ABSTRACT
PROC SQL was added to the SAS system specifically to appease new SAS users who were already familiar with SQL language. Many programmers already familiar with SAS are hesitant to learn PROC SQL; after all, everything that can be done with PROC SQL can also be accomplished by the data step we already know. However, PROC SQL accomplishes some tasks more efficiently than the data step method. This paper covers the basics of SQL, and offers a few examples demonstrating the benefits of using SQL.

INTRODUCTION
Many database systems use SQL (Structured Query Language) as the method for retrieving and analyzing data. When programmers already familiar with SQL began to use SAS, they found that they would prefer to use SQL in some situations. SAS developers, upon hearing this request, obliged relatively immediately with PROC SQL. Programmers already comfortable with SAS had little use for this new functionality. SQL, however, has its place even in the most experienced SAS user’s code. Not to mention the fact that SQL is relatively platform independent; once it has been learned in SAS that information can easily be transferred to work with ORACLE or other databases.

The main deterrent to using SQL is that it is essentially an entirely different language to learn. Few programmers want to invest precious time learning something they deem unnecessary. In an effort to expedite the process of learning PROC SQL, the following paper will cover the basics one needs to know to get started. Each topic is presented so that a person trained in the SAS language should be able to quickly learn SQL. After the basic topics have been covered, a few selected examples showing some of the handier uses of SQL will be offered.

SQL BASICS
The term “SQL Basics,” as used here, can best be interpreted as those SQL statements appearing in the majority of SQL queries. While there are many other keywords and options in the SQL system, what is presented here will allow you to begin “thinking in SQL.” As you become more comfortable using SQL and have need of more involved methods, you will be able to easily find and understand the necessary keywords and options in the PROC SQL documentation for the SAS System.

TURNING IT ON
Before we can use SQL keywords and statements we must use the code:
```sql
proc sql;
```
This “turns on” the SQL system. In essence, it tells SAS “heads up, what’s about to follow are SQL statements not SAS statements, please read them accordingly.” Once the system is turned on, as many datasets as desired can be created. SAS will continue to read any code as SQL statements until the SQL system is turned off. (See TURNING IT OFF.)

SPECIFYING THE OUTPUT DATASET
To create a dataset called DSNNAME with a data step we say:
```sql
data DSNNAME;
```
To create the same dataset using SQL, we say:
```sql
proc sql;
create table DSNNAME as
```
In the SQL system, creating a dataset is optional. If we simply omit the “create table DSNNAME as” section of the code, the SQL system processes all the statements and prints the results to the output file, instead of creating a dataset.

SPECIFYING A SINGLE INPUT DATASET
The next step in a typical SAS data step is to specify the set statement (i.e. input dataset). The code looks like this:
```sql
data dsnname;
set indsnA;
```
or like this:
```sql
data dsnname;
merge indsnA indsnB;
by subject;
```
In an SQL statement, the specification of input datasets comes later in the code, but since we are SAS programmers it is our nature to think about the input dataset next. If we want to gather data from one dataset, the SQL code is:
```sql
proc sql;
create table DSNNAME as
<==various SQL statements,
to be discussed later>>>
from indsnA
```

SPECIFYING MULTIPLE INPUT DATASETS
If we want to gather data from more than one dataset, we can use a join or a union. Unions are outside the scope of this paper, so we will focus only on joins. A join can be thought of as a merge, as long as we remember it behaves a little differently than the SAS merge statement.

The SAS merge statement merges the first matching observation in one dataset to the first matching observation in the other dataset. If there are multiple matches, the merge puts the first match in INDSNA with the first available match in INDSNB, then puts the second match in INDSNA with the second available match in INDSNB, and so forth. If there are more matches in one dataset than the other, the merge continues to put the last match in the one dataset with every remaining match in the other dataset. For example, if INDSNA had 3 observations with subject=1, and INDSNB had 3 observations with subject=1, like this:

INDSNA
<table>
<thead>
<tr>
<th>Subject</th>
<th>varA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
</tbody>
</table>

INDSNB
<table>
<thead>
<tr>
<th>Subject</th>
<th>varB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
</tbody>
</table>

When we merge on subject, a note would be printed to the log stating “MERGE statement had more than one dataset with repeats of a by value” and the resulting dataset, DSNNAME, would have 3 observations like this:

DSNNAME
<table>
<thead>
<tr>
<th>Subject</th>
<th>varA</th>
<th>varB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

If, instead, INDSNA had 3 observations with subject=1 and INDSNB had only 2 observations with subject=1, then the resulting
dataset DSNNAME would still have 3 observations, but like this:

<table>
<thead>
<tr>
<th>Subject</th>
<th>varA</th>
<th>varB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>

A join in SQL puts every observation in one dataset with every matching observation in the other dataset. For our first example, INDSNA and INDSNB both with 3 observations with subject=1, the resulting dataset DSNNAME would have 9 observations:

<table>
<thead>
<tr>
<th>Subject</th>
<th>varA</th>
<th>varB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

For our second example, INDSNA with 3 observations with subject=1 and INDSNB with only 2, the resulting dataset DSNNAME would have 6 observations like this:

<table>
<thead>
<tr>
<th>Subject</th>
<th>varA</th>
<th>varB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

There are 4 types of joins available: full join, left join, right join and inner join. A full join keeps all the matches and any observations from both datasets which did not have a match. A left join keeps all the matches and any observations from the dataset listed first (i.e. the one on the left) which did not have a match. A right join keeps all the matches and any observations from the dataset listed second (i.e. the one on the right) which did not have a match. An inner join keeps only the observations that had matches. There are two advantages to an SQL join versus a data step merge: (1) a separate sort is not required before a join (however, if you are working with large datasets, a sort can significantly improve the efficiency of the join); and (2) datasets with multiple repeats of a by value can be merged, keeping all the observations.

To use a join, the SQL code looks like this:

```
proc sql;
create table DSNNAME as
    create various SQL statements,
    to be discussed later
from indsnA as a
    full join
indsnB as b
on a.subject=b.subject
```

The important pieces of this code are (1) give the datasets an “alias,” (2) specify what type of join to use and (3) tell SQL what variables to join on. There are shortcut methods; for example, aliases are not necessary if the datasets have no variable names in common and a comma between the datasets implies an inner join. Including all three parts will make the join work exactly as expected every time. So, for simplicity, we will assume all three parts are necessary.

First, give the datasets an alias. This is the “as a” and “as b” section of the code. This alias will be used in other parts of the SQL statement to refer back to the source dataset for the variables used. (See SELECTING VARIABLES and CREATING VARIABLES.) The alias can be a single letter or a string of letters. For example, instead of

```
from indsnA as a
```

we could have used

```
from indsnA as indsnA
```

For efficiency considerations, both in typing and in executing, it is best to keep aliases short.

Second, specify the type of join to use. The join specified depends entirely on what is expected from the data. To better understand the differences in joins, we can compare them to merges. If we had two datasets with ONE observation for each subject, and we used an inner join:

```
from indsnA as a
inner join
indsnB as b
on a.subject=b.subject
```

This would be the same as:

```
merge indsnA(in=a) indsnB(in=b);
by subject;
if a and b;
```

If we used a left join, this would be the same as:

```
if a;
```

If we used a right join, this would be the same as:

```
if b;
```

If we used a full join, this would be the same as:

```
if a or b;
```

These comparisons are based on a maximum of one match for each observation in either dataset. If there is more than one match, the differences in how SAS merges and how SQL joins, will make the resulting datasets different.

Third, tell SQL what criteria to join on. Typically, the criterion is for a variable in one dataset to equal a variable in the other. The variables do not have to be the same length or have the same name, but they do need to be the same type, (i.e. character or numeric). As few as 1 criterion, and as many as needed, can be used. To use additional criteria for the join, simply put an “AND” between each statement:

```
from indsnA as a
full join
indsnB as b
on a.subject=b.subject
and a.visitdt=b.labdt
```

SELECTING VARIABLES

The main portion of a SAS data step is given over to specifying which input variables to keep and to creating new variables. The main portion of an SQL statement is given over to the same task, but the task is executed in a slightly different manner.

By default, a SAS data statement keeps every variable from all input datasets. SQL must be told every time what variables to keep. This is done with the SELECT statement. To keep all variables from the input dataset, the following code is used:

```
proc sql;
create table DSNNAME as
    select *
from indsnA
```

or, if using joined datasets, with this code:

```
proc sql;
create table DSNNAME as
    select a.*, b.*
from indsnA as a
    full join
indsnB as b
on a.subject=b.subject
```
If we do not want to keep all the variables from all the data sets, with a DATA statement we would use a KEEP. With an SQL statement, we simply specify which variables to select:

```
proc sql;
    create table DSNNAME as
    select subject,
           var1,
           var2
    from indsnA
```

Notice that a comma is used to separate each variable. Commas are used extensively in SQL statements, primarily to delineate one variable from another.

To select variables from joined tables, use the alias assigned in the join step to specify which dataset the variable comes from:

```
proc sql;
    create table DSNNAME as
    select a.subject,
           a.varA1,
           a.varA2,
           b.varB1,
           b.varB2
    from indsnA as a
    full join
    indsnB as b
    on a.subject=b.subject
```

The code in both above-mentioned examples would select each variable from the input datasets with the variable's existing name and attributes. To rename the variable and/or change its attributes, use code similar to:

```
proc sql;
    create table DSNNAME as
    select subject as ID format=$9.,
           var1 as age label='Age',
           var2 as sex format=1. label='Sex'
    from indsnA
```

Notice again, the commas delineate between variables.

CREATING VARIABLES AS MATHEMATICAL OPERATIONS

We can also create new variables with SQL. We can create exactly the same types of variables we would create in a data step, but using the language of SQL.

In a data step, to create a variable as a mathematical operation of 2 other variables, the code would be:

```
format newvar 7.2;
newvar=((var1-var2)/b_var)*100;
```

To create this same variable in SQL, the code would be:

```
proc sql;
    create table DSNNAME as
    select subject as ID format=$9.,
           var1 as age label='Age',
           var2 as sex format=1. label='Sex'
    from indsnA
```

Notice that if we want to use a variable that is created in the SQL statement, ABSCHG in our example, we must put the keyword CALCULATED in front of it. This tells SQL to look for the variable in the dataset being made, instead of looking in the input dataset.

CREATING VARIABLES BASED ON OTHER VARIABLES

In a data step, to create a variable based on the value of other variables the code might look like this:

```
fomat agecat 1.;
    if . lt age le 18 then agecat=1;
    else if 18 lt age le 25 then agecat=2;
    else if 25 lt age le 40 then agecat=3;
    else if age gt 40 then agecat=4;
end as agecat format=1.
```

We can create variables in the same manner in SQL, by using the CASE expression. The code looks like this:

```
proc sql;
    create table DSNNAME as
    select subject as ID format=$9.,
           var1 as age label='Age',
           var2 as sex format=1. label='Sex'
    from indsnA
```

The CASE expression functions much like the SELECT statement in a SAS data step. The keyword CASE begins the step. Then the various WHEN...THEN statements specify what criteria are to be used to assign each value. Notice the "else;" as with a SELECT statement, if this is omitted SQL will print a NOTE to the log. The keyword END closes the CASE expression. The AS statement specifies the name, and any other information about, the variable that is to be created by the CASE expression.

Variables can also be created by using functions specific to SQL. These will be discussed in the POWER OF SQL Section.

SUBSETTING THE INPUT DATASET

The code developed so far, keeps every observation from the input datasets. Often only select observations from the input datasets are to be used in the final data set. In a typical SAS data step, the code to subset the input dataset would be:

```
data dsnname;
    set indsnA(where=(varA1 gt 35 and varA2 ne 'M'));
```

To keep the same subset in an SQL statement, the code would be:

```
proc sql;
    create table DSNNAME as
    select * from indsnA
    where varA1 gt 35
    and varA2 ne 'M'
```

To keep only specific observations from one of the datasets in a data step merge:

```
data dsnname;
    merge indsnA(where=(varA1 gt 35))
    indsnB;
    by subject;
```

SQL code:

```
proc sql;
    create table DSNNAME as
    select a.subject,
           a.varA1,
           a.varA2,
           b.varB1,
           b.varB2
```
from indsnA as a 
full join 
indsnB as b 
on a.subject=b.subject 
where a.varA1 gt 35

An alternate form for the previous SQL code would be:

```sql
proc sql;
create table DSNNAME as
select a.subject 
, a.varA1 
, a.varA2 
, b.varB1 
, b.varB2 
from indsnA(where=(varA1 gt 35)) as a 
full join 
indsnB as b 
on a.subject=b.subject
```

SUBSETTING THE OUTPUT DATASET

In some situations, it is only necessary to retain a subset of the final dataset. Often, some variable created in the data step is needed to subset the final output. For example, SAS data step code might be:

```sas
data dsname;
merge indsnA(where=(varA1 gt 35)) indsnB;
by subject;
newvar=((var1-var2)/b_var)*100;
if newvar gt 50;
run;
```

To subset the output dataset in an SQL step, using a variable created in the statement, we must use the HAVING keyword. The SQL code for the above would be:

```sql
proc sql;
create table DSNNAME as
select subject 
, var1
, var2
, ((var1-var2)/b_var)*100
as newwar format=7.2
, b_var
, varA1
from indsnA
where varA1 gt 35
having calculated newvar gt 50;
```

Notice that the HAVING statement comes after the WHERE statement. It is not necessary to have both, we may use either/or. If both are present, the HAVING statement must come after the WHERE statement.

SORTING THE DATA

Using a data step to create a dataset does not allow the output dataset to sorted in any particular order. The output dataset will be in the same order as the input dataset(s). A separate PROC SORT would be required, if we wanted the data in a different order. With SQL, however, we can build a sort directly into the creation of the dataset using the ORDER BY statement.

```sql
proc sql;
create table DSNNAME as
select subject 
, var1
, var2
, ((var1-var2)/b_var)*100
as newwar format=7.2
, b_var
, varA1
from indsnA
where varA1 gt 35
having calculated newvar gt 50
```

Note that the ORDER BY statement always comes after the FROM statement. It also comes after the WHERE statement, but before the HAVING statement, if either of these are used.

ENDING THE DATA STEP

Once we have finished selecting and creating all the variables, we must close the step. In a SAS DATA step, we would use a RUN to do this:

```sas
data dsname;
merge indsnA(where=(varA1 gt 35)) 
indsnB;
by subject;
newvar=((var1-var2)/b_var)*100;
if newvar gt 50;
run;
```

To close a statement in SQL we use a semi-colon;. Like this:

```sql
proc sql;
create table DSNNAME as
select subject 
, var1
, var2
, ((var1-var2)/b_var)*100
as newwar format=7.2
, b_var
, varA1
from indsnA
where varA1 gt 35
having calculated newvar gt 50;
```

After the semi-colon, we can immediately begin another SQL statement. For example:

```sql
proc sql;
create table DSNNAME2 as
select a.*,
b.varB1
from INDSNA as a
inner join
INDSNB as b
having a.varA1=b.varB1;
```

The above code will create two independent datasets, DSNNAME1 and DSNNAME2. We may continue with as many, or as few, SQL statements as desired.

TURNING IT OFF

Once we are finished using SQL, and are ready to go back to using standard SAS code, we must turn the SQL system off. This is done with a QUIT; statement:

```sql
proc sql;
create table DSNNAME as
select subject 
, var1
, var2
, ((var1-var2)/b_var)*100
as newwar format=7.2
, b_var
, varA1
from indsnA
where varA1 gt 35
having calculated newvar gt 50;
quit;
```

Now, we have told SAS “Okay, I’m done with SQL now. Everything from here on out will be regular SAS code. Please
read it accordingly.” If we don’t turn SQL off, SAS will continue to try and read our SAS data steps as SQL statements. We can use SQL again at anytime simply by turning it back on (see TURNING IT ON).

POWER OF SQL
All that has been discussed thus far is how to make SQL do what we already know how to make a SAS data step do. At the moment, there is no apparent benefit, except the potential advantages of a join. It was, however, necessary to first understand how to construct an SQL statement before we can take advantage of SQL’s power.

SQL’s power lies mainly in the fact that it looks at the input datasets differently than does a SAS data step. In a SAS data step, each observation is processed separately. We can look at only one observation at a time. Looking at the observations before or after the line being processed is difficult. It can be done with some imaginative uses of the RETAIN and LAG statements and/or PROC TRANSPOSE, but it is not easy.

SQL, on the other hand, looks at the input dataset as a whole. Thus, it can very easily take into account the observations before and after a given observation. For example, we can look at all the observations for a given subject as a whole and make calculations based on what the “whole picture” looks like. To do this, we need to learn about three things: SQL specific summary functions, the GROUP BY statement and the keyword DISTINCT.

SQL FUNCTIONS
PROC SQL has several specific summary functions available for use. The ones most commonly are the following:

- COUNT counts the values
- SUM sums the values
- MAX identifies the largest value
- MIN identifies the smallest value
- MEAN averages the values

A complete list of available functions can be found in the SAS Procedures Guide in the PROC SQL documentation under “summary-function.”

GROUP BY
Utilizing the full power of the SQL summary functions requires the GROUP BY statement. This statement defines for SQL what we want the summary functions performed on (ie. what we want to see as “the whole picture”). If we omit the GROUP BY statement, SQL understands “the whole picture” to be the whole dataset. Thus, all the summary functions will be performed on the whole dataset.

For example,

```sql
create table MAXDSN as
select subject
, visitdt
, max(var1) as max
from indsnA
; quit;
```

will create a dataset MAXDSN in which there are two variables, SUBJECT from the input dataset and MAX. MAX, however, will now be the maximum value of VAR1 for each subject. The variable MAX will be the same for every observation for each subject.

We can extend the GROUP BY statement to more than one variable. If we wanted to look at the maximum value for each subject on each day, then we would use:

```sql
create table MAXDSN as
select subject
, visitdt
, max(var1) as max
from indsnA
group by subject, visitdt;
quit;
```

Now the output dataset MAXDSN would have three variables, SUBJECT and VISITDT from the input dataset and MAX. MAX will now have the maximum value of VAR1 for each subject on each visit date. The variable MAX will be the same for every observation with the same subject and same visit date.

Notice that the GROUP BY statement comes after the FROM statement. If we had used a WHERE statement, the WHERE statement would have followed the FROM statement and the GROUP BY statement would have followed it. A HAVING statement, if employed, would follow the GROUP BY statement.

KEYWORD DISTINCT
SQL summary functions group values from several observations into one value and then use that value for every observation. Frequently, the final dataset will have multiple observations which are identical. To have only one observation, we can use the SQL keyword DISTINCT. DISTINCT tells SQL to pick each unique combination of values for the specified variables. For example, we can use a dataset like this:

```sql
INDSN
Subject var1 var2
1   10   25
1   10   16
1   10   25
2   11   25
2   12   25
2   19   25
```

If we use the following code:

```sql
select DISTINCT subject
, var2
from indsnA;
```

we get the following output:

```sql
Subject var2
1   25
1   16
2   25
```

If we use the following code:

```sql
select DISTINCT subject
, var1
, var2
from indsnA;
```

we get the following output:

```sql
Subject var1 var2
1   10   25
1   10   16
2   11   25
2   12   25
2   19   25
```
Our example from the GROUP BY section,

```
proc sql;
    create table MAXDSN as
    select DISTINCT subject
         , visitdt
    from indsnA
    group by subject, visitdt;
quit;
```

will now create a dataset MAXDSN which has a single observation for each unique value of subject and each value of visitdt available for that subject and the value of MAX for that subject and visitdt. Note that when DISTINCT is used this way, only each unique combination is kept in the output dataset, but ALL the observations are used in calculating the value of the SQL summary functions.

HANDBY USES FOR SQL

Now that we understand the basics of building SQL statements, we can look at some examples of how SQL can be used. Additional keywords and concepts that were not covered in the main part of the paper (for simplicity and brevity) are included in some of the examples. These concepts are explained as they are presented.

COMPARING DATASETS

SQL is extremely useful for extracting observations from one dataset based on the presence of corresponding observations in another dataset. To do this with a data step would require (1) sorting both datasets, (2) merging, (3) identifying which observations are desired and (4) subsetting out the desired observations. By using SQL and a subquery, the process is simplified. In SAS data step terms, a subquery is the creation of an intermediate dataset which is used to create the output dataset. The SQL code, with a subquery, to select observations NOT in both datasets would look like this:

```
proc sql;
    create table diffsub as
    select *
    from indsnA
    where subject not in (select distinct
                           subject from indsnB);
quit;
```

The subquery (“select distinct subject from indsnB”) creates a list of all the subjects in INDNSB. Then, in the WHERE statement, all the subjects in INDNSA are compared to this list. Only the observations for the subjects that are not in the list are kept in the output dataset.

If we are interested in observations that are in BOTH datasets we could use:

```
where subject in (select distinct
                 subject from indsnB)
```

to select the observations for subjects in INDSNA that are also in INDSNB.

FINDING SUBJECTS WHO DO NOT HAVE AN OBSERVATION

When we have unexpectedly missing data, we need to go back to the source data and figure out the reasons why. Figuring out which subjects DO have specific observations, is as simple as using a WHERE statement and getting a list. But figuring out which subjects do NOT have observations can be much more complicated.

Suppose, for example, we wanted to know which subjects had a specific lab value at week=5. We could use the following code to get a list of subjects who had a non-missing value at week=5:

```
Proc print data=inlabs;
    Var subject;
    (where=(week=5 and labval ne .));
run;
```

If, however, we try using "where week=5 and labval = ". to get a list of the subject’s who do NOT have an observation, we get only a list of those subjects who have an observation at week=5 but with a missing value. The subjects who have no week=5 observation at all will still be omitted. To get the same section (see COMPARING DATASETS):

```
Or, we can use a simple variation on the SQL code presented in that same section (see COMPARING DATASETS):
```

```
proc sql;
    select distinct subject
    from inlabs
    where subject not in (select distinct
                          subject from inlabs where
                          week=5 and labval ne .);
quit;
```

As in the previous section, this code uses a subquery. It first generates a list of the subjects in the input dataset (inlabs) which DO have observations. Then it compares the input dataset against that list and keeps the subject number for only those subjects that are NOT in the list. The end result is a complete list of subjects who DO NOT have a value at week=5 printed to the .LST file.

GETTING TOTAL COUNTS

Getting counts of all shapes and sizes is a common task in SAS. Accurate counts require (1) sorting the dataset, (2) un-duplicating the dataset, and (3) using PROC FREQ or PROC UNIVARIATE to get the count. With SQL, we can skip steps 1 and 2. The SQL code would look like this:

```
proc sql;
    create table COUNTS as
    select count(distinct id)
         , agecat
    from indsnA
    where varA ne .
    group by agecat;
quit;
```

Notice the use of the keyword DISTINCT inside the SQL summary function. This tells the COUNT function to count each ID only once, which is why there is no need to un-duplicate the data. We can use DISTINCT inside any summary function to tell SQL to use each unique value only once in the calculation.

PUTTING CALCULATED VALUES INTO MACROS

SQL’s capability to put calculated values directly into macro variables is also useful. Say, in the example from GETTING TOTAL COUNTS above, we wanted to put the total counts in each age category into macro variables. To do this in a data step code, we would need steps 1-3 detailed in that section, to calculate accurate counts. Then we would need a 4th data step using CALL SYMPUT to put the counts in macro variables. We can do all 4 steps in a single step with SQL, using the INTO function.

```
proc sql;
    select count(distinct id)
    from indsnA
    where agecat ne .
    group by agecat;
    into :cat1 - :cat4
quit;
```
The INTO function takes each of the values for the variable [here for count(distinct id)] and stores them in the listed macro variables. Some things to note about making macro variables:

1. You cannot use the 'create table' statement and the INTO function in the same SQL statement.
2. SQL will put the values into the macro variables as they are encountered. For our example, we have 4 possible age categories. If the data only has observations in age categories 1 and 3, then &cat1 will have the count for age category=1, &cat2 will have the count for age category=3, and &cat3 and &cat4 will be empty.
3. Different calculations can be put into different macro variables using a single SQL statement. For example, we could put the age categories into macros and put the max age for each category into another macro variable in the same step. We use a SELECT statement to list all the calculations to be made, followed by an INTO statement to tell SQL where to store each of the calculated values. The code would look like this:

```sql
proc sql;
    select count(distinct id)
       , max(age) into :cat1 - :cat4
       , max(age) - :maxage1 - :maxage4
    from indsnA
    where agecat ne .
    group by agecat;
quit;
```

In the output file, (i.e the .lst file) we would have our counts neatly listed under headings. The only requirement is that there be only one observation per subject. If there are multiple observations for a subject, then that subject will be incorrectly counted more than once in the output sum.

### VALIDATING CATEGORICAL VARIABLES – TYPE 1

Creating categorical variables is often a key step in programming. A quick and easy way to ensure that these categorical variables have been created correctly can be an invaluable asset. When the categorical variable is created from other categorical (aka. discrete) variables, a PROC FREQ is generally used to ensure the accuracy of the new variable. This PROC can give a lot of extra information to dig through.

Using SQL, the categorical variables can be checked with the results displayed in a very concise format. For example, say a variable newcat was created based on the values of catvar1, catvar2 and catvar3. A PROC FREQ could be used to check this new variable:

```sql
proc sql;
    Title "Validate NEWCAT variable";
    select distinct newcat
        , catvar1
        , catvar2
        , catvar3
    from indsnA
    group by newcat;
quit;
```

The output file will now have a list of each value of NEWCAT and the associated values for variables that went into making it:

<table>
<thead>
<tr>
<th>NEWCAT</th>
<th>CATVAR1</th>
<th>CATVAR2</th>
<th>CATVAR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

### VALIDATING CATEGORICAL VARIABLES – TYPE 2

When categorical variables are created from continuous data, a PROC UNIVARIATE can generally be used to check the accuracy, with code like:

```sql
proc sql;
    Title "Validate NEWCAT variable";
    select distinct newcat
        , min(invar1) as min_var
        , max(invar1) as max_var
    from indsnA
    group by newcat;
quit;
```

The output file will now have a list of each of the NEWCAT values and the minimum and maximum values of the input variables for each value. Like this:

<table>
<thead>
<tr>
<th>NEWCAT</th>
<th>MIN_VAR</th>
<th>MAX_VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>67</td>
<td>70</td>
</tr>
</tbody>
</table>

### IDENTIFYING CHARACTERISTICS OF DATA

SQL’s SUM function can be very useful for getting a quick overview of the data. For example, we can use it to get a quick count of the number of subjects in each of our analysis sets:

```sql
proc sql;
    select sum(randdt ne .) as random
       , sum(fdosedt ne .) as safety
       , sum(ldosedy gt 124) as analysis
       , sum(respvar="Y") as respond
    from crt.keyvars;
quit;
```

When categorical variables are created from both discrete and continuous data, it becomes even more difficult to check the values with standard SAS DATA step and PROC codes. But with SQL it is simply a matter of combining the two previous techniques.

For example, assume a categorical variable is created based on two criteria: (1) the subject’s age category and (2) the subject’s baseline value for a key lab parameter. We can quickly and easily check our outcome with SQL:

```sql
proc sql;
    Title "Validate NEWCAT variable";
    select distinct newcat
        , catvar1
        , catvar2
        , catvar3
    from indsnA
    group by newcat;
quit;
```

The output file will now have a list of each value of NEWCAT and the associated values for variables that went into making it:
proc sql;
  Title "Validate NEWCAT variable";
  select distinct newcat
    ,   agecat
    ,   min(b_keylab) as min_lab
    ,   max(b_keylab) as max_lab
  from indsnA
  group by newcat;
quit;

This neatly displays in the output file, all the information we need to check the new categorical variable:

<table>
<thead>
<tr>
<th>NEWCAT</th>
<th>AGECAT</th>
<th>MIN_LAB</th>
<th>MAX_LAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>1</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>3</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>4</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4.3</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3.3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3.3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3.3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10.2</td>
<td>13.8</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>10.2</td>
<td>13.8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10.2</td>
<td>13.8</td>
</tr>
</tbody>
</table>

MERGING ON CLOSE VALUES
A SAS data step merge requires an exact match. This may not always be possible; for example, we have one dataset with observations taken on a specific date and another dataset with observations taken within a day of that date. To merge these two datasets using a SAS data step, we would have to merge once for exact matches, outputting the observations with no matches into another dataset. Add 1 to all the dates with no match and merge again. With an SQL join we can do this merge in a single step:

```
proc sql;
create table closmerg as
select a.*
    ,   b.*
from indsnA as a
inner join
indsnB as b
on a.subject=b.patient
and abs(b.visitdt-a.visitdt) le 1;
quit;
```

This joins the datasets on subject and visit dates which are within one day of each other. If there are multiple matches, then all the possible matches will be retained in the final output dataset. Since we used an inner join, only the observations for which we find a match will be kept in the final dataset.

MERGING BETWEEN TWO VALUES
Another situation which may require a merge, but does not offer the exact match required by the standard SAS merge, is when one dataset has two dates and the other dataset has a date that needs to fall between those two dates. For example, we may set a “start” and “stop” date for the treatment “week”. Then we have daily dosing information that needs to be matched into those “weeks”. This can, of course, be done in several different ways using the standard SAS data step. But every way is complicated. SQL offers a much simpler method:

```
proc sql;
create table mid_merge as
select a.*
    ,   b.*
from indsnA as a
inner join
indsnB as b
on a.subject=b.patient
and (a.startdt le b.visitdt le a.stopdt);
quit;
```

Or we could do both these steps in a single SQL statement:

```
proc sql;
create table finalDSN as
select distinct a.subject
    ,   a.startdt
    ,   a.stopdt
    ,   sum(b.missed="Y") as missno
    ,   sum(b.offsched="Y") as offshno
    ,   sum(b.visitdt ne .) as numdoses
from mid_merge
group by a.subject, a.startdt;
quit;
```

CONCLUSION
PROC SQL has many more capabilities far outside the scope of this paper. As you use SQL and have need of more involved processes, you can find these other capabilities in the SQL procedure documentation of the SAS Manuals and the SAS Online Documentation. Most importantly, you should now understand the basics well enough to decide if SQL would make a task easier than using the standard SAS data step. The main point, after all, is to get at the data as quickly and easily as possible. SQL simply provides another means to that end.

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

Erin Christen
STATPROBE, Inc.
10052 Mesa Ridge Court
Suite 200
San Diego, CA  92121
Work Phone: (865) 522 -0993
Email: erin.christen@sd.statprobe.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.

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