Proc SQL versus The Data Step
JoAnn Matthews, Highmark Blue Shield, Pittsburgh, PA

ABSTRACT
Most data within organizations is stored in relational databases. Structured Query Language (SQL) has evolved as the standard for accessing, updating, and modifying data stored in relational databases. PROC SQL is a powerful procedure available in SAS® that can minimize keystrokes and maximize CPU resources. PROC SQL syntax looks very similar to SQL and can be used in place of traditional SAS data steps.

In this Hands–On Workshop, PROC SQL will be compared to the traditional data step. First, PROC SQL will be compared to a simple data step. SELECT coupled with FROM, WHERE, GROUP BY and ORDER BY statements will be demonstrated. CASE statements will be compared to IF THEN ELSE logic. Summary functions using SUM, MIN, AVG, MAX and COUNT will be demonstrated. The true power of PROC SQL will be demonstrated with UNION ALL, UNION DISTINCT and JOINS. Three types of join will be discussed, EQUIJOINS, based on one common value, INNER JOINS, that discard all rows from the resultant table not having a corresponding row in the source table, and OUTER JOINS, joins that exclude unmatched data. This workshop will also be helpful to those who are new to SAS or Display Manager in Version 9.

INTRODUCTION
Given the complex data that is used by many large corporations, most corporations store their data in relational database systems or DMBS, such as Oracle, DB2, Access, Teradata tables, or permanent SAS® datasets. Structured Query Language (SQL) is defined as a standard interactive programming language for getting information from and updating a database. Many database products support SQL commonly called ‘sequel.’ SQL has evolved as the standard for defining, accessing, updating, manipulating and modifying data stored in relational databases. SQL is carefully controlled and standardized by guidelines set up by the American National Standards Institute (ANSI).

SQL is one of the most commonly used query languages for relational data base management systems (DBMS). A data base management system (DBMS) is a set of programs used to define, administer and process data in a database. DBMS programs run on a multitude of platforms including the mainframe, minicomputers and personal computers. In many instances DBMS’s reside on a several platforms within an organization, and contain all three classes of machine, mainframe, mini and PC. A DBMS that runs on multiple platforms, large and small, is called ‘scalable’. The flow of information within the DBMS is from the user at the keyboard, to the interface, to the program, through the DBMS, and finally to the database. SQL is a set of user friendly commands that are flexible and easy to understand. The query is written in syntax that is like English, making it easy to interpret. A query is a question to the database. The beauty of SQL is that when the data in the database satisfies your query, SQL retrieves the data in the most efficient way possible. SQL is “nonprocedural” meaning that you state what you want, and the DBMS decides how best to retrieve the answer. In contrast, a procedural language, such as FORTRAN, BASIC, C, or Java, requires that you to write a “procedure” to program what you want and need.

PROC SQL and SAS
SQL married SAS sometime after Version 5. PROC SQL was developed by SAS to make use of some of the powerful components of SQL. PROC SQL is so similar to SQL that anyone who has programmed using standard query language will quickly understand the basics. The PROC SQL procedure available in SAS can minimize keystrokes and maximize CPU resources, particularly when working with very large databases. There are some subtle differences with naming and security conventions between ANSI SQL statements and the SAS SQL procedure but these differences are so subtle that the average SAS user need not be concerned about them. For anyone who has ever used SQL, the syntax of the PROC SQL statement looks similar to standard SQL and can be used in place of the traditional SAS data step.

When we compare the SQL procedure to the data step, there are some differences that should be taken into consideration. One big difference is that you do not need to use the run after the procedure. PROC SQL will run without the run statement. If you include the RUN statement in the PROC SQL, SAS ignores the RUN and will give you a warning statement.
You do need a QUIT statement at the end of the PROC SQL procedure.

You can execute a number of SQL steps within the procedure without having to repeat the PROC SQL statement. This is an advantage when manipulating data. In traditional data step processing, each step must begin with a procedure.

In PROC SQL, variables (columns) are separated with commas, not blanks as in a data step.

The Select statement in PROC SQL outputs the data automatically, so that you do not have to execute a PROC PRINT statement to see your output.

One of the most significant differences in the two methods is that when using PROC SQL, the table does not have to be sorted. When using traditional data step processing, each table must be sorted before merging. This may not be a major issue when using small datasets, but when using millions of observations, presorting a dataset can be a major space issue. Using PROC SQL can be a major advantage when working with large volumes of data.

BASIC SQL SYNTAX

The most common components used in the SQL procedure are: SELECT, UPDATE, DELETE, CREATE, DROP, INSERT, RESET, VALIDATE, ALTER and DESCRIBE. In alphabetical order, these SQL statements are described:

ALTER   <is used to add, delete or alter columns in a table>
CREATE   <is used to create a view or table>
DELETE   <is used to delete rows from a table>
DESCRIBE <is used to explain how a view has been defined>
DROP     <is used to delete a table or view>
INSERT   <is used to add rows to a table>
UPDATE   <is used to update a table>
VALIDATE <is used to validate SQL syntax>
DESCRIBE <is used to describe the table>
RESET    <is used to add, change or alter options on the PROC SQL line>
SELECT   <is used to generate a report and is the most important statement as it evaluates the query, formats rows and sends the output to the output window>

Within the SELECT statement there are subcomponents:

SELECT   <FIRST AND MOST IMPORTANT COMPONENT – identifies what your query will contain>
FROM     <is used to identify the source table for the query>
WHERE    <is used to select specific rows or subset the query>
ORDER BY <is used to define the order of the data>
GROUP BY <is used to group the data>

Let's begin with a table of sleep participants' demographic information. The name of this table is mine.demographic and it resides on the c:drive. This table contains 20 observations (rows) and 7 variables (columns). In order to confirm this data, let's do a PROC CONTENTS in a traditional data step. Go to the program editor and type the following code, then highlight the code, put your cursor over the run icon, and execute the code.

```sql
Hands-On Workshops
NESUG 2006
```

libname mine 'c:\deleteme\how';
proc contents data=mine.demographic;
run;

The PROC CONTENTS tells you everything you need to know about your data including the number of observations, the number of variables, the names and characteristics of the variables. PROC CONTENTS is a robust method to learn characteristics of the dataset. If the data is stored in a permanent dataset, running a PROC CONTENTS allows the programmer to become familiar with a dataset. And a word of caution, get in the habit of checking your log every single time you run a bit of code. Checking for errors and warnings is critical.

Now check the log. The LOG window contains valuable information about your session, as well as notes about the code that you submitted. When doing a PROC CONTENTS, the log only tells you that the procedure ran. The output has all the stats about the dataset.
Now go to the output window. Your output from the PROC CONTENTS should look like this:

Notice the top right portion of the output. Observations and Variables are described. In this dataset we have 20 observations (rows) and 7 variables (columns). Observe the bottom of the output where attributes of the variables are described. Let’s print all the variables using a traditional data step. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```
libname mine 'c:\deleteme\how';
data a;
set mine.demographic;
run;
proc print data=mine.demographic;
run;
```

Notice that to see the data, you must run a second procedure, a PROC PRINT.
To accomplish the same thing with PROC SQL, we would use the following code with an asterisk in the SELECT statement to bring in all the rows and columns. Think of the asterisk as a wild card that says “Give me everything.” Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
PROC SQL;
    Select *
    From mine.demographic;
run;
quit;
```

Notice that if this PROC SQL statement is correct, the output appears automatically in the output window without having to type proc print. This is one of the differences cited in the section describing differences between PROC SQL and the data step.
Always check the log. Notice that the run statement produced a warning that it had no effect. You do not need a run statement when using PROC SQL. You do however need a `quit;` statement. This is another example of two of the differences cited in the section on differences between PROC SQL and the data step.

### SUBSETTING THE DATA

Now let's subset the data by defining just those columns that we want using a subsetting `WHERE` clause. In this instance, let's extract the sleep participants “WHERE age > 50”.

Using a data step, we would set the table and then do a proc print naming just those variables that we wanted to see printed. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.
Data a (keep = l_name sex age);
Set mine.demographic;
If age > 50;
Run;

Proc print data=a;
Run;

You should see the following output.

Now let's subset this data using PROC SQL. Go to the Program Editor window, clear the log and type the following code, then execute the code by using the submit icon on your toolbar.
PROC SQL;
    SELECT l_name, sex, age
    From mine.demographic
    Where age > 50;
quit;

Notice that variables in the SELECT statement in PROC SQL require a comma between them but not at the end of the select statement. This is another difference between PROC SQL and the data step. Again, unlike the data step, the PROC SQL statement does not require a run statement. It will produce a message into the log that says the run statement has no effect if you do include it.

You should see the following output.

So far, you will notice that the PROC SQL code does use a few less keystrokes, but the output is similar to the data step.
ORDER BY AND GROUP BY
Let’s take the WHERE clause and build upon it. Using the same logic as above, let’s put some order to our data. ORDER BY returns rows in ascending order unless you specify DESC, for descending. Let’s order persons over 50 in descending order. Notice that instead of using the actual variable name, we can use an alias. The number 1 says give me the variable that is in position 1 on the SELECT statement. This is an example of a relative column number that is used in substitute for a variable name. A relative column number can be used in the ORDER BY and GROUP BY statements. Go to the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

PROC SQL;
    SELECT l_name, sex, age
    From mine.demographic
    Where age > 50
    Order by 1 DESC;
quit;

Your output should look like this.

You can group the data by the same variable with the GROUP BY statement. The syntax for this would look thus:

PROC SQL;
    SELECT l_name, sex, age
    From mine.demographic
    Where age > 50
    Order by 1;
    Group by 1;
quit;

One word of caution: Do not try to GROUP BY using summarized values. An error message pertaining to non-aggregate values on the GROUP BY statement can be confusing but says that you have tried to do a group by with summarized values. This is a common error when first using the GROUP BY.

DISTINCT FUNCTION
The DISTINCT function in the SELECT statement is very powerful. If you have more than one row for a value and you only want one value returned from your query, use DISTINCT. Let’s bring in all the variables in our table to see how many distinct last names there are in this dataset. In a traditional data step we would have to set the
data, sort the data, then subset the data by the sort field, using an **IF FIRST.** statement, and finally print the data.

Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sas
DATA a;
SET MINE.demographic;

PROC SORT;
BY l_name;

DATA HOWMANY;
SET a;
BY l_name;
IF FIRST.l_name;
RUN;

PROC PRINT DATA=HOWMANY;
VAR l_name;
RUN;
```

Your output should look like this:
With PROC SQL the distinct operator passes the statement directly to the DBMS, not through SAS, so that the operator checks for duplicate rows. Distinct is a very powerful operator in the PROC SQL statement. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```
PROC SQL;
    select distinct l_name
    from mine.demographic
    quit;
```
Notice that the output appears immediately. Notice also how much less code this query required as compared to a traditional data step. The beauty of SQL is impressive when the table contains millions of observations, and lots of distinct values.

PROC SQL using the SELECT DISTINCT can be used for one or more variables at the same time, and the table does not have to be presorted. This truly is a powerful difference between traditional data steps and PROC SQL. In addition, you can use DISTINCT with COUNT to nest the query and count the number of DISTINCT values. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
PROC SQL;
   select count (distinct l_name)
   from mine.demographic
quit;
```
The count function counted the number of distinct last names in the data. Your output should look like this:

![Output Image]

**CASE STATEMENT**
The CASE statement allows you to give meaningful names to variables or created fields in your program. This would compare to a data step using *if then else* logic. Note that the value ‘missing’ will be truncated (cut off in this example) unless you leave spaces for six characters. Let’s create meaningful values for the sex variable using traditional data steps. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sas
Data a;
Set mine.demographic;

If sex = 1 then sex_desc = 'Male '; /* leave two spaces*/
Else if sex=2 then sex_desc = 'Female';
Else sex_desc = 'missing';

Proc print data = a;
Var sex sex_desc;

Run;
```
Your output should look like this:

Now let's compare the CASE statement using PROC SQL. The following is a simple case statement for our Disney sleep research data. Notice that the CASE statement requires an END statement. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
PROC SQL;
  SELECT sex,
            Case When sex eq 1 then 'Male'
                When sex eq 2 then 'Female'
                Else 'Missing' end as Sex_Desc
  From mine.demographic;
QUIT;
```
You should see the following output:

You can create many new variables by combining case statements, and the syntax would look thus:

```sql
PROC SQL;
   SELECT sex, rel, employ,
   Case When sex eq 1 then 'Male' 
     When sex eq 2 then 'Female'
     Else 'Missing' end as Sex_Desc,
   Case When rel eq 1 then 'Father' 
     When rel eq 2 then 'Mother'
     When rel eq 3 then 'Daughter'
     When rel eq 4 then 'Son'
     Else 'Missing' end as Rel_Desc,
FROM some_table;
QUIT;
```
Case When employ eq 1 then 'Employee '
    When employ eq 2 then 'Dependent'
Else 'Missing' end as Emp_Desc,
from mine.demographic;
QUIT;

SUMMARY FUNCTIONS
Let's take that code we just created and remove the CASE statement and modify it to calculate the mean age with a summary function. Notice we are giving the summary statistic an alias so that it has a meaningful label. We have generated the mean, minimum and maximum but there are lots of other summary functions, including AVG, MEAN, COUNT, FREQ, MIN, MAX, NMISS, to name a few. Revise the code from our last PROC SQL, then execute the code by using the submit icon on your toolbar.

PROC SQL;
    SELECT mean (age) as average_age,
          Min (age) as minimum_age,
          Max (age) as maximum_age
From mine.demographic;
QUIT;

Your output should look like this:
MERGING VERSUS JOINS – The true power of PROC SQL

The true power of the PROC SQL procedure will become apparent when you are merging very large datasets or combining two datasets with some common variable. Because PROC SQL does not require presorting of the tables you are joining, computer resources are saved when using joins. There are ways to think about which is more advantageous, to merge using a data step or join tables using PROC SQL.

ONE TO ONE MERGES

For one to one matching, both the data step and PROC SQL are acceptable and use about the same resources. Begin by considering a simple SAS program that takes data from two tables and merges both into one final table: The demographics data set has 20 ID numbers and demographic information about each person. The rem_sleep dataset also has 20 rem sleep values for the same id’s. We want all demographic data and all the rem sleep data to be combined into one table with 20 observations. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```
DATA A;
SET mine.demographic;
Proc sort;
BY ID_NO;
RUN;
/*this dataset contains one family's demographic data and one table with one observation of rem sleep data per person. */

DATA B;
SET mine.rem_sleep;
Proc sort;
BY ID_NO;
RUN;
/*we want all demographics and all the rem sleep variables in one row for all 20 persons*/

DATA ALL;
MERGE A B;
```
In this example we are doing a one to one merge. That is to say we have one match from each table. We can merge the two datasets and the resulting final dataset will contain one observation with id_no and the variables from each dataset.

Submit the code and output should look like this.
We can accomplish the same merging of these two datasets with a simple PROC SQL. In this instance we want to join both tables on a common variable, id_no. An inner join will discard any rows that do not have a corresponding row in both source tables. In this sample, however there is an equal match for each row, i.e. a ONE to ONE merge. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
PROC SQL;
SELECT *
FROM mine.demographic, mine.rem_sleep
WHERE demographic.id_no = rem_sleep.id_no;
QUIT;
```

You should see the following in your output window:
ONE TO MANY MERGES
Like one to one merges, one to many merges can be accomplished either with data steps or an PROC SQL procedure. So far we have worked with very small datasets. Let’s look at a larger dataset that contains multiple evaluation times for Disney characters who have participated in a sleep study for depression. The dataset is called mine.demo_diag. It contains all the Disney characters evaluation times and diagnoses. The mine.dx file contains one value for each of 33 psychiatric diagnoses. What we want to see are only those diagnoses for the Disney dataset (n=32). Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sas
libname mine 'c:\deleteme\how';
Data a (keep = id_no f_name l_name date diag);
Set mine.demo_diag;

proc sort data=a;
By diag;
Run;

proc print data=a;
run;

Data b;
Set mine.dx;

proc sort data=b;
By diag;
run;

proc print data=b;
Run;

Data all;
Merge a b;
By diag;
Run;

Proc print data = all;
Run;
```

Hands-On Workshops
NESUG 2006
In this example we are doing a one to many merge, with many id_no’s in the first dataset but only one value for each of the diagnosis numbers on the second dataset. We want to merge the datasets and have one diagnosis description for each row of the demo_diag table or n = 32.

Look at the output. Displayed is the second page of output, and it is messy and incorrect.

Check the log. We want to see 32 rows of data in our final merge. Unfortunately, what we have is the total of the two tables. 32 rows from demo_dx and 33 rows from the diagnosis table. The final table contains 52 rows, not 32.
In order to correct the data step, we need to add the IN = option to our merged data. The IN = option creates a new temporary variable that allows you to select only those observations that are on the data set A. Go back to the program editor window, and add two more lines of code (in red), then execute.

```plaintext
Data all;
Merge a (in=x) b (in=y);
By diag;
if x;
Run
```

Check the log and confirm that you have only 32 observations. Now look at the output. The data is clean and the results are the correct number of rows (n=32).

We can accomplish this same result with a simple PROC SQL and the code is significantly shorter. Go to the Program Editor, clear the log and output and type the following code: Notice how we are specifically naming the variables depending on what table they are being queried from – i.e. demo_diag.id_no. *(Give me id_no from the mine.demo_diag table, please).*

```plaintext
PROC SQL;
Select demo_diag.id_no,
demo_diag.f_name,
demo_diag.l_name,
demo_diag.date,
demo_diag.diag,
dx.desc
From mine.demo_diag, mine.dx
Where demo_diag.diag = dx.diag;
Quit;
```
Notice the log. It does not display the number of observations. You need to know your data and what number of observations you would expect, when using PROC SQL.

Note also that this is an example of an inner join. An inner join is the result of variables that are in one table that have a match in a second table. These are also called equijoins. An equijoin combines tables based on a common variable in both tables, eliminating the redundant columns.

i.e. Where a.id = b.id

And your output should look like this:
Keep in mind that you can join more than one table at a time. In fact PROC SQL can join up to 32 tables. The syntax for two tables would look like this:

```
PROC SQL;
SELECT *
from mine.demographic, mine.rem_sleep
WHERE demographic.id_no = rem_sleep.id_no
   and demographic.place = rem_sleep.place;
quit;
```

While both the data step and PROC SQL can be used for one to many merges, where PROC SQL becomes very powerful is when accessing huge datasets. When using small datasets, the data step may be just as efficient as PROC SQL but when accessing huge datasets or large tables with many rows and many columns, PROC SQL is more powerful and generally uses less resources.

**MANY TO MANY MATCHES**

Let's use a new rem sleep dataset that contains multiple sleep evaluations for each of our Disney families. The study is now several years in funding, and every participant was asked to complete a second and third sleep time to establish average REM times. Some families were able and some were not. We need to merge the two tables of multiple sleep REM values. This is an example of many to many merge. Use PROC SQL when the merge is a many to many merge and especially when some data is missing. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
libname mine 'c:\deleteme\how';

Data a (keep = id_no f_name l_name date diag);
Set mine.demo_diag;
proc sort;
By id_no;
Run;

proc print data=a;
run;

Data b;
Set mine.rem_two;
```
What you get is messy and incorrect. The best way to do a MANY to MANY MATCH is with PROC SQL. This is particularly true if some of the values are missing, as PROC SQL will adjust for this condition.

Let's do a many to many merge with PROC SQL using the WHERE diag on demo_diag matches diag on demo_dx. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
PROC SQL;
SELECT demo_diag.id_no,
       demo_diag.f_name,
       demo_diag.l_name,
       demo_diag.date,
       demo_diag.diag,
       demo_dx.desc
FROM mine.demo_diag, mine.demo_dx
WHERE demo_diag.diag = demo_dx.diag;
QUIT;
```
This output is correct with correct diagnoses for each member of the study.

Let's look at the log.
Notice the log for PROC SQL does not confirm the number of observations, as the log does when using the data step! This can be a real disadvantage of SQL. With the data step you see exactly how many observations were read in for each data step. Checking the log can help you to be assured that the data you are getting is the data you really want. With SQL, YOU MUST KNOW YOUR DATA! If not, use only a few observations, using the INOBS = n option on the PROC SQL statement,

i.e. PROC SQL inobs = 10;

The INOBS = option restricts the number of rows (observations) that PROC SQL will process. Check the log:
CARTESIAN PRODUCT

If you join two tables without using the WHERE clause, you will get the product of the two numbers (a Cartesian product). $20 \times 20 = 100$. Generally, a Cartesian join is accidental. Usually if you want to join two tables you are intending to do a concatenation, using SET. An accidental Cartesian, when done on a large table can bring your system to its knees. 10,000 rows x 10,000 rows may be more than your system resource allocation will allow.

If your intent is to accomplish a Cartesian product, this is difficult to do using traditional data step processing. For example a table with 20 observations in one table, and 20 observations in the other table will result in 400 observations in total. Cartesian products can best be accomplished by the use of PROC SQL. Here is an example of a Cartesian product but PROC SQL was smart enough to know this isn’t what we really want to do, because we should be using a WHERE statement.

Let’s make an intentional programming error. Go to the SAS Editor window, clear the log and type the following code, then execute the code by using the submit icon on your toolbar.

```sql
PROC SQL;
CREATE table all as
SELECT *
from mine.demographic, mine.rem_sleep;
QUIT;
```

Since id_no appears on both tables, you will see an error message as a result of trying to do a Cartesian product (a join without a WHERE clause) on two datasets, a many to many merge. Most Cartesian products are accidental and are resource intensive.
OUTER JOINS

In contract to an inner join, an outer join retrieves records from both tables but only returns those observations that do not match the first table. When you join two tables, the first one may have rows that don’t have matching counterparts in the second table. Conversely, the table on the right may have rows that don’t have a match in the table on the left. If you do an inner join on those tables all the unmatched rows are excluded. There are three types of outer join, left outer, right outer and full outer join. The syntax is similar to the inner join, except that you substitute outer for inner. The left outer join preserves unmatched rows from the left table but discards unmatched rows from the right table. Right outer joins preserves unmatched rows from the right table but discards unmatched rows from the left table. You can use this on the same tables and get the same result by reversing the order in which you present the tables in the join: Full outer joins combine the functions of the left outer join and the right outer join. It retains the unmatched rows from both the left and the right tables.

UNION ALL

Suppose you have three tables that you wish to concatenate. You can use the SET statement in SAS® to concatenate the tables. Or you can use the UNION ALL, as long as all the variables in all the tables are input in the same position. This type of union creates another table that has everything, all the columns in all the source tables.

The Disney participants used three separate sleep labs. The data for those three labs needs to be combined into one table. LAB1 was housed in Magic Kingdom and contains 11 sleep evaluations (rows of data) for the Mouse family. LAB2 was in Epcot Center and contains 9 sleep evaluations (9 rows) for the Duck family. The third sleep lab was in Animal Kingdom and contains 12 sleep evaluations for the Bunny, Goat and Beaver families. Using a traditional data step to concatenate we would use the SET, instead of MERGE. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```
libname mine ':\delete\me\how';

Data a;
Set mine.lab1;
proc sort;
By id_no;
Run;
```
```sas
proc print data=a;
run;

Data b;
Set mine.lab2;

proc sort;
By id_no;

proc print data=b;
Run;

data c;
set mine.lab3;

proc sort;
by id_no;
run;

proc print data=c;
run;

Data all;
set a  b  c;
By id_no;
Run;

Proc print data=all;
by id_no;
run;
```

Always check the log when running data steps. Let's examine the log.
Observe the number of observations in the final table ALL. We would expect to see 32 observations, 11 from A, 9 from B, and 12 from C. Look at the output window and observe that the final dataset appears clean and concatenated.

Now let's use a UNION ALL in PROC SQL. One quick and dirty way to do a UNION ALL is to simply use the asterisk function that says “Give me all the variables”. This works beautifully if both tables have the exact same format. If the tables are union compatible, i.e. share the same format, the resultant table would be all the rows in the first table, plus all the rows from the second table. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.
libname mine 'C:\delete\emehow';

PROC SQL;

CREATE table all AS
    SELECT * From mine.lab1

UNION ALL
    SELECT * from mine.lab2;

UNION ALL
    SELECT * from mine.lab3;
quit;

Always check your log. Make sure that the row counts are what you expect.
If on the other hand, the tables are not union compatible, you can use the UNION ALL and explicitly define the variables that you wish to concatenate. Explicitly listing the columns that you want rather than relying on the * shorthand is usually a good idea. It is quite possible that even though the tables were union compatible when you first ran the query, when running the query later, one of the tables could have been modified. The resultant tables are no longer union compatible. Explicit definition is always safer.

Let's do a join with explicitly defining the variables. Go into the program editor, clear the log and output window, and type the following code, then execute the code by using the submit icon on your toolbar.

```
libname mine 'C:\deleteme\how';

PROC SQL;
CREATE table mine.all AS
  (Select id_no, f_name, l_name, eval_time, date, rel, sex, age, days, cost
   from mine.lab1
   UNION ALL
   Select id_no, f_name, l_name, eval_time, date, rel, sex, age, days, cost
   from mine.lab2
   UNION ALL
   Select id_no, f_name, l_name, eval_time, date, rel, sex, age, days, cost
   from mine.lab3
  );
quit;
```
Check the log. With UNION ALL, you will see how many total rows and columns were created.

**UNION DISTINCT**

If you want duplicate rows eliminated from the final table use the UNION DISTINCT function. It behaves like the UNION ALL, but returns only non duplicate rows.

```sql
libname mine 'C:\deleteme\how';

PROC SQL;

CREATE table mine.all AS
```
CONCLUSION:

PROC SQL is a very powerful addition to your SAS bag of tools. When working with huge datasets, the CPU and resource savings can be dramatic. The beauty of PROC SQL is that the data does not need to be sorted like with the traditional data step. This Hands-On Workshop is intended to get you started exploring the SQL procedure in SAS. Each of the PROC SQL statements that we have discussed has many additional capabilities which are beyond the scope of this paper. And while we used the Display Manager in SAS to submit the code, check the log and list file, the PROC SQL syntax will work on any platform, whether you are running on the mainframe, UNIX, or SAS Version 9's Enterprise Guide.

REFERENCES:

Michael J. Larkins and Thomas L. Coffing, Jr., Teradata SQL, Unleash the Power, Coffing Publishing, 2002

ACKNOWLEDGMENTS

SAS® and all 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 registered trademarks of their respective companies.

CONTACT INFORMATION

Your comments or questions are encouraged. Please feel free to contact the author at:

JoAnn Matthews
Highmark Inc
120 Fifth Avenue
Pittsburgh PA 15222-3099
e-mail: joann.matthews@highmark.com