Paper 70-27

An Introduction to SAS® PROC SQL

Timothy J Harrington, Venturi Partners Consulting, Waukegan, Illinois

Abstract

This paper introduces SAS® users with at least a basic understanding of SAS data sets to the SAS SQL Procedure. The subjects discussed are (1) extracting specific data items and observations from data sets and views (2) creating and organizing new tables and views (3) creating and grouping summary statistics (4) joining two or more data tables (5) assigning data to SAS Macro variables (6) comparing and contrasting PROC SQL with the SAS DATA Step and other SAS Procedures.

Creating output and new tables

The SQL Procedure is a SAS Procedure, but uses the ANSI standards. This procedure begins with the declaration PROC SQL; and ends with a QUIT; statement (not RUN;). In between these two declarations SQL code may be used. SQL code obeys the rules of the SAS system regarding maximum lengths of variable names, reserved words, and the use of the semi colon as the line delimiter. The simplest and most commonly used SQL statement is SELECT, which is used to extract data from a table. In this paper the term 'table' refers to either a SAS data set or a view. In the following simple example the contents of a table named VITALS are extracted and printed to the SAS output file.

```
PROC SQL;
  SELECT *
    FROM VITALS;
QUIT;
```

The '*' means select all of the variables in the table. By default the data selected is printed to the SAS output window or file. If the VITALS table contains this information:

OBS	PATIENT	DATE	PULSE	TEMP	BPS	BPD
1	101	25MAY01	72	98.5	130	88
2	101	01JUN01	75	98.6	133	92
3	101	08JUN01	74	98.5	136	90
4	102	30JUL01	81	99.0	141	93
5	102	06AUG01	77	98.7	144	97
6	102	13AUG01	78	98.7	142	93
7	103	24JUL01	77	98.3	137	79
8	103	31JUL01	77	98.5	133	74
9	103	07AUG01	78	98.6	140	80
10	103	14AUG01	75	99.2	147	89
11	104	22AUG01	72	98.8	128	83
12	104	29AUG01	69	99.1	131	86
13	104	05SEP01	71	98.9	127	82

This same table is printed to the next page of the output window, with the column names as headings but without the observation numbers. To select individual columns the column names must be specified, separated by commas. PROC SQL, like a SAS DATA step, is often used to create new tables, and this is done using the CREATE keyword. CREATE TABLE creates a data set and CREATE VIEW creates a view. The following example creates a table called 'BP' (Blood Pressure) and stores the Patient, Date, and Systolic and Diastolic Blood Pressure columns in that table.

```
PROC SQL;
  CREATE TABLE BP AS
    SELECT PATIENT, DATE, BPS, BPD
      FROM VITALS;
QUIT;
```

Note: References to table names in PROC SQL may be followed with SAS keyword expressions such as DROP, KEEP, RENAME, and WHERE. In the above example, if instead, all of the columns in VITALS *except* TEMP are needed in the new table this code can be used:

```
PROC SQL;
  CREATE TABLE BP AS
    SELECT *
      FROM VITALS(DROP=TEMP);
QUIT;
```

Duplicates and sorting

To select unique values of one or more columns the DISTINCT keyword is used. The following code produces a table of the unique patient numbers

```
PROC SQL;
  CREATE TABLE PATIDS AS
    SELECT DISTINCT PATIENT
      FROM VITALS;
QUIT;
```

Sorting a table in PROC SQL by the values of one or more columns (sort keys) is achieved using the ORDER BY clause. In this next example, the table VISITS would be the same as the VITALS table but the observations would be sorted by increasing PATIENT values and then by DATE in reverse order (most recent date first). Reverse order is specified by using the DESCENDING keyword, after the column to which it applies.

```
PROC SQL;
  CREATE TABLE VISITS AS
    SELECT PATIENT, DATE
      FROM VITALS
        ORDER BY PATIENT, DATE DESCENDING;
QUIT;
```

Sub-setting and Calculating

Just like in a SAS DATA step or any other SAS PROCedure the WHERE clause is used subset observations by one or more criteria. In the example which follows a new table called BPODD is created using observations from the VITALS table where the values of PATIENT are 101 and 103. To use values in columns to perform calculations and to store such calculated results in a new column the AS keyword is used in the SELECT statement. In this example the value of TEMP, the temperature in Fahrenheit, is converted to degrees Celsius and the result is stored in TEMPC. Also, column attributes such as FORMAT, LABEL, and LENGTH can be assigned to columns in a SELECT statement. In this example the format DATE7. is assigned to the column date in the new table. Attributes in the original table columns are left unchanged.

During the execution of this code the first event is the WHERE clause sub-sets the observations, then the calculations and column attribute changes are made. The output table is then created and then sorted as specified by the ORDER BY clause.

```
PROC SQL;
  CREATE TABLE BPODD AS
    SELECT PATIENT, DATE FORMAT=DATE7., BPS,
      BPD, (TEMP-32)/9*5 AS TEMPC
      FROM VITALS
        WHERE PATIENT IN (101 103)
          ORDER BY PATIENT, DATE DESCENDING;
QUIT;
```

```
PATIENT DATE     BPS  BPD  TEMPC

    101 08JUN01  136   90  36.944444444
    101 01JUN01  133   92  37
    101 25MAY01  130   88  36.944444444
    103 14AUG01  147   89  37.333333333
    103 07AUG01  140   80  37
    103 31JUL01  133   74  36.944444444
    103 24JUL01  137   79  36.833333333
```

An improvement which should be made to this table is to round the calculated result TEMPC to a predetermined number of decimal places. As in a SAS DATA step this is done using the ROUND function. Most SAS functions which can be used in a DATA step are also valid in PROC SQL code. For example the DATE column could be stored as a character string instead of a formatted numeric value by using the PUT function. In this case

DATE FORMAT=DATE7.

would be replaced with

PUT(DATE,DATE7.) AS CDATE LENGTH=7

CDATE would be a new column of character type and length 7 containing the formatted date as characters.

In the following example a table called MEANBP is created which contains the mean blood pressure values BPS and BPD as BPSMEAN and BPDMEAN respectively. The COUNT function calculates the total number of observations selected (in this case all observations because no WHERE clause is used) and the result is stored in the new column N. These values are the means for the entire VITALS table stored in a single observation. The ROUND function is used to save the results rounded to two decimal places.

```
PROC SQL;
  CREATE TABLE MEANBP AS
    SELECT COUNT(*) AS N, ROUND(MEAN(BPS),
      0.01) FORMAT=6.2 AS BPSMEAN,
      ROUND(MEAN(BPD),0.01) FORMAT=6.2 AS
        BPDMEAN
        FROM VITALS;
QUIT;
```

```
 N   BPSMEAN  BPDMEAN

13    136.08    86.62
```

Taking this a stage further, calculations can not only be performed on a whole table, but on groups of observations identified by key values. One or more groupings are specified using the GROUP BY clause. In this next example the column N is the number of observations for each occurrence of the same value of PATIENT. Similarly the column BPDHIGH is the maximum BPD value for each PATIENT. The column BPDPCT is the percentage of each value of BPD as compared with BPDHIGH for the same PATIENT. To avoid calculating the same value of BPDHIGH twice the keyword CALCULATED is used to identify the calculated value of BPDHIGH for the same observation. If the word CALCULATED was omitted a 'Column not found' error would result because BPDHIGH is not a column in the input table VITALS.

```
PROC SQL;
  CREATE TABLE HIGHBPP1 AS
    SELECT PATIENT, COUNT(PATIENT) AS N,
      DATE FORMAT=DATE7., BPD,
      MAX(BPD) AS BPDHIGH,
    ROUND(BPD/(CALCULATED BPDHIGH)*100, 0.01)
      FORMAT=6.2 AS BPDPCT
      FROM VITALS
        GROUP BY PATIENT;
QUIT;
```

```
PATIENT   N   DATE     BPD   BPDHIGH BPDPCT

    101   3   08JUN01   90        92  97.83
    101   3   25MAY01   88        92  95.65
    101   3   01JUN01   92        92 100.00
    102   3   30JUL01   93        97  95.88
    102   3   06AUG01   97        97 100.00
    102   3   13AUG01   93        97  95.88
    103   4   31JUL01   74        89  83.15
    103   4   07AUG01   80        89  89.89
    103   4   24JUL01   79        89  88.76
    103   4   14AUG01   89        89 100.00
    104   3   05SEP01   82        86  95.35
    104   3   29AUG01   86        86 100.00
    104   3   22AUG01   83        86  96.51
```

Assume now there is a requirement to create a table like this one but containing only the observations where the value of BPD is equal to BPDHIGH. A WHERE clause can be used to subset observations from the input table, but this keyword cannot be used with calculated values. Instead the HAVING keyword is used. HAVING is like a second WHERE clause, acting on the results of the WHERE and newly calculated values before producing the output table. In this next example the WHERE clause selects an observation from VITALS only if the value of PATIENT is 101, 102, or 103. The maximum value of BPHIGH is then calculated for each patient (the GROUP BY clause). The HAVING clause then compares the current value of BPD with the calculated maximum for the given group (PATIENT). If the HAVING condition is true the observation is output to HIGHBPP2. In this example if there were two or more values of BPD which equaled BPHIGH, all of these matching observations would be output. Finally the output data set is sorted in order of the increasing value of the calculated column BPDHIGH. The result is a table of one observation per patient and their maximum blood pressure reading, sorted in order of increasing maximum blood pressure value.

```
PROC SQL;
  CREATE TABLE HIGHBPP2 AS
    SELECT PATIENT, COUNT(PATIENT) AS N,
      DATE FORMAT=DATE7., MAX(BPD) AS BPDHIGH
      FROM VITALS
        WHERE PATIENT IN (101 102 103)
          GROUP BY PATIENT
            HAVING BPD = CALCULATED BPDHIGH
              ORDER BY CALCULATED BPDHIGH;
QUIT;
```

```
PATIENT   N   DATE     BPDHIGH

    103   4   14AUG01       89
    101   3   01JUN01       92
    102   3   06AUG01       97
```

To calculate values based on conditions on column values in the input table the CASE construct is used. This is like the IF THEN or the SELECT construct in a SAS DATA step. The general syntax of the CASE construct is

CASE <expression> WHEN <value 1> THEN <value A> WHEN <value 2> THEN <value B> …..
ELSE <value Z> END AS <result column>.

Values 1,2,3,… are values of the input 'expression' and values A,B,C… are the calculated values to be output into the result column. Value Z represents the result column value when none of the WHEN conditions are true. To ensure maximum run time efficiency the order of the WHEN values should be in decreasing likelihood of being true.

In this example even numbered patients are given a medication 'Med A' and odd numbered patients are given a medication 'Med B'. When the value of PATIENT is even the CASE expression has a Boolean result of 1 so the column DOSEGRP is assigned 'Med A'. When PATIENT is odd the assigned value for DOSEGRP is 'Med B'. If, for some reason, the value of PATIENT was missing the result of the expression would be 2, neither 0 nor 1 so the ELSE value 'Error' would be assigned to DOSEGRP.

```
PROC SQL;
   CREATE TABLE TESTMED AS
     SELECT PATIENT,
       CASE ((PATIENT/2 = INT(PATIENT/2)) +
         (PATIENT = .))
       WHEN 1 THEN 'Med A' WHEN 0 THEN
      'Med B' ELSE 'Error' END AS DOSEGRP
       LENGTH=5
       FROM VITALS
         ORDER BY PATIENT;
QUIT;
```

A second syntax for the CASE construct is:

CASE WHEN <expression 1> THEN <value A> WHEN <expression 2> THEN <value B> …..
ELSE <value Z> END AS <result column>.

In this construct the 'Expression' values are any valid SAS expression. Where two or more values of 'expression' are true the first output value is taken. If none of the values of 'Expression' are true the 'Value Z' result is taken.

Joining Tables

PROC SQL is very useful for joining tables by observation key values. SQL table joins usually execute faster than DATA step Merges, and SQL joins do not require the key columns (BY variables) to be sorted prior to the join. Another important difference between a SQL join and a DATA step Merge is when there are duplicate key values a complete Cartesian product is formed instead of just matching successive observations from each data

set. For this reason, 'many to many' SQL joins are valid.

The four types of table join are performed using the keywords ON and JOIN. The following examples use the VITALS table above and the following table named DOSING:

```
OBS PATIENT DATE    MED   DOSES AMT UNIT

1   102     30JUL01 Med A     2 1.2 mg
2   102     06AUG01 Med A     3 1.0 mg
3   102     13AUG01 Med A     2 1.2 mg
4   103     24JUL01 Med B     3 3.5 mg
5   103     31JUL01 Med B     3 3.5 mg
6   103     08AUG01 Med B     3 3.5 mg
7   104     22AUG01 Med A     2 1.5 mg
8   104     29AUG01 Med A     2 1.5 mg
9   104     05SEP01 Med A     2 1.5 mg
10  105     18JUN01 Med B     1 4.5 mg
11  105     25JUN01 Med B     2 3.0 mg
12  105     02JUL01 Med B     1 5.0 mg
```

The following code performs an 'inner' join on PATIENT and DATE, that is only observations with both these key values matching are selected.

```
PROC SQL;
  CREATE TABLE BOTH AS
    SELECT A.PATIENT,
      A.DATE FORMAT=DATE7. AS DATE, A.PULSE,
      B.MED, B.DOSES, B.AMT FORMAT=4.1
      FROM VITALS A INNER JOIN DOSING B
        ON (A.PATIENT = B.PATIENT) AND
          (A.DATE = B.DATE)
          ORDER BY PATIENT, DATE;
QUIT;
```

The two main points to note are the FROM and the ON clauses. The FROM clause contains both source tables and corresponding aliases, A for VITALS, and B for DOSING. The keywords INNER JOIN specify this type of join. The ON statement specifies one or more key columns (Akin to the BY variables in a DATA step Merge) and any related logical operators (In this case = and AND). Column attributes remain unchanged unless they are altered in the SELECT statement. If an attempt is made to output columns with the same name from more than one table an error results. In the SELECT clause the aliases denote which source table the data item is read from. This is the resulting table named BOTH:

```
PATIENT DATE    PULSE MED   DOSES  AMT

    102 30JUL01    81 Med A     2  1.2
    102 06AUG01    77 Med A     3  1.0
    102 13AUG01    78 Med A     2  1.2
    103 24JUL01    77 Med B     3  3.5
    103 31JUL01    77 Med B     3  3.5
    104 22AUG01    72 Med A     2  1.5
    104 29AUG01    69 Med A     2  1.5
    104 05SEP01    71 Med A     2  1.5
```

This example does not show duplicate key values, but if the join had been performed on PATIENT only (not on both PATIENT and DATE) there would have been a Cartesian product for each patient. In the case of Patient 102 the result would have been:

```
PATIENT DATE    PULSE MED   DOSES  AMT

    102 30JUL01    81 Med A     3  1.0
    102 30JUL01    81 Med A     2  1.2
    102 30JUL01    81 Med A     2  1.2
    102 06AUG01    77 Med A     2  1.2
    102 06AUG01    77 Med A     2  1.2
    102 06AUG01    77 Med A     3  1.0
    102 13AUG01    78 Med A     3  1.0
    102 13AUG01    78 Med A     2  1.2
    102 13AUG01    78 Med A     2  1.2
```

There would have been 12 observations for Patient 103 and 9 observations for Patient 104.

In this next example a 'left' join is being performed, again using the same tables and the same key values PATIENT and DATE.

```
PROC SQL;
  CREATE TABLE LEFT AS
    SELECT A.PATIENT,
      A.DATE FORMAT=DATE7. AS DATE, A.PULSE,
      B.MED, B.DOSES, B.AMT FORMAT=4.1
      FROM VITALS A LEFT JOIN DOSING B
        ON (A.PATIENT = B.PATIENT) AND
          (A.DATE = B.DATE)
          ORDER BY PATIENT, DATE;
QUIT;
```

The only difference between this and the 'inner' join is the keyword LEFT instead of INNER in the FROM statement. This time all observations are taken from the VITALS (A) table and only matching observations from the DOSING (B) table. Columns taken from the B table are substituted with missing values where a key match does not occur.

```
PATIENT  DATE    PULSE  MED   DOSES  AMT

    101  25MAY01    72            .    .
    101  01JUN01    75            .    .
    101  08JUN01    74            .    .
    102  30JUL01    81  Med A     2  1.2
    102  06AUG01    77  Med A     3  1.0
    102  13AUG01    78  Med A     2  1.2
    103  24JUL01    77  Med B     3  3.5
    103  31JUL01    77  Med B     3  3.5
    103  07AUG01    78            .    .
    103  14AUG01    75            .    .
    104  22AUG01    72  Med A     2  1.5
    104  29AUG01    69  Med A     2  1.5
    104  05SEP01    71  Med A     2  1.5
```

A 'left' join is not commutative and neither is a 'right' join a 'left' join with the tables reversed. A 'right' join considers the key values in the 'right' (B) table first, before performing the matching process. The

following code performs a ‘right’ join on the same two tables.

```
PROC SQL;
   CREATE TABLE RIGHT AS
     SELECT A.PATIENT,
       A.DATE FORMAT=DATE7. AS DATE, A.PULSE,
       B.MED, B.DOSES, B.AMT FORMAT=4.1
       FROM VITALS A RIGHT JOIN DOSING B
         ON (A.PATIENT = B.PATIENT) AND
           (A.DATE = B.DATE)
           ORDER BY PATIENT, DATE;
QUIT;
```

The resulting table is:

```
PATIENT DATE    PULSE MED   DOSES AMT

      .       .     . Med B     3 3.5
      .       .     . Med B     1 4.5
      .       .     . Med B     2 3.0
      .       .     . Med B     1 5.0
    102 30JUL01    81 Med A     2 1.2
    102 06AUG01    77 Med A     3 1.0
    102 13AUG01    78 Med A     2 1.2
    103 24JUL01    77 Med B     3 3.5
    103 31JUL01    77 Med B     3 3.5
    104 22AUG01    72 Med A     2 1.5
    104 29AUG01    69 Med A     2 1.5
    104 05SEP01    71 Med A     2 1.5
```

The missing values are where observations are present in DOSING (B) and have no matching observations in VITALS (A) (Patient 101).

The fourth and final join is the ‘full’ join, which is a combination of the ‘left’ and ‘right’ joins. The ‘full’ join, unlike the ‘inner’ join is not commutative.

```
PROC SQL;
  CREATE TABLE FULL AS
    SELECT A.PATIENT,
      A.DATE FORMAT=DATE7. AS DATE, A.PULSE,
      B.MED, B.DOSES, B.AMT FORMAT=4.1
      FROM VITALS A FULL JOIN DOSING B
        ON (A.PATIENT = B.PATIENT) AND
          (A.DATE = B.DATE)
          ORDER BY PATIENT, DATE;
QUIT;
```

The resulting table FULL looks like this:

```
PATIENT DATE    PULSE MED   DOSES AMT

      .       .     . Med B     3 3.5
      .       .     . Med B     1 5.0
      .       .     . Med B     1 4.5
      .       .     . Med B     2 3.0
    101 25MAY01    72           .   .
    101 01JUN01    75           .   .
    101 08JUN01    74           .   .
    102 30JUL01    81 Med A     2 1.2
    102 06AUG01    77 Med A     3 1.0
    102 13AUG01    78 Med A     2 1.2
    103 24JUL01    77 Med B     3 3.5
    103 31JUL01    77 Med B     3 3.5
    103 07AUG01    78           .   .
    103 14AUG01    75           .   .
    104 22AUG01    72 Med A     2 1.5
    104 29AUG01    69 Med A     2 1.5
    104 05SEP01    71 Med A     2 1.5
```

PROC SQL and SAS Macro language

The PROC SQL SELECT statement is also useful for copying the run time contents of columns into one or more macro variables. The macro variables are each preceded by a colon and all follow the keyword INTO. The following code selects the distinct list of patient numbers from the VITALS table into the macro variables PAT1, PAT2, PAT3…, PATn, where n is the number of distinct patient numbers. The excess macro variables are left blank. In this case the maximum number of values which can be handled for PATIENT is 999. The ORDER BY clause performs a sort before the data is written to the macro variables, so the lowest value of PATIENT is stored in PAT1, the next value in PAT2, and so on until the highest value is stored in PATn.

```
PROC SQL NOPRINT;
  SELECT DISTINCT PATIENT INTO :PAT1- :PAT999
    FROM VITALS
      ORDER BY PATIENT;
QUIT;
```

The NOPRINT statement disables output to the SAS output device. (When using a CREATE statement NOPRINT is the default).

A second method of storing run time data in macro variables is to write all of the data into one macro variable. This code creates a macro variable PATLIST and contains a sorted list of PATIENT numbers delimited by commas.

```
PROC SQL NOPRINT;
  SELECT DISTINCT PATIENT INTO :PATLIST
    SEPARATED BY ‘,’
    FROM VITALS
      ORDER BY PATIENT;
QUIT;
```

The SEPARATED BY keyword is used to specify a delimiter, which may be one or more characters contained in quotation marks. No delimiter is placed before the first item or after the last item.

Editing Observations

To add new observations to a table the INSERT command is used. The following code inserts two new observations into the VITALS table. The VALUES clause specifies the column values to be added. These values are in parenthesis and are separated by spaces (no commas), they must be of the same data types as the columns and in the same column order as in the table. A second method is to use the SET keyword followed by the column names and their newly assigned values delimited by commas. Any columns which are omitted in a VALUES or SET clause are set to missing. More than one VALUES or SET expression may appear between INSERT INTO and the next semicolon. The newly inserted observations are appended to the table. To maintain a key sequence the table must be resorted by any key columns using ORDER BY.

```
PROC SQL NOPRINT;
  INSERT INTO VITALS
    VALUES(102 '20AUG2001'd 75 98.4 122
      90);
  INSERT INTO VITALS
    SET PATIENT=102, DATE='27AUG2001'd,
      PULSE=77, TEMP=98.8, BPS=129,
      BPD=88;
QUIT;
```

To remove one or more observations from a table the DELETE command is specified, usually with a WHERE clause. If the WHERE clause is omitted *the entire table contents are deleted* and an empty table remains. The following code deletes all the observations for Patient 101.

```
PROC SQL NOPRINT;
  DELETE FROM VITALS
    WHERE PATIENT = 101;
QUIT;
```

To make changes to the values of one or more of the columns in one or multiple observations UPDATE is invoked. In this example Patient 103 on July 31 2001 had the PULSE and BPD columns values inadvertently transposed. This code corrects this error by updating that observation accordingly. The SET statement defines the columns to be changed and the WHERE clause identifies the observations to be changed. If more than one observation satisfies the WHERE condition all of those observations are updated with the new data in the SET statement.

```
PROC SQL NOPRINT;
  UPDATE VITALS
    SET PULSE=77, BPD=74
      WHERE PATIENT=103 AND
        DATE='31JUL2001'd;
QUIT;
```

Data Table Management

To change column attributes or to delete columns ALTER TABLE is specified. In this example the MODIFY keyword is used to change the DATE format and the DROP keyword removes the TEMP column from VITALS.

```
PROC SQL NOPRINT;
  ALTER TABLE VITALS
    MODIFY DATE FORMAT=MMDDYY8.
    DROP TEMP;
QUIT;
```

To delete a table DROP TABLE is specified. If the table is a view the original data is not changed, only the view itself is deleted.

```
PROC SQL NOPRINT;
  DROP TABLE BP;
QUIT;
```

To obtain run time information on table contents from within PROC SQL there is a system table called DICTIONARY.COLUMNS. This table contains information such as column names, their data types, labels, and formats. To retrieve observations about a particular table the WHERE clause should be used on the library (libname) and table (member) name columns. This code creates a table called VARINFO to hold the dictionary observations for the VITALS table, which is assumed to be in the WORK library.

```
PROC SQL;
  CREATE TABLE VARINFO AS
    SELECT *
      FROM DICTIONARY.COLUMNS
        WHERE UPCASE(LIBNAME) = 'WORK' AND
          UPCASE(MEMNAME) = 'VITALS';
QUIT;
```

Concluding Summary

One of the main advantages of PROC SQL over a DATA step is faster execution speed. Run time performance is improved still further by using indexes. PROC SQL is also very convenient for performing table joins compared to a DATA step merge. SQL calls can be nested and can be used to access external databases such as Oracle or DB2. SAS OPTIONS settings are valid within a PROC SQL – QUIT block. A DATA Step is better suited for sequential observation-by -observation processing.

Further information about PROC SQL is available from the SAS HELP menus and from the 'Guide to SAS PROC SQL' manual.