use FLOOR to produce the age at time of service for reports and analytic groupings.

```
DATA work.professional_billing;
  SET billing.charges
      age_on_date_of_svc = FLOOR(svc_date-dob/364.25);
RUN;
```

With an understanding of the DATA STEP and SAS functions you can gain significant control over creation of your analytic datasets—creating new calculated fields, keeping records that meet certain criteria and deleting those that do not. By this point, however, most analysts are eager to start using the analytic power of SAS, so we follow the discussion the DATA STEP and FUNCTIONS with an introduction to SAS Procures (or, PROCs).

SAS PROCEDURES

"SAS procedures analyze data in SAS data sets to produce statistics, tables, reports, charts, and plots, to create SQL queries, and to perform other analyses and operations on your data. They also provide ways to manage and print SAS files" (SAS, 2010b). Put simply, PROCs are specialized program units that provide a wide array of functionality for manipulating and analyzing your data. Like functions, each PROC has its own syntax and is built to provide a specific type of functionality, but unlike functions, PROCs operate on entire datasets and can produce widely varied output depending on how their execution is specified. Once you learn what a specific PROC can do and its basic syntax, you can continue to expand its usefulness through utilization of a rich set of syntax extensions and options. Some of the PROCs that we use most heavily are described below.

PROC FREQ. Among the most heavily used PROCs is PROC FREQ, which is used to produce frequency distributions and perform inferential statistical tests on categorical data.. Like many other PROCs, PROC FREQ begins by specifying the dataset on which the proc will be performed (DATA=). This is followed by a TABLE statement that specifies the dataset variables (or combination of variables) that will be analyzed. In the following example, we want to look at the distribution of our health plan members across the plans to which they belong.

```
TITLE 'Membership Distribution';
PROC FREQ DATA=work.members;
  TABLE plan_code account_type;
RUN;
```

Membership Distribution
The FREQ Procedure

plan_code	Frequency	Percent	Cumulative Frequency	Cumulative Percent
21	42311	18.40	42311	18.40
37	54081	23.51	96392	41.91
41	45675	19.86	142067	61.77
70	39048	16.98	181115	78.75
90	48885	21.25	230000	100.00

account_ type	Frequency	Percent	Cumulative Frequency	Cumulative Percent
FMFD	24266	10.55	24266	10.55
FMSD	181478	78.90	205744	89.45
SMSD	24256	10.55	230000	100.00

The frequency procedure produces a table in the output window for each variable (or combination of variables) indicated in the TABLE statement. In the preceding example, we produced separate have a frequency distributions of the 230,000 lives covered by our health plans distributed across "plan_code" and "account type".

If we want to look at these same data as a crosstab of "plan_code" and "account_type", we would run the following:

```
TITLE 'Membership Distribution by Plan Code and Account Type';
PROC FREQ DATA=work.members;
  TABLE plan_code*account_type;
RUN;
```

Membership Distribution by Plan Code and Account Type
The FREQ Procedure

Table of plan_code by account_type

plan_code	account_type			Total
Frequency	FMFD	FMSD	SMSD	
Percent				
Row Pct				
Col Pct				
21	4504	33385	4422	42311
	1.96	14.52	1.92	18.40
	10.64	78.90	10.45	
	18.56	18.40	18.23	
37	5645	42769	5667	54081
	2.45	18.60	2.46	23.51
	10.44	79.08	10.48	
	23.26	23.57	23.36	
<Remaining "plan_code" rows>				
Total	24266	181478	24256	230000
	10.55	78.90	10.55	100.00

In addition to this tabular output, PROC FREQ is used to produce inferential statistics (e.g., Chi-Square tests) and can be used to produce a dataset (e.g., "plan_by_acctyp" in the following code) that can be used as source data for other steps in your program:

```
PROC FREQ DATA=work.members;
  TABLE plan_code*account_type / chisq out = plan_by_acctyp;
RUN;
```

In addition to the crosstab produced in the previous example, this code produces the following Chi-Square test, as well as the dataset "plan_by_acctyp".

Membership Distribution by Plan Code and Account Type
The FREQ Procedure
Statistics for Table of plan_code by account_type

Statistic	DF	Value	Prob
Chi-Square	8	3.1840	0.9223
Likelihood Ratio Chi-Square	8	3.1823	0.9224
Mantel-Haenszel Chi-Square	1	0.1352	0.7131
Phi Coefficient		0.0037	
Contingency Coefficient		0.0037	
Cramer's V		0.0026	

Sample Size = 230000

	plan_code	account_type	Frequency Count	Percent of Total Frequency
1	21	FMFD	4504	1.952608696
2	21	FMSD	33385	14.515217391
3	21	SMSD	4422	1.9226086957
4	37	FMFD	5645	2.4543478261
5	37	FMSD	42769	18.595217391
6	37	SMSD	5667	2.463913043
7	41	FMFD	4821	2.0960869565
8	41	FMSD	35973	15.640434783
9	41	SMSD	4881	2.122173911
10	70	FMFD	4150	1.8043478261
11	70	FMSD	30790	13.386956522
12	70	SMSD	4108	1.7860869565
13	90	FMFD	5146	2.2373913043
14	90	FMSD	38561	16.76562174
15	90	SMSD	5178	2.2513043478

PROC MEANS / PROC UNIVARIATE. PROC MEANS can be used to provide the Mean, Standard Deviation, Min./Max. values for numeric variables in a dataset. In addition to being useful for standard reporting of descriptive statistics, however, it is also valuable as a data cleaning tool—as it can reveal out-of-range values (e.g., identifying negative values for "Patient Age", revealing variables like "Cost", and "Billed Amount" that may have too little or too much variance based on historical data, etc.). In the following example, we produce descriptive statistics related to the "age" of a specific patient cohort.

```
TITLE 'Mean age of patients in cohort XYZ';
PROC MEANS DATA=patients2;
  VAR age;
RUN;
```

Mean age of patients in cohort XYZ
The MEANS Procedure
Analysis Variable : age

N	Mean	Std Dev	Minimum	Maximum
18883	59.1338481	15.1844640	3.9560741	85.8970487

Whereas PROC MEANS provides the descriptive statistics relevant to understanding the mean value of numeric variables, PROC UNIVARIATE provides a more comprehensive set of descriptive statistics. In the following example, we run PROC UNIVARIATE on the same dataset and variable. Notably, in addition to the Mean and Standard Deviation, PROC UNIVARIATE provides the Median and Mode in addition to the Quartile break (which show us for this dataset that our population breaks into patients 0 – 49, 50 – 60, 61 – 70, and 70 & above in roughly equal numbers).

```
PROC UNIVARIATE DATA=patients2;
  VAR age;
RUN;
```

The UNIVARIATE Procedure
Variable: age

Moments			
N	18883	Sum Weights	18883
Mean	59.1338481	Sum Observations	1116624.45
Std Deviation	15.184464	Variance	230.567946
Skewness	-0.4991838	Kurtosis	-0.1120803
Uncorrected SS	70383884.7	Corrected SS	4353583.96
Coeff Variation	25.6781259	Std Error Mean	0.11050043

Basic Statistical Measures

Location		Variability	
Mean	59.13385	Std Deviation	15.18446
Median	60.43102	Variance	230.56795
Mode	63.17090	Range	81.94097
		Interquartile Range	20.81263

Quantile	Estimate
100% Max	85.89705
99%	84.91146
95%	81.57310
90%	78.47083
75% Q3	70.57241
50% Median	60.43102
25% Q1	49.75978
10%	38.13315
5%	31.29719
1%	20.00000
0% Min	3.95607

PROC FORMAT. In our opinion, PROC FORMAT is one of the most important PROCs utilized in healthcare analytics. Just as INFORMATS, discussed previously, are used by SAS to know how to read-in data, FORMATS are used as instructions to tell SAS how to display data. As discussed previously, dates are stored as integers such that "17659" represents the date May 7, 2008. By applying the SAS format MMDDYY10. to the variable that contains this date, we transform the dataset from being a jumble of nearly uninterpretable integers to being data columns with an easily recognized format. We can set the format of a variable in the dataset as follows:

```
DATA work.claim_detail;
  SET billing.claim_detail;
```

```
FORMAT bill_date MMDDYY10.;
RUN;
```

If the "bill_date" in this dataset exists as an integer, its representation when browsing the dataset or being output by an analytic PROC will be in the familiar form "MM/DD/YYYY". If the variable does not yet exist in the dataset, it will be created as a numeric variable that will be displayed in that same MM/DD/YYYY format. What is important to remember is that by applying a format, YOU HAVE NOT CHANGED HOW THE DATA ARE STORED; you have simply instructed SAS how to DISPLAY the data that the formatted column displays.

Before applying the MMDDYY10. format, the bill_date appears as an integer:

VIEWTABLE: Work.Billing_tem		
	acct_num	bill_date
1	810960	17850
2	810958	17851
3	810973	17852

But after applying the MMDDYY10. format to "bill_date", it is displayed with the assigned format.

VIEWTABLE: Work.Billing_te		
	acct_num	bill_date
1	810960	11/14/2008
2	810958	11/15/2008
3	810973	11/16/2008

Again, please note that what is stored in the dataset is still the integer value. We have simply instructed SAS (by applying the format) to display the bill_date variable in a more familiar manner. All you need to do is find the SAS FORMAT that displays the data the way you want it displayed (e.g., DATE9.==> 14NOV2008, MMDDYY8. ==> 11/14/08, etc.).

That is great for numeric values that have a system-defined SAS FORMAT, but what if you want to display the labels that correspond to your own codes. For example, you want to display "race/ethnicity" not as "0", "1", etc., but as "White", "Black". Or more importantly, you want to display the ICD9 code description (or short description) instead of just the code. Instead of "536.3", you want to display the description "Gastroparesis". This is where PROC FORMAT comes in; it allows you to create your own (user-defined) formats based on the code-label pairs that are important to you. We use two methods for creating these formats: hard-coding the values in a proc format statement and using datasets to automatically generate them.

As an example of the former method, suppose we want to create a format called "race" to format character (string) codes into a text description of that code and a format called "gender" to do the same thing for formatting numeric code values for gender. We could accomplish this with the following:

```
PROC FORMAT LIBRARY = formats;
VALUE $ethnic
'A' = 'Asian'
'B' = 'Black'
'H' = 'Hispanic'
'I' = 'Native American'
'W' = 'White/Caucasion'
other = 'Incorrect Code';
VALUE gender
0 = 'Male'
1 = 'Female';
RUN;
```

First we issue the PROC FORMAT command and define the library in which we want to create the format—in this example we have a library called "formats" where we want to store our formats. We identify the name of the format with the "VALUE" statement followed by the value-label pairs that will define how SAS is to display each of the values encountered. Applying the format "\$ethnic" to a column of character string data will result in 'A' being displayed as "Asian", for example. Likewise, applying the format "gender" will display "0" as "Male" and "1" as "Female".

Applying our user-defined formats to the dataset "billing_temp", we see the dataset now displays the data in the manner dictated by our user-defined format.

```
DATA billing_temp;
SET billing_temp;
FORMAT gender gender. ethnicity $ethnic.;
RUN;
```

The dataset on the left represents the data as it looks without the formats applied; the one on the right, with the formats applied. One very important point about the last record in this dataset should be pointed out; the value "X" is not included in the '\$ethnic' format, but there is an "other" entry in the VALUES statement that has the effect of

making any values not identified in the format display as (in this case) "Incorrect Code". By applying this format to your datasets and then performing a PROC FREQ on the formatted variable can quickly resolve errors in either your FORMAT or (sometimes) source data system.

VIEWTABLE: Work.Billing_temp			
	acct_num	gender	ethnicity
1	810960	0 W	
2	810958	1 A	
3	810973	1 W	
4	811053	0 X	

VIEWTABLE: Work.Billing_temp			
	acct_num	gender	ethnicity
1	810960	Male	White/Caucasion
2	810958	Female	Asian
3	810973	Female	White/Caucasion
4	811053	Male	Incorrect Code

Creating user-defined formats can be a very powerful way to add useful information to your dataset, but what if you wanted to replace CPT/HCPCS or ICD9 codes with their descriptors (or a shortname)? You certainly do not want to type all of that information in a VALUES statement, and you probably have it stored in a database somewhere already. In these situations, we use PROC FORMAT in a slightly different way to create our user-defined formats (see Cody, 2010; Scerbo, 2007 for further discussion of this and alternative approaches).

In the following example, we use a DATA STEP to pull the diagnosis codes and their descriptions from the billing system's "diagnosis" table. Our goal is to create a "control file" that can be passed to PROC FORMAT in a manner that will create a format similar to the ones that we coded by hand in the previous example. The form of the control file that PROC FORMAT will accept is very specific. For the type of format we are creating (which involves simple value-label combinations) we must have a column named "start" that will hold the "value" and a column called "label" that will hold (you guessed it) the "label". We assign those two variables the values of "diagcd" and "diagnosis_description" from our system's "diagnosis" table. We also create a column named "fmtname" which contains the name of the format we are creating.

```
DATA work.diagnosis_format;
  SET billing.diagnosis END=last;
  start=diagcd;
  label=diagnosis_description;
  fmtname='$DIAGNOSIS'; output;
  IF last THEN DO;
    hlo='o';
    start='';
    label='***Diagnosis Not Assigned***';
    OUTPUT;
  END;
KEEP start label fmtname hlo;
RUN;
```

In order to insert our "other" format value ('Incorrect Value' in the previous example) we need to determine where the end of the source file is and then insert one more record into the control dataset. This is accomplished by defining the end of the dataset with the identifier "last" and then (with each execution of the DATA STEP) evaluating "last" to see if SAS has yet reached the end of the dataset. When the last record is read from "billing.diagnosis" SAS will write that record to the target dataset (note the explicit "OUTPUT" statement prior to the "IF" statement) and create one additional record to write to the source dataset. This record will create a new column in the dataset "hlo" (which stands for High, Low, Other), assign a null value to "start" and assign the "label" a value that will let us know that a value not contained in our format has been encountered in the column to which the resulting format is applied. In other words, when the format is applied, values in the formatted column will be presented as "***Diagnosis Not Assigned***" for records that do not have a diagnosis code that is in our "diagnosis_format" dataset.

Once we have the control dataset created, creating the format is simply a matter of executing PROC FORMAT with the optional "CNTLIN" (or, control in) dataset identified along with the library in which we wish to write the format.

```
PROC FORMAT CNTLIN=work.diagnosis_format LIBRARY = formats;
RUN;
```

With the format created, we can now apply it to our ICD9 data and write descriptions instead of codes in the results of

our analytic PROCs.

VIEWTABLE: Work.Billing_temp		
	acct_num	ICD9
1	810960	296.2
2	810958	715.3
3	810973	396.3
4	811053	61233

VIEWTABLE: Work.Billing_temp		
	acct_num	ICD9
1	810960	Major depressive disorder, single episode; unspecified
2	810958	Osteoarthritis, localized, not specified; site unspecified
3	810973	Mitral valve insufficiency and aortic valve insufficiency, Mitral and aortic (valve): incompetence, regurgitation

PROC TRANSPOSE Just as PROC FORMAT can change the presentation of your data in dramatic ways, so can PROC TRANSPOSE change the "shape" of your data. Specifically, there are occasions on which, for reporting or analytic purposes, you may need to rearrange your data—making rows of data into columns and columns into rows. That is when you should consider using PROC TRANSPOSE. In the following example, we need to take data summarized for each plan year per client and transpose the data for loading into a third-party reporting system. The data in each column of the "inpatient_admits" dataset need to be represented in a row of data representing that column's values in the 2008 and 2009 fiscal years.

```
PROC TRANSPOSE DATA=inpatient_admits OUT=inpatient_transposed;
  BY client_id ;
  VAR admits admits_per1000 admit_days admit_netpay_pmpm ;
  ID year;
  IDLABEL year;
RUN;
```

In order to do this, we identify to PROC TRANSPOSE both the original dataset "inpatient_admits" and the target dataset that will contain the transposed data "inpatient_transposed". The "BY" statement is key to transposing the data. You can think of the BY variables as the pivot-point on which the data will be transposed. For each occurrence of client_id with the same value, a column will be created to contain the value associated with each of the transformed variables for that client_id. As there are two rows for each "client_id" in our example dataset, two columns will be created in the transposed dataset to hold the values for each transformed variable. If there had been a client for which we had included three years worth of data, the output dataset would have three columns of transformed data, but the third column would be null for all clients except the one with three years of data.

Next, the VAR statement is used to identify the variables that are to be transformed from the original dataset. These variable names will now be stored in a column called (by default) "_NAME_". The "ID" and "IDLABEL" statements identify the columns that are to be utilized to identify the row origin of the new transposed columns. In this example, "year" was used as both the name of the columns (ID) and the label for those columns (IDLABEL), but you could imagine if you had another column in the original dataset that you might want to use as a label instead that you would use "year" for the ID and that other column (e.g., Fiscal_Year_Label with values of 'FY2008' and 'FY2009') as the label.

VIEWTABLE: Work.Inpatient_admits						
	client_id	YEAR	admits	admits_per1000	admit_days	admit_netpay_pmpm
1	AB	2009	67	39	147	115.2
2	AB	2008	73	42	154	114.32
3	FB	2009	78	24	94	85.58
4	FB	2008	66	20	79	85.43
5	GH	2009	81	36	114	93.72
6	GH	2008	83	37	116	93.21

VIEWTABLE: Work.Inpatient_transposed				
	client_id	NAME OF FORMER VARIABLE	2009	2008
1	AB	admits	67	73
2	AB	admits_per1000	39	42
3	AB	admit_days	147	154
4	AB	admit_netpay_pmpm	115	114
5	FB	admits	78	66
6	FB	admits_per1000	24	20
7	FB	admit_days	94	79
8	FB	admit_netpay_pmpm	85	85

One question that is almost always asked when discussing PROC TRANSPOSE with new users is "can you transpose the dataset back?". The answer is yes; likewise, if you get a dataset that is in a form similar to "inpatient_transposed" and you want to turn it into a dataset similar to "inpatient_admits", just do a transpose similar to the following:


```

PROC TRANSPOSE DATA=inpatient_transposed OUT=inpatient_admits2;
  BY client_id ;
  VAR _2008 _2009;
  ID _name_;
RUN;

```

Again, we are simply creating a row for every column in the VAR statement and creating a column for each unique value of ID (in this case _NAME_).

Finally, it should be noted that PROC TRANSPOSE is not a PROC that is used in every data analysis project, but when this particular functionality is needed there are few (if any) better options. Another data management tool that is not always needed (but is indispensable when it is needed) is PROC COMPARE.

PROC COMPARE. With all of the changes that occur to datasets throughout an analytic project, every analyst, at some point, finds him/herself with datasets that they swear "should be identical now". Whether it is the mapping of a new pharmacy claims source file that is being done on the fly to meet a deadline or validation of a carefully planned data migration, there will come times when you ask yourself "are these datasets the same?" When that time comes, turn to PROC COMPARE for help. In its simplest form, PROC COMPARE needs only have the two datasets being compared designated as the BASE and COMPARE datasets.

```

PROC COMPARE BASE = work.encounters_a COMPARE = work.encounters_b;
RUN;

```

When executed, PROC COMPARE compares the values in every column across every row of these two datasets to confirm that they are identical. It produces wonderful output that summarizes both datasets at a high level (e.g., number of rows, number of columns, etc.), but in cases where differences are found, it also identifies the record where differences were first encountered and provides examples of those differences. Because this comparison proceeds row by row, however, you will need to have the datasets sorted in the same order. In addition to comparing final production datasets to one another, PROC COMPARE is a great tool for validating transformations to your data; with a copy of the original (untransformed) dataset, one can write the code that should "undo" the transformations and arrive back at an identical dataset from which they started.

PROC SORT, BY STATEMENTS, & RETAIN. The descriptions of the previous two PROCs have hinted at this, but it deserves to be explicitly stated; several PROCs rely on the order of records in a dataset for their correct operation. The reason this fact gets overlooked so often is that SAS does a good job of telling you when this simple requirement is violated, and the solution is usually straight-forward. In the previous PROC TRANSPOSE example, if the records had not been sorted by the "client_id", an error similar to the following would have been written to the LOG:

```

ERROR: Data set WORK.INPATIENT_ADMITS is not sorted in ascending sequence. The
current by-group has client_id = GH and the next by-group has client_id = FB.

```

So naturally, based on your interpretation of this error you think "hey, I need to sort this dataset by the client_id in order for my PROC TRANSPOSE to work." You find an example of PROC SORT and determine that the following code will solve your problem:

```

PROC SORT DATA=inpatient_admits;
  BY client_id;
RUN;

```

Similarly, in PROC MEANS, PROC FREQ, and many other PROCs, utilization of a BY statement is an important technique for efficiently manipulating and analyzing your data on a subset by subset basis, and PROC SORT is the PROC that you can use to assure your dataset is sorted correctly before using a BY statement.

Used in conjunction with the DATA STEP, a BY statement exposes some additional automatic (*magic*) variables that can help you do some fairly sophisticated things with your data. In the following example, we sort a medical claims summary dataset (which contains total net paid claims for each plan member) by plan year, age/gender cohort, and (in descending order) total net payments per member.

```
PROC SORT DATA=member_claim_totals;
  BY planyr cohort DESCENDING member_total;
RUN;
```

This sorted dataset can then be used in a DATA STEP in which the source data are processed "BY" these same three variables. Processing the dataset using these BY variables gives the DATA STEP access to system variables that are not otherwise accessible—'FIRST.' and 'LAST.'. These two system variables exist for each of the variables in the BY statement, so in the following DATA STEP there exist both a FIRST.planyear/LAST.planyear and a FIRST.cohort/LAST.cohort for each record in the dataset. These FIRST. and LAST. variables are Boolean indicators of whether the current record being processed is the FIRST record in the group of records with that record's value for that variable (e.g., the first record for planyear 2002). [NOTE: The automatic variables FIRST.MEMBER_TOTAL and LAST.MEMBER_TOTAL also exist during this DATA STEP, but because there is only one record per member per plan year (and only one total for each member in each plan year), both of these variables will have the value of "TRUE" for each record in the dataset.]

MEMID	PLANYR	COHORT	FIRST.PLANYR	LAST.PLANYR	FIRST.CHORT	LAST.COHORT	MEMBER_TOTAL
111001	2002	18FEMALE	TRUE	FALSE	TRUE	FALSE	\$58,250
111002	2002	18FEMALE	FALSE	FALSE	FALSE	TRUE	\$34,600
222001	2002	18MALE	FALSE	FALSE	TRUE	FALSE	\$27,000
222002	2002	18MALE	FALSE	TRUE	FALSE	TRUE	\$26,500
333001	2003	18FEMALE	TRUE	FALSE	TRUE	FALSE	\$54,200
333001	2003	18FEMALE	FALSE	TRUE	FALSE	TRUE	\$44,800

As a practical example of how one might use these automatic variables, suppose we wanted to create a dataset of the highest cost claimants for each of the cohorts for each plan year.

```
DATA high_cost_claimants;
  SET member_claim_totals;
  BY planyr cohort DESCENDING member_total;
  IF FIRST.cohort then OUTPUT;
RUN;
```

MEMID	PLANYR	COHORT	MEMBER_TOTAL
111001	2002	18FEMALE	\$58,250
222001	2002	18MALE	\$27,000
333001	2003	18FEMALE	\$54,200

With the data sorted by planyr, cohort, and member_total (descending), identifying the "first.cohort = TRUE" records results in the dataset 'high_cost_claimants'. However, for this type of analysis to be meaningful, you probably want to look at the top 10 or 20 highest cost claimants to determine if there are meaningful trends associated with a given diagnosis or procedure for a given plan. Unlike determining the first or last record in a sorted BY group, however, there is no system variable that will provide you with a rank order within a sorted group. You will have to create that rank-order yourself. One way to do so is by using a RETAIN statement.

One of the basic tenants of the DATA STEP is that rows from the source dataset are processed serially, one after the other. One logical extension of that behavior that would seem reasonable would be that information from one row cannot be available for SAS to use in processing subsequent records. Even with "FIRST." and "LAST.", all SAS knows is that the current record either IS or IS NOT the first or last record of the sorted "by-group". It is not aware of whether it is (say) the 19th record of the sorted group or the 5th—knowledge that would require information being transferred from one record to the next. In order to achieve this feat, SAS uses the RETAIN statement which allows the program to "retain" information from one record to the next.

Expanding on the previous example, we now want to identify the top 20 highest cost claimants for each cohort and year. In order to create this ranking, we start with the knowledge that the record with "FIRST.planyr=TRUE" and "FIRST.cohort=TRUE" represents the highest cost claimant (ranking=1) for that plan year and cohort. For each subsequent record within that plan year and cohort, we need to know what the ranking of the previous record was so that we can assign the appropriate ranking for the current record. We will do that by retaining the rank, which will make the rank available to SAS during the processing of the subsequent record.

```
DATA high_cost_claimants;
  SET member_claim_totals;
  BY planyr cohort DESCENDING member_total;
  RETAIN rank;
  IF FIRST.planyr and FIRST.cohort then rank = 1;
  ELSE rank = rank + 1;
```

```
IF rank <= 20 then OUTPUT;
RUN;
```

In this example, we start by assigning the rank of "1" to the first record for each planyr and cohort. We know this is the record with the highest costs because of how we previously sorted the dataset. Further, because the variable "rank" is being retained, SAS will be able to use that information when processing the next record in that sorted group. It will assign the rank of that record to be the previous rank plus one, and will repeat this process until the next planyr/cohort group is encountered. At that point, FIRST.planyr and FIRST.cohort will both be "TRUE" again, the rank of that first record will be assigned the value "1" and the process will repeat for this group defined by the current plan year and cohort. After the rank is assigned, it is later evaluated to determine if the current record represents one of the top 20 highest cost claimants for that plan year and cohort. If it does, the record will be output to the "high_cost_claimants" dataset.

In reality, one would probably want to put some additional logic in place to deal with "ties" in the rankings—perhaps by also retaining a variable that contains the value of the member_total from the previous record and only increasing the rank if that value is different from the current member_total.

```
DATA high_cost_claimants;
SET member_claim_totals;
  BY planyr cohort DESCENDING member_total;
  RETAIN rank previous_total 0;
  IF FIRST.planyr and FIRST.cohort THEN rank = 1;
    ELSE if previous_total NE member_total THEN rank = rank + 1;
previous_total = member_total;
IF rank <= 20 THEN OUTPUT
RUN;
```

Here, in addition to retaining the rank, we are also retaining the total claims paid for the member associated with the previous member record (previous_total). Note that we are assigning that value at the bottom of the DATA STEP. The retained value from the "current" record will then be used in the subsequent record ("if previous_total ne member_total then rank = rank + 1) to determine whether the ranking should be increased. The resulting high_cost_claimants dataset in this example may well contain more than 20 records for each client, but produces a more valid representation of the "top 20" costs because it deals with situations where more than one member has exactly the same value of member_total. This RETAIN technique can also be used to compute multiple "running totals" in a sorted dataset, but we generally prefer to create summarized (or, aggregate) datasets using PROC SQL.

PROC SQL. The structured query language (SQL) is a powerful tool for managing data. Many individuals coming into the role of healthcare analyst have significant experience with the structured query language (SQL) and therefore, introducing them to SAS by focusing on its SQL capabilities is often a good way to engage them in SAS programming and analysis. For the experienced DATA STEP programmer, SQL offers some very powerful methods for doing some of the same work as the DATA STEP, but can also be used to quickly and efficiently merge datasets in a manner that is simple and easy to understand.

PROC SQL serves as the SQL engine for the SAS system. Like other PROCs, it has its own specific syntax, but within this PROC, any ANSI-standard SQL statement can be executed. Selects, inserts, updates, and deletes can be performed as in any other ANSI-standard SQL engine—a fact that usually helps new analysts versed in SQL make the transition to SAS.

As an introductory example to creating a new dataset with PROC SQL, we will use it to create one of our earlier datasets using both the DATA STEP and PROC SQL:

```
DATA work.inpatient_encounters;
SET hca.encounters;
  total_charges = professional_charges + facility_charges;
  IF encounter_type = 'IP' THEN OUTPUT;
RUN;
```

Using PROC SQL, the same dataset is created as follows:

```

PROC SQL;
  CREATE TABLE work.inpatient_encounters AS
  SELECT *,professional_charges + facility_charges AS total_charges
  FROM hca.encounters
  WHERE encounter_type='IP';
QUIT;

```

This example demonstrates many of the basic components of a PROC SQL query. The SELECT statement identifies the variables that will go into creating the new dataset. The FROM clause defines the source dataset(s) from which the data are to be extracted and the WHERE clause specifies conditions that will be used to filter the data being returned from the source dataset(s). Although PROC SQL is frequently used to extract data from a single dataset, probably the most valuable functionality PROC SQL provides is the ability to extract and join datasets.

There are a variety of types of joins that can be performed. The simplest form of the PROC SQL JOIN is that in which one joins only those rows in each dataset that have a matching row (or rows) in the other dataset. For example, with respect to the "inpatient" dataset created earlier, if we wanted to create a dataset that contained this same charge-level detail but we also wanted to provide some additional identifying information and patient demographics (e.g., medical_record_number, gender, date of birth, etc.) to the resulting dataset, we might join the table containing patient charges to the table containing patient demographics:

```

PROC SQL;
  CREATE TABLE expanded_inpatients AS
  SELECT a.*,b.medical_record_number, b.gender, b.date_of_birth
  FROM work.inpatient_encounters a, hca.patients b
  WHERE a.patient_id = b.patient_id;
QUIT;

```

In this example, the charges and patients tables are joined via an "equi-join". In an equi-join only those rows from each table that have a matching row in the other table will contribute information to the resulting dataset. In other words if there is a patient in the patients table that does not have any charges in the charges table, that patient's medical record number, gender, and date_of_birth will not appear in the dataset "expanded_inpatients". Likewise, if there are billing charge items for a patient whose patient_id does not appear in the patients table, those charges will not appear in the resulting dataset. In order to use the equi-join correctly, we are assuming that all patients in the charges table have a matching record in the patients table. To the extent that this assumption is violated (e.g., there can be patients in the charges table that are not in the patients table), the validity of this query in showing "all charges" will be affected.

If, on the other hand, we wanted to show the charges for the month for all patients in the patients table, we could perform a "Left-Join" from the patients table to the billing table. In a left join all records in one dataset are in the result set regardless of whether that record has a matching record in the other dataset.

```

PROC SQL;
  CREATE TABLE expanded_inpatients AS
  SELECT a.medical_record_number, a.gender, a.date_of_birth, b.member_total
  FROM billing.patients a LEFT JOIN billing.charges b
  ON a.patient_id = b.patient_id;
QUIT;

```

Specifying the LEFT JOIN as the type of join being performed forces SQL to include the selected patient variables for all records in the patients table regardless of whether there are charge records for that patient in the charges table. Notice also that instead of the condition joining the tables being in a WHERE clause, the dataset on the left side of the join is joined to the other dataset ON the condition that is used to match records in the other dataset.

Conversely, sometimes we want to make sure that we include all records from two datasets—regardless of whether records in one have a match in the other. For example, if we have two patient datasets that contain complementary data (say claims data on screening procedures and health fair questionnaires) we might want to start our analysis by bringing these two data sets together and assess where we have missing data, how it can be filled in from other sources, and whether there is enough clean data to provide a meaningful analysis. One way to do this would be to do a FULL OUTER JOIN.

```

PROC SQL;
CREATE TABLE expanded_inpatients AS
SELECT a.medical_record_number, a.gender, a.date_of_birth, b.member_total
FROM billing.patients a FULL JOIN billing.charges b
ON a.patient_id = b.patient_id;
QUIT;

```

In terms of frequently used joins, there is one last join, the UNION JOIN, that is very helpful when creating analytic datasets. The union join takes records from two datasets (usually with the same variables) and "stacks" them on top of one another:

VIEWTABLE: Work.Inpatient_admits						
	client_id	YEAR	admits	admits_per1000	admit_days	admit_netpay_pmpm
1	AB	2009	67	39	147	115.2
2	AB	2008	73	42	154	114.32
3	FB	2009	78	24	94	85.58
4	FB	2008	66	20	79	85.43
5	GH	2009	81	36	114	93.72
6	GH	2008	83	37	116	93.21

VIEWTABLE: Work.Inpatient_admits3						
	client_id	YEAR	admits	admits_per1000	admit_days	admit_netpay_pmpm
1	KJ	2009	72	41	138	117.84
2	KJ	2008	66	20	137	121.67
3	MU	2009	77	37	84	92.85
4	MU	2008	58	52	83	85.72
5	ST	2008	91	26	115	92.67
6	ST	2009	67	48	115	94.13

If, for example, you were working on producing the analytic dataset for clients AB, FB, and GH and a colleague was working on producing the same dataset for clients KJ, MU, and ST you could produce the final dataset with all variables for all clients using a union join:

```

PROC SQL;
CREATE TABLE inpatient_admits_final AS
SELECT *
FROM WORK.INPATIENT_ADMITS
UNION
SELECT *
FROM WORK.INPATIENT_ADMITS3;
QUIT;

```

This will produce a dataset that is 12 rows long containing the data from the two individual datasets. Although the union join can be very helpful, it can also be somewhat treacherous. One issue to be aware of is the difference between UNION and UNION ALL. Using UNION in your SQL statement will automatically de-duplicate records in your dataset. This is usually a good thing, but if you are not aware of this difference you can spend quite a bit of time trying to figure out why two 25,000 record datasets create a 49,980 record dataset instead of the 50,000 record dataset you thought they should create. If you do need to retain duplicate records for some reason use UNION ALL instead of UNION.

Also, the UNION join does not check to make sure the columns being "stacked" (joined) have the same name, definition, or meaning; it takes them in the order they appear in the dataset. So if "admits" is the second variable in the dataset "inpatient_admits", and "admits_per1000" is the second variable in "inpatient_admits3", the second variable in "inpatient_admits_final" will be called "admits", but it will contain the "admits" from "inpatient_admits" and the "admits_per1000" from "inpatient_admits3". In this case, you must select the variables explicitly so that they are in the same order. The point here is to make sure you really understand the structure of your data and the functionality and limitations of the union join:

```

PROC SQL;
CREATE TABLE inpatient_admits_final AS
SELECT client_id, year, admits, admits_per1000, admit_days, admit_netpay_pmpm
FROM WORK.INPATIENT_ADMITS
UNION
SELECT client_id, year, admits, admits_per1000, admit_days, admit_netpay_pmpm
FROM WORK.INPATIENT_ADMITS3;

```

In addition to the different types of joins that can be performed, you can use PROC SQL to perform myriad types of summarizations, subsetting, and transformation operations. You can even utilize SAS Functions within your SQL queries:

```

PROC SQL;
  SELECT PUT(client_id,$CLIENT_NAME.), SUM(inpt_svc_costs_pmpm) as AS
         sum_ip_costs_pmpm
  FROM work.inpatient_services
GROUP BY PUT(client_id,$CLIENT_NAME.);
QUIT;

```

In this example, we use PUT to convert "client_id" to the client company's name using a user-defined format and sum the per member per month (pmpm) costs of inpatient services for each company.

As with other PROCs, there are a dizzying number of variations on how one can use PROC SQL to create and update datasets, and the present overview cannot cover them all. See, instead, DeFoor (2006), Lafler (2003; 2004a; 2005), Lauderdale (2007), and Sherman (2004), introductory tutorials on using this powerful procedure and Lafler (2004b), Prairie (2005), and Schreier (2008) for more in-depth references. One final, important piece of information on using PROC SQL in SAS, however, is provided here--a description of "pass-through SQL".

PASS-THROUGH SQL. Pass-through SQL refers to the ability of SAS to communicate with other SQL engines (e.g., Oracle, MS SQL Server, etc.) in their native language. Most of these other SQL products have their own proprietary procedural language (e.g., Oracle's PL/SQL, Microsoft's T-SQL), and writing PROC SQL statements that use these languages does not result in the output achieved when executing those statements in their native environment. SAS, instead, returns an error. For example, in Oracle's PL/SQL there is a function (to_number) that can be used to convert character strings to numeric values. However, writing the following PROC SQL query against an Oracle datasource (where "patient_number" is a character string.) does not work:

```

PROC SQL;
  SELECT TO_NUMBER(patient_number), SUM(charge) AS charges
  FROM billing.charge_detail
  WHERE DATEPART(bill_date) BETWEEN '01JUL2010'd AND '31JUL2010'd
GROUP BY TO_NUMBER(patient_number);
QUIT;

```

ERROR: Function TO_NUMBER could not be located.

This is particularly frustrating at first because you might think "TO_NUMBER is right there in the Oracle database "billing" to which I am connected." You might be tempted to rewrite your LIBNAME connection to the Oracle server, thinking you don't have the "path" correct or the connection has timed out. Although both of those things might be true, what this error is telling you is that within SAS, no function called "to_number" is found. The problem is not in the database server, it is in where you are trying to execute this function. It needs to be executed on the server (not SAS), and you need to find a way to tell SAS to "pass it through" to the server.

```

PROC SQL;
  CONNECT to ORACLE (
    USER = cschacherer
    DBPROMPT = yes
    PATH = "billing");
  CREATE TABLE july_charge_summary AS
  SELECT * FROM CONNECTION TO ORACLE
    (SELECT TO_NUMBER(patient_number), SUM(charges) as charges
     FROM charge_detail
     WHERE bill_date BETWEEN TO_DATE('07/01/2010','MM/DD/YYYY') AND
                           TO_DATE('07/31/2010','MM/DD/YYYY')
     GROUP BY TO_NUMBER(patient_number));
  DISCONNECT FROM ORACLE;
QUIT;

```

What this code is doing is creating a connection to the Oracle database "billing" and then creating the SAS dataset "july_charge_summary" by selecting, from that connection to Oracle, the result of a native PL/SQL SQL query of the Oracle database. This native Oracle SQL query is being packaged and sent to Oracle and because we are selecting

from the "CONNECTION TO ORACLE" SAS knows to not attempt to interpret the SQL query with its own SQL engine—but to pass it instead to the specified Oracle SQL Engine connection. Whatever result is returned from that native Oracle query will be selected by the SELECT statement against the connection to Oracle. If it is a result set, you could select individual data elements from it explicitly, or you can use SELECT * to simply select all returned data elements. Finally, it is good network, security, and database hygiene to clean up after yourself by issuing a DISCONNECT from the database so your connection does not remain open.

We have only scratched the surface of PROC SQL, but there are a host of wonderful PROC SQL references available. We highly recommend developing a facility with this workhorse PROC.

CONCLUSION

Throughout this paper we have attempted to provide new SAS users with an overview of the most common programming concepts and point out techniques and methods that might be immediately beneficial. In reality, most new SAS users in healthcare analytics come to have their first exposure to SAS through an existing program that is designed to perform a specific function—produce a report, load a data mart, or produce a data extract for a third-party data aggregator. Some never venture beyond that exposure to wonder "what else could SAS do for me in my day-to-day work?" Clearly, if you have read this far, you are not in that latter group. Hopefully at least one of the examples in the paper will spark an idea, and you will continue to pursue The Power to Know®.

REFERENCES

- Aster, R. & Seidman, R. (1997). Professional SAS Programming Secrets. McGraw-Hill.
- Buck, D. (2005). A Hands-On Introduction to SAS DATA Step Programming. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Carpenter, A.L. (2000). Functioning with Data Step Functions. Proceedings of the 25th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Cassidy, D. (2004). An Introduction to SAS Function-ality. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Cody, R. (2007a). Learning SAS by Example: A Programmer's Guide. Cary, NC: SAS Institute, Inc.
- Cody, R. (2007b). An Introduction to SAS Character Functions. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.
- Cody, R. (2010). SAS Functions by Example, 2nd Edition. Cary, NC: SAS Institute, Inc.
- Cody, R. (2010). Using Advanced Features of User-Defined Formats and Informats. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.
- DeFoor, J. (2006). Proc SQL - A Primer for SAS Programmers. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Droogen, H. (2004). (In)Formats (In)Decently Exposed. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Gilsen, B. (1999). SAS Program Efficiency for Beginners. Proceedings of the 24th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Gona, V. & Van Wyk, J. (1998). Version 7 Enhancements to SAS/ACCESS Software. Proceedings of the 23rd Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Gupta, S. (2006). WHERE vs. IF Statements: Knowing the Difference in How and When to Apply. Proceedings of the 31st Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Harp, A.H. (1998). Working with SAS Date and Time Functions. Proceedings of the 23rd Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Helf, G. W. (2002). Can't Relate? A Primer on Using SAS With Your Relational Database. Proceedings of the 27th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Howard, N. (1999). Advanced DATA Step Topics. Proceedings of the 24th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Howard, N. (1999). Introduction to SAS Functions. Proceedings of the 24th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Howard, N. (2003). How SAS Thinks or Why the DATA Step Does What it Does. Proceedings of the 28th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Lafler, K.P. (2000). Efficient SAS Programming Techniques. Proceedings of the 25th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Lafler, K.P. (2003). Undocumented and Hard-to-find SQL Features. Proceedings of the 28th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

- Lafler, K.P. (2004a). Efficiency Techniques for Beginning PROC SQL Users. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Lafler, K.P. (2004b). PROC SQL: Beyond the Basics Using SAS. Cary, NC: SAS Institute, Inc.
- Lafler, K.P. (2005). Manipulating Data with PROC SQL. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Lauderdale, K. (2007). PROC SQL - The Dark Side of SAS? Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.
- Levin, L. (2004). Methods of Storing SAS Data into Oracle Tables. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Levine, F. (2001). Using SAS/ACCESS Libname Technology to Get Improvements in Performand and Optimizations in SAS/SQL Queries. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Morgan, D.P. (2006). The Essential Guide to SAS Dates and Times. Cary, NC: SAS Institute, Inc.
- Prairie, K. (2005). The Essential PROC SQL Handbook for SAS Users. Cary, NC: SAS Institute, Inc.
- Rhodes, D.L. (2007). Talking to Your RDBMS Using SAS/ACCESS. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.
- Riley, C. (2008). Getting SAS to Play Nice With Others: Connecting SAS to Microsoft SQL Server Using an ODBC Connection. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.
- SAS Institute, Inc. (2004a). SAS/ACCESS 9.1 Supplement for Microsoft SQL Server. Cary, NC: SAS Institute, Inc.
- SAS Institute, Inc. (2004b). SAS/ACCESS 9.1 Supplement for Oracle. Cary, NC: SAS Institute, Inc.
- SAS Institute, Inc. (2010a). Accessing a Microsoft SQL Server Database from SAS on Microsoft Windows. SAS Technical Support Document TS-765. Retrieved July 28, 2010 from: <http://support.sas.com/techsup/technote/ts765.pdf>.
- SAS Institute, Inc. (2010b). SAS Learning Edition>>SAS Procedures. Retrieved July 28, 2010 from: http://support.sas.com/learn/le/proc/proc_2.html.
- Scerbo, M. (2007). Win the Pennant! Use PROC FORMAT. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.
- Schreier, H. (2008). PROC SQL by Example: Using SQL within SAS. Cary, NC: SAS Institute, Inc.
- Sherman, P.D. (2004). Creating Efficient SQL - Union Join without the Union Clause. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- Whitlock, I. (2007). How to Think Through the SAS DATA Step. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.

ACKNOWLEDGMENTS

The authors wish to thank Arlene Guindon, Amy Johnson, and Brian Rotty of Mayo Clinic Health Solutions for making this paper possible.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You can contact the author at:

Christopher W. Schacherer, Ph.D.
 Clinical Data Management Systems, LLC
 6666 Odana Road #505
 Madison, WI 53719
 Phone: 608.630.2637
 E-mail: CSchacherer@cdms-llc.com
 Web: www.cdms-llc.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

"The Power to Know" is a registered trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.