Paper 62

An Introduction to PROC SQL®

Rosalind K. Gusinow and J. Michael Miscisin
Systems Seminar Consultants, Madison, WI

Abstract
PROC SQL is a powerful Base SAS® PROC which combines the functionality of the DATA and PROC Steps into a single procedure. PROC SQL in many cases can be a more efficient alternative to traditional SAS code.

PROC SQL can be used to retrieve, update, and report on information from SAS data sets or other database products. This workshop will concentrate on SQL's syntax and how to access information from existing SAS data sets. Some of the topics covered include:

- Write SQL code using various styles of the SELECT statement.
- Dynamically create new variables on the SELECT statement.
- Use SQL options to control the appearance of reports.
- Create multiple reports on a single PROC SQL statement.
- Create reports containing percentages using PROC SQL.
- Use CASE/WHEN clauses for conditionally processing the data.
- Joining data from two data sets (like a MERGE!).

Why Learn PROC SQL?
PROC SQL can not only retrieve information without having to learn SAS syntax, but it can also oftentimes do this with fewer and shorter statements than traditional SAS code. Additionally, on average it uses fewer resources than conventional DATA and PROC steps. This means PROC SQL is usually a more efficient alternative to traditional SAS code. Further, the knowledge learned is transferrable.

An Example of PROC SQL's Syntax
Every PROC SQL must have at least one SELECT statement. The purpose of the SELECT statement is to name the columns that will appear on the report and the order in which they will appear (similar to a VAR statement on PROC PRINT). The FROM clause names the data set from which the information will come from (similar to the SET statement). One advantages of SQL is that new variables can be dynamically created on the SELECT statement, which is a feature we do not normally associate with a SAS Procedure:

```
PROC SQL;
 SELECT STATE, SALES,
  (SALES * .05) AS TAX
 FROM USSALES;
QUIT;
```
(no output shown for this code)

The SELECT Statement's Syntax
The purpose of the SELECT statement is to describe how the report will look. It consists of the SELECT statement and several sub-clauses. The purpose of the sub-clauses is to name the input dataset, order (or sort) the data, group (or aggregate) the data, and select rows meeting certain conditions (subsetting):

```
PROC SQL options;
 SELECT column(s)
 FROM table-name | view-name
 ORDER BY column(s)
 GROUP BY column(s)
 WHERE expression
 HAVING expression;
QUIT;
```

A Simple PROC SQL
An '*' on the SELECT statement will select all columns. By default a row will wrap when there is too much information to fit across the page. Also by default, column headings will be separated from the data with a line and no observation number will appear:

```
PROC SQL;
 SELECT *
 FROM USSALES;
QUIT;
```
(see output #1 for results)

Limiting Information on the SELECT
Multiple requests are delimited by commas on the SELECT statement. The SELECT statement DOES NOT limit the number of variables read.

The NUMBER option will print a column on the report labeled 'ROW' which contains the observation number:

```
PROC SQL NUMBER;
 SELECT STATE, SALES
 FROM USSALES;
QUIT;
```

(see output #2 for results)

Creating New Variables

Variables can be dynamically created in PROC SQL. Dynamically created variables can be given a variable name, label, or neither. If a dynamically created variable is not given a name or a label, it will appear on the report as a column with no column heading associated with it. Any of the DATA step functions can be used in an expression to create a new variable except LAG, DIF, and SOUND:

```
PROC SQL;
 SELECT SUBSTR(STORENO,1,3)
  LABEL='REGION', SALES,
  (SALES * .05) AS TAX, (SALES * .05) * .01
 FROM USSALES;
QUIT;
```

(see output #3 for results)

Options on the PROC SQL Statement

There are several useful options that can be used on the PROC SQL statement to help control the appearance of the report. Be careful, once coded, these options will apply to all SELECT statements within PROC SQL unless a RESET statement is used:

```
PROC SQL INOBS=100 OUTOBS=9 DOUBLE;
 SELECT STORE, (SALES * .05) AS TAX
 FROM USSALES;
QUIT;
```

(see output #4 for results)

The FLOW Option and Using RESET

The FLOW option allows text to continue in its column rather than wrapping the text on to the next line. If a value is not specified on the FLOW option, SAS will "flow" the value to the length of the column. The RESET statement changes options within the same step without respecifying the procedure. The option FLOW=30 40 floats the width of the column between the values specified to produce a better layout:

```
PROC SQL FLOW=30;
 SELECT STATE, STORENAM, COMMENT
 FROM USSALES;

 RESET FLOW=30 40 DOUBLE NUMBER
  OUTOBS=150;

 SELECT STATE, STORENAM, COMMENT
 FROM USSALES;
QUIT;
```

(see output #5 for results)

Note: multiple SELECT clauses can be coded under a single PROC SQL. Each SELECT clause will generate a separate report.

The CALCULATED Option on the SELECT

Starting with Version 6.07, the CALCULATED component refers to a previously calculated variable so recalculation is not necessary. The CALCULATED component must refer to a variable created in the same SELECT statement as it is used:

```
PROC SQL INOBS=9;
 SELECT STATE, (SALES * .05) AS TAX,
  (SALES * .05) * .01 AS REBATE
 FROM USSALES;

 SELECT STATE, (SALES * .05) AS TAX,
  CALCULATED TAX * .01 AS REBATE
 FROM USSALES;
QUIT;
```

(see output #6 for results)

Associating LABELS and FORMATS

SAS-defined or user-defined formats can be used to improve the appearance of the body of a report. By default variable names appear as column headings on reports. LABELS gives the ability to define up to forty characters to appear as column headings on the report. Both LABELS and FORMATS DO NOT change the way in which a value or variable is stored. They are for appearances ONLY. Be sure when providing formats that their values are adequately large enough, otherwise the values will not appear fully formatted on the report:

```
TITLE 'REPORT OF THE U.S. SALES';
FOOTNOTE 'PREPARED BY THE MARKETING
          DEPT.';
```

```
OPTIONS LS=132 PS=80;

PROC SQL;
 SELECT STATE,SALES
  FORMAT=DOLLAR10.2
  LABEL='AMOUNT OF SALES',
  (SALES * .05) AS TAX
  FORMAT=DOLLAR7.2
  LABEL='5% TAX'
 FROM USSALES;
QUIT;
```

(see output #7 for results)

The CASE Expression on the SELECT

The CASE Expression allows conditional processing within PROC SQL:

```
PROC SQL;
 SELECT STATE,
  CASE
  WHEN SALES<10000 THEN 'LOW'
  WHEN SALES<15000 THEN 'AVG'
  WHEN SALES<20000 THEN 'HIGH'
  ELSE 'VERY HIGH'
  END AS SALESCAT
 FROM USSALES;
QUIT;
```

(see results #8 for results)

The END is required when using the CASE. Coding the WHEN in descending order of probability will improve efficiency because it will stop checking when it finds the first value to be true. You do not have to worry about the length of the newly created variable's value being assigned with WHENs like you do with the IF.

The CASE Expression can be coded many different ways; perhaps this WHEN clause looks more familiar to you. It does the exact same thing as the previous syntax:

```
PROC SQL;
 SELECT STATE,
  CASE
  WHEN 0    <= SALES <= 10000
    THEN 'LOW'
   WHEN 10001 <= SALES <= 15000
    THEN 'AVG'
  WHEN 15001 <= SALES <= 20000
    THEN 'HIGH'
   ELSE 'VERY HIGH'
  END AS SALESCAT
 FROM USSALES;
QUIT;
```

(output is same as output #8)

Basically, you can do all the same things on a CASE statement as you can on an IF. Here is yet another variation on the CASE expression:

```
PROC SQL;
 SELECT STATE,
  CASE
  WHEN SALES > 20000 AND STORENO
    IN ('33281','31983') THEN 'CHECKIT'
   ELSE 'OKAY'
  END AS SALESCAT
 FROM USSALES;
QUIT;
```

(see output #9 for results)

Additional SELECT Statement Clauses

The GROUP BY clause can be used to summarize or aggregate data. Summary functions (also referred to as aggregate functions) are used on the SELECT statement for each of the analysis variables:

```
PROC SQL;
 SELECT STATE, SUM(SALES) AS TOTSALES
  FROM USSALES
  GROUP BY STATE;
QUIT;
```

(see output #10 for results)

Other summary functions available are the AVG/MEAN, COUNT/FREQ/N, MAX, MIN, NMISS, STD, SUM, and VAR.

Remerging will occur when a summary function is used without a GROUP BY. The result is a grand total shown on every line:

```
PROC SQL;
 SELECT STATE, SUM(SALES) AS TOTSALES
  FROM USSALES;
QUIT;
```

(see output #11 for results)

Sometimes remerging is good, as in the case when the SELECT statement does not contain any character variables, or in the case of calculating a percentage:

```
PROC SQL;
 SELECT SUM(SALES) AS TOTSALES
 FROM USSALES;
QUIT;
```
(see output #12 for results)

```
PROC SQL;
 SELECT STATE, SALES,
      (SALES/SUM(SALES)) AS PCTSALES

  FORMAT=PERCENT7.2
 FROM USSALES;
QUIT;
```
(see output #13 for results)

Always check your output carefully when the remerging note appears in your log to determine if you have gotten the desired results.

Sorting the Data in PROC SQL
The ORDER BY clause will return the data in sorted order:

```
PROC SQL;
 SELECT STATE, SALES
 FROM USSALES
 ORDER BY STATE, SALES DESC;
QUIT;
```
(see output #14 for results)

Much like PROC SORT, if the data are already in sorted order, PROC SQL will print a message in the LOG stating the sorting utility was not used. When sorting on an existing column, PROC SQL and PROC SORT are nearly comparable in terms of efficiency. SQL is more efficient when you need to sort on a dynamically created variable:

```
PROC SQL;
 SELECT SUBSTR(STORENO,1,3)
  LABEL='REGION',
  (SALES * .05) AS TAX
 FROM USSALES
 ORDER BY 1 ASC, TAX DESC;
QUIT;
```
(see output #15 for results)

Columns can be referred to by their name or by their position on either the ORDER BY or GROUP BY clauses. The option 'ASC' (ascending) on the ORDER BY clause is the default, it does not need to be specified.

Subsetting Using the WHERE
The WHERE statement will subset rows before they are read:

```
PROC SQL;
 SELECT *
 FROM USSALES
 WHERE STATE IN
  ('OH','IN','IL');

 SELECT *
 FROM USSALES
 WHERE NSTATE IN (10 20 ,30);

 SELECT *
 FROM USSALES
 WHERE STATE IN
  ('OH','IN','IL')
  AND SALES > 500;
QUIT;
```
(no output shown for this example)

Be careful of the WHERE clause, it cannot reference a computed variable:

```
PROC SQL;
 SELECT STATE, SALES,
  (SALES * .05) AS TAX
 FROM USSALES
 WHERE STATE IN
  ('OH','IN','IL')
  AND TAX > 10 ;
QUIT;
```
(see output #16 for results)

To use computed variables on the WHERE clause they must be recomputed:

```
PROC SQL;
 SELECT STATE, SALES,
  (SALES * .05) AS TAX
 FROM USSALES
 WHERE STATE IN
  ('OH','IL','IN')
  AND (SALES * .05) > 10;
QUIT;
```
(see output #17 for results)

Also be aware that the WHERE statement cannot be used with the GROUP BY:

```
PROC SQL;
 SELECT STATE, STORE,
```

```
  SUM(SALES) AS TOTSALES
 FROM USSALES
 GROUP BY STATE, STORE DESC
 WHERE TOTSALES > 500;
QUIT;
```

(see output #18 for results)

In order to subset data when grouping is in effect, the HAVING statement must be used:

```
PROC SQL;
 SELECT STATE, STORENO,
  SUM(SALES) AS TOTSALES
 FROM USSALES
 GROUP BY STATE, STORENO
 HAVING SUM(SALES) > 500;
QUIT;
```

(see output #19 for results)

The HAVING clause is needed even if it is not referring to a computed variable:

```
PROC SQL;
 SELECT STATE,
  SUM(SALES) AS TOTSALES
 FROM USSALES
 GROUP BY STATE
 HAVING STATE IN ('IL','WI');
QUIT;
```

(see output #20 for results)

The CREATE Statement

The CREATE statement provides the ability to create a new data set as output in lieu of a report (which is what happens when a SELECT is present without a CREATE statement). The CREATE statement can either build a TABLE (a traditional SAS dataset, like what is built on a SAS DATA statement) or a VIEW (not covered in this paper):

```
PROC SQL;
 CREATE TABLE TESTA AS
 SELECT STATE, SALES
 FROM USSALES
 WHERE STATE IN ('IL','OH');

 SELECT * FROM TESTA;
QUIT;
```

(see output #21 for results)

The name given on the create statement can either be temporary or permanent. Only one table or view can be created by CREATE statement. The second SELECT statement (without a CREATE) is used to generate the report.

Joining Datasets Using Proc SQL

A join is used to combine information from multiple files. One advantage of using PROC SQL to join files is that it does not require sorting the datasets prior to joining as is required with a DATA step merge.

A Cartesion Join combines all rows from one file with all rows from another file. This type of join is difficult to perform using traditional SAS code.

```
PROC SQL;
  SELECT *
 FROM DATA1, DATA2;
QUIT;
```

(see output #22 for results)

A Conventional or Inner Join combines datasets only if an observation is in both datasets. This type of join is similar to a DATA step merge using the IN Data Set Option and IF logic requiring that the observation is on both data sets (IF ONA AND ONB).

```
PROC SQL;
  SELECT *
  FROM DATA1, DATA2;
  WHERE DATA1.VAR1=DATA2.VAR1;
QUIT;
```

(see output #23 for results)

An Associative Join combines information from three or more tables. Performing this operation using traditional SAS code would require several PROC SORTs and several DATA step merges. The same result can be achieved with one PROC SQL:

```
PROC SQL;
 SELECT B.FNAME, B.LNAME, CLAIMS,
        E.STORENO, STATE
 FROM   BENEFITS B, EMPLOYEE E,
        FEBSALES F;
 WHERE B.FNAME=E.FNAME AND
        B.LNAME=E.LNAME AND
        E.STORENO=F.STORENO AND
        CLAIMS >  1000;
QUIT;
```

(see output #24 for dataset list and results)

In Summary

PROC SQL is a powerful data analysis tool. It can perform many of the same operations as found in traditional SAS code, but can oftentimes do it more efficiently because of its dense language structure.

PROC SQL can be an effective tool for joining data, particularly when doing associative, or three-way joins. For more information regarding SQL joins reference the papers noted in the bibliography.

Trademark Notice

SAS and PROC SQL are registered trademarks of the SAS Institute Inc., Cary, NC, USA and other countries. ® indicates USA registration.

Useful Publications

SAS Institute Inc., Getting Started with the SQL Procedure, Version 6, First Edition

SAS Institute Inc., SAS® Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition

Dickson, Alan and Pass, Ray, *"Select Items from PROC.SQL Where Items > Basics"*, Proceedings of the 20th Annual SAS® Users Group International Conference

Kent, Paul, *"SQL Joins- The Long and Short of It"*, Proceedings of the 20th Annual SAS® Users Group International Conference

Kolbe Ritzow, Kim, *"Joining Data with SQL"*, Proceedings of the 6th Annual MidWest SAS® Users Group Conference

Kolbe Ritzow, Kim, *"An Introduction to PROC SQL"*, Proceedings of the 21st Annual SAS® Users Group International Conference

Lafler, Kirk Paul, *"Diving into SAS® Software with the SQL Procedure"*, Proceedings of the 20th Annual SAS® Users Group International Conference

Levin, Howard, *"A New Approach to Outer Joins with More than Two Tables"*, Proceedings of the 20th Annual SAS® Users Group International Conference

Any questions or comments regarding the paper may be directed to the authors:

Rosalind Gusinow and Michael Miscisin
Systems Seminar Consultants
2997 Yarmouth Greenway Drive
Madison, Wi 53711
Phone: (608) 278-9964
Fax: (608) 278-0065
E-mail: rgusinow@sys-seminar.com
mmiscisin@sys-seminar.com

Output #1 (partial):

```
STATE     SALES  STORENO
COMMENT
STORENAM
-------------------------------------------------
WI      10103.23  32331
SALES WERE SLOW BECAUSE OF COMPETITORS SALE
RON'S VALUE RITE STORE

WI       9103.23  32320
SALES SLOWER THAN NORMAL BECAUSE OF BAD WEATHER
PRICED SMART GROCERS

WI      15032.11  32311
AVERAGE SALES ACTIVITY REPORTED
VALUE CITY
```

Output #2 (partial):

ROW	STATE	SALES
1	WI	10103.23
2	WI	9103.23
3	WI	15032.11

Output #3 (partial):

REGION	SALES	TAX	
323	10103.23	505.1615	5.051615
323	9103.23	455.1615	4.551615
323	15032.11	751.6055	7.516055
332	33209.23	1660.462	16.60461

Output #4 (partial):

STATE	TAX
WI	505.1615
WI	455.1615
WI	751.6055
MI	1660.462

Output #5 (partial):

STATE	STORENAM	COMMENT
WI	RON'S VALUE RITE STORE	SALES WERE SLOW BECAUSE OF COMPETITORS SALE
WI	PRICED SMART GROCERS	SALES SLOWER THAN NORMAL BECAUSE OF BAD WEATHER

ROW	STATE	STORENAM	COMMENT
1	WI	RON'S VALUE RITE STORE	SALES WERE SLOW BECAUSE OF COMPETITORS SALE
2	WI	PRICED SMART GROCERS	SALES SLOWER THAN NORMAL BECAUSE OF BAD WEATHER

Output #6 (partial):

STATE	TAX	REBATE
WI	505.1615	5.051615
WI	455.1615	4.551615
WI	751.6055	7.516055
MI	1660.462	16.60461

Output #7 (partial):

REPORT OF THE U.S. SALES

STATE	AMOUNT OF SALES	5% TAX
WI	$10,103.23	$505.16
WI	$9,103.23	$455.16
WI	$15,032.11	$751.61
MI	$33,209.23	1660.46

PREPARED BY THE MARKETING DEPT.

Output #8 (partial):

STATE	SALESCAT
WI	AVG
WI	LOW
WI	HIGH
MI	VERY HIGH

Output #9 (partial):

STATE	SALESCAT
WI	OKAY
WI	OKAY
WI	OKAY
MI	CHECKIT

Output #10:

STATE	TOTSALES
IL	84976.57
MI	53341.66
WI	34238.57

Output #11 (partial):

STATE	TOTSALES
WI	172556.8
WI	172556.8
WI	172556.8
MI	172556.8

Output #12:

TOTSALES
172556.8

Output #13 (partial):

STATE	SALES	PCTSALES
WI	10103.23	5.86%
WI	9103.23	5.28%
WI	15032.11	8.71%
MI	33209.23	19.2%

Output #14 (partial):

STATE	SALES
IL	32083.22
IL	22223.12
IL	20338.12
IL	10332.11
MI	33209.23

Output #15 (partial):

REGION	TAX
312	516.6055
313	1604.161
313	1111.156
319	1016.906

Output #16 (The resulting SAS LOG- partial):

```
26
27          PROC SQL;
28            SELECT STATE,SALES, (SALES * .05) AS TAX
29            FROM USSALES
30            WHERE STATE IN ('OH','IN','IL') AND TAX > 10;
ERROR: THE FOLLOWING COLUMNS WERE NOT FOUND IN THE CONTRIBUTING TABLES: TAX.
NOTE: PROC SQL SET OPTION NOEXEC AND WILL CONTINUE TO CHECK THE SYNTAX OF STATEMENTS.
```

Output #17 (partial):

STATE	SALES	TAX
WI	10103.23	505.1615
WI	9103.23	455.1615
WI	15032.11	751.6055
IL	20338.12	1016.906

Output #18 (The resulting SAS LOG- partial):

```
31      PROC SQL;
32          SELECT STATE, STORE,SUM(SALES) AS TOTSALES
33          FROM USSALES
34          GROUP BY STATE
35          WHERE TOTSALES > 500;
35          WHERE TOTSALES > 500;
+
+            _____
             22
             202
ERROR 22-322: EXPECTING ONE OF THE FOLLOWING: (, **, *, /, +, -, ]], !!, <, <>, <=, ^=, >, >=,
?, =, ~=, CONTAINS, EQ, GE, GT, LE, LT, NE, &, AND, ], !, OR, ',', HAVING, ORDER.  THE STATEMENT
IS BEING IGNORED.

ERROR 202-322: THE OPTION OR PARAMETER IS NOT RECOGNIZED.
```

Output #19 (partial):

STATE	STORENO	TOTSALES
IL	31212	10332.11
IL	31373	22223.12
IL	31381	32083.22
IL	31983	20338.12
MI	33281	33209.23

Output #20:

STATE	TOTSALES
IL	84976.57
WI	34238.57

Output #21:

STATE	SALES
IL	20338.12
IL	10332.11
IL	32083.22
IL	22223.12

Output #22:

VAR1	VAR2	VAR1	VAR3
ABC	10	ABC	20
ABC	10	JKL	25
ABC	10	PQR	30
GHI	15	ABC	20
GHI	15	JKL	25
GHI	15	PQR	30
MNO	20	ABC	20
MNO	20	JKL	25
MNO	20	PQR	30

Output #23:

VAR1	VAR2	VAR1	VAR3
ABC	10	ABC	20

Output #24:

	EMPLOYEE			FEBSALES				BENEFITS			
OBS	FNAME	LNAME	STORENO	OBS	STATE	SALES	STORENO	OBS	FNAME	LNAME	CLAIMS
1	ANN	BECKER	33281	1	MI	31209.23	33281	1	ANN	BECKER	2003
2	CHRIS	DOBSON	33281	2	MI	15132.43	33312	2	CHRIS	DOBSON	100
3	EARL	FISHER	33281	3	IL	25338.12	31983	3	ALLEN	PARK	10392
4	ALLEN	PARK	31373	4	IL	26223.12	31373	4	BETTY	JOHNSON	3832
5	BETTY	JOHNSON	31373								
6	KAREN	ADAMS	31373								

FNAME	LNAME	CLAIMS	STORENO	STATE
ANN	BECKER	2003	33281	MI
ALLEN	PARK	10392	31373	IL
BETTY	JOHNSON	3832	31373	IL