Introduction to PROC SQL

Steven First, Systems Seminar Consultants, Madison, WI

ABSTRACT

PROC SQL is a powerful Base SAS Procedure that combines the functionality of DATA and PROC steps into a single step. PROC SQL can sort, summarize, subset, join (merge), and concatenate datasets, create new variables, and print the results or create a new table or view all in one step!

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

Write SQL code using various styles of the SELECT statement.
Dynamically create new variables on the SELECT statement.
Use CASE/WHEN clauses for conditionally processing the data.
Joining data from two or more data sets (like a MERGE!).
Concatenating query results together.

WHY LEARN PROC SQL?

PROC SQL can not only retrieve information without having to learn SAS syntax, but it can often do this with fewer and shorter statements than traditional SAS code. Additionally, SQL often uses fewer resources than conventional DATA and PROC steps. Further, the knowledge learned is transferable to other SQL packages.

AN EXAMPLE OF PROC SQL SYNTAX

Every PROC SQL query 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 be extracted from (similar to the SET statement). One advantage nof 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 SYNTAX

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

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

A SIMPLE PROC SQL

An asterisk on the SELECT statement will select all columns from the data set. By default a row will wrap when there is too much information to fit across the page. 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

To specify that only certain variables should appear on the report, the variables are listed and separated 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. Any of the DATA step functions can be used in an expression to create a new variable except LAG, DIF, and SOUND. Notice the commas separating the columns:

```
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)

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 within the same SELECT statement:

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

(see output #4 for results)

USING LABELS AND FORMATS

SAS-defined or user-defined formats can be used to improve the appearance of the body of a report. LABELs give the ability to define longer column headings:

```
TITLE 'REPORT OF THE U.S. SALES';
FOOTNOTE 'PREPARED BY THE MARKETING DEPT.';
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 #5 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 #6 for results)

The END is required when using the CASE. Coding the WHEN in descending order of probability will improve efficiency because SAS will stop checking the CASE conditions as soon as it finds the first true value.

ANOTHER CASE

The CASE statement has much of the same functionality as an IF statement. 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 #7 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 #8 for results)

Other summary functions available are the AVG/MEAN, COUNT/FREQ/N, MAX, MIN, NMISS, STD, SUM, and VAR.
This capability Is similar to PROC SUMMARY with a CLASS statement.

REMERGING

Remerging occurs 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 #9 for results)

REMERGING FOR TOTALS

Sometimes remerging is good, as in the case when the SELECT statement does not contain any other variables:

```
  PROC SQL;
    SELECT SUM(SALES) AS TOTSALES
    FROM USSALES;
  QUIT;
```
(see output #10 for results)

CALCULATING PERCENTAGE

Remerging can also be used to calculate percentages:

```
PROC SQL;
  SELECT STATE, SALES,
    (SALES/SUM(SALES)) AS PCTSALES
          FORMAT=PERCENT7.2
  FROM USSALES;
QUIT;
```
(see output #11 for results)

Check your output carefully when the remerging note appears in your log to determine if the results are what you expect.

SORTING THE DATA IN PROC SQL

The ORDER BY clause will return the data in sorted order: Much like PROC SORT, if the data is 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 may be more efficient when you need to sort on a dynamically created variable:

```
PROC SQL;
  SELECT STATE, SALES
  FROM USSALES
  ORDER BY STATE, SALES DESC;
QUIT;
```
(see output #12 for results)

SORT ON NEW COLUMN

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

```
PROC SQL;
  SELECT SUBSTR(STORENO,1,3)
    LABEL='REGION',
    (SALES * .05) AS TAX
  FROM USSALES
  ORDER BY 1 ASC, TAX DESC;
QUIT;
```
(see output #13 for results)

SUBSETTING USING THE WHERE

The WHERE statement will process a subset of data rows before they are processed:

```
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)

INCORRECT WHERE CLAUSE

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 #14 for results)

WHERE ON COMPUTED COLUMN

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 #15 for results)

SELECTION ON GROUP COLUMN

The WHERE clause cannot be used with the GROUP BY:

```
  PROC SQL;
    SELECT STATE, STORE,
      SUM(SALES) AS TOTSALES
    FROM USSALES
    GROUP BY STATE, STORE
    WHERE TOTSALES > 500;
  QUIT;
```
(see output #16 for results)

USE HAVING CLAUSE

In order to subset data when grouping is in effect, the HAVING clause 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 #17 for results)

HAVING WITHOUT A COMPUTED COLUMN

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 #18 for results)

CREATING NEW TABLES OR VIEWS

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 #19 for results)

The name given on the create statement can either be temporary or permanent. Only one table or view can be created by a 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 Cartesian 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 JANSALES, FEBSALES;
   QUIT;
```

(see output #20 for results)

INNER JOIN

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 U.STORENO, U.STATE,
         F.SALES AS FEBSALES
  FROM USSALES U, FEBSALES F
  WHERE U.STORENO=F.STORENO;
QUIT;
```

(see output #21 for results)

JOINING THREE OR MORE TABLES

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 #22 for dataset list and results)

CONCATENATING QUERY RESULTS

Query results can be concatenated with the UNION operator.
The UNION operator keeps only unique observations. To keep all observations, the UNION ALL operator can be used. Traditional SAS syntax would require the creation of multiple tables and then either a SET concatenation or a PROC APPEND. Again, the results can be achieved with one PROC SQL:

```
   PROC SQL;
     CREATE TABLE YTDSALES AS
     SELECT TRANCODE, STORENO, SALES
     FROM JANSALES

     UNION
     SELECT TRANCODE, STORENO,
            SALES * .99
     FROM FEBSALES;
   QUIT;

    (no output shown for this example)
```

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 often be more efficient 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

CONTACT INFORMATION

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

Steve First
Systems Seminar Consultants, Inc.
2997 Yarmouth Greenway Drive
Madison, WI 53711
Phone: (608) 278-9964
Fax: (608) 278-0065
Email: train@sys-seminar.com