Ten Great Reasons to Learn SAS® Software's SQL Procedure

Kirk Paul Lafler, Software Intelligence Corporation

ABSTRACT

The SQL Procedure has so many great features for both end-users and programmers. It's fun, easy to learn and use, and can often result in fewer and shorter lines of code than using conventional DATA and PROC step methods.

This paper will present ten great reasons for learning the SQL procedure. Several examples will illustrate how PROC SQL can be used to retrieve data, subset and query data, order and group data, create and modify tables, perform statistical computations, produce great looking reports, construct views, join two or more tables, and exchange data between your favorite data base and the SAS System by using the SQL Pass-Through Facility.

INTRODUCTION

Structured Query Language (SQL) is a universal language that was originally developed to access data stored in relational databases or tables. Relations can be thought of as two-dimensional tables consisting of rows and columns. By using simple statements and option, you'll be able to define, manipulate, and control tables easily and quickly.

PROC SQL runs in both interactive and batch environments and is bundled in the base product of the SAS System. Using a comprehensive and powerful set of statements, expressions, and options, the Structured Query Language creates, modifies, and retrieves data from tables. Global statements such as TITLE and OPTIONS can be used with PROC SQL. Tables are accessed via a two-level name where the first level represents the libref and the second level represents the name of the table.

As with any language, there are some formalities, such as syntax, but it's not too overwhelming. Rather than spend time going over syntax rules, this paper concentrates on why you'll want to learn SQL. Then you can spend time reviewing the specific rules related to each statement on a "need-to-know" basis.

TEN GREAT REASONS TO LEARN PROC SQL

Over the years I have compiled my own list of PROC SQL favorites. They represent a powerful combination of features that I frequently use and can't be without. An added benefit is that these favorites rarely let me down. Basically, I never leave home without them (or at least when I'm off on some consulting assignment).

I want this list to be shared because I feel so strongly about its value and believe that there will be an immediate benefit for all that use it. At very least, these personal favorites will provide you with a "starting" point to begin your research into the exciting world of SQL. (I hope you'll forgive the sales pitch.) Drum role please. . .

Top 10 List

 # 10. Retrieving (Querying) Data

To extract and retrieve data, the SQL Procedure's SELECT statement is used. The SELECT statement tells the SQL processor what column(s) to extract and what table(s) to use. In the first example, a wildcard '*' (asterisk) is used to list all columns from the PATIENTS table.

```
PROC SQL ;
  SELECT   *
    FROM  libref.PATIENTS ;
QUIT ;
```

In the next example, specific columns (variables) are displayed in the order indicated in the SELECT statement.

```
PROC SQL ;
  SELECT   SSN,  GENDER
         FROM  libref.PATIENTS ;
QUIT ;
```

In this example, the columns SSN (Social Security Number) and GENDER from the PATIENTS table are selected and displayed. Notice the comma ',' between the columns SSN and GENDER. The comma separates one column from another.

 # 9. Subsetting Data

Subsetting data is easy in SQL. Just specify a WHERE clause as part of the SELECT statement. The WHERE clause tells the SQL processor to extract only those rows (observations) that satisfy the specified criteria. Rows not meeting this criteria are automatically excluded and not selected.

```
PROC SQL ;
  SELECT   GENDER,  SSN,  WEIGHT
          FROM  libref.PATIENTS
              WHERE  GENDER = 'M' ;
QUIT ;
```

In this example, only rows having a value of 'M' (Male) in the GENDER column are selected. Once selected, the rows and columns GENDER, SSN, and WEIGHT from the PATIENTS table are displayed. The original table is not altered, since this is a ***"read-only"*** operation.

 # 8. Ordering Data

There will be times when the data needs to first be arranged in some order before being sent to output. The ORDER BY clause of the SELECT statement can sort one or more columns in ascending or descending order. As with any sorting procedure, there are certain performance costs associated with using the ORDER BY clause (so avoid unnecessary sorting). Ascending is the default sort order and causes the data to be sorted from the lowest to the highest value.

To order data from the highest to lowest value for a specific column, use the ***DESC*** (descending) option after the column name it is meant for.

```
PROC SQL ;
  SELECT   GENDER,  SSN,  WEIGHT
          FROM  libref.PATIENTS
              ORDER BY WEIGHT ;
QUIT ;
```

In this example, data is arranged in ascending order (lowest to highest) for the column of values associated with WEIGHT from the PATIENTS table.

In the next example, data is arranged in descending order by specifying the DESC option. ***(Note the difference in the placement of the DESC option in SQL versus in the SORT procedure.)***

```
PROC SQL ;
  SELECT   GENDER,  SSN,  WEIGHT
          FROM  libref.PATIENTS
              ORDER BY WEIGHT DESC ;
QUIT ;
```

 # 7. Grouping Data

At times it may be necessary to display data in designated groups. The GROUP BY clause used in the SELECT statement identifies column values and groups the rows of data together. When one or more summary functions are specified in a query-expression, a GROUP BY clause is used.

In the next example, the patient's weight is summed and then grouped according to their gender.

```
PROC SQL ;
  SELECT REGION, SUM(WEIGHT)
    FROM libref.PATIENTS
      GROUP BY GENDER ;
QUIT ;
```

 # 6. Creating and Modifying Tables

Entire books have been written on the subject of planning, creating, and modifying tables, so it is doubtful that much can be said in such a short piece. Still, a few examples will illustrate the capabilities of the SQL procedure.

There are three methods to create a table:

- Column-Definition List.
- Using the LIKE Clause.
- Using Tables to Derive new Tables.

While each method results in the creation of a table, one method stands out as the front-runner (and is the approach I use most frequently). Using tables to derive new tables creates a new table containing column names, column attributes (data type, length, format), and rows of data from one or more existing tables.

```
PROC SQL ;
  CREATE TABLE  libref.FEMALES  AS
    SELECT SSN, GENDER, DOB
      FROM libref.PATIENTS
        WHERE GENDER  EQ  'F' ;
QUIT ;
```

In this example, a new table called FEMALES is created and populated from the PATIENTS table with rows of data having a value of 'F' (female) in the GENDER column.

Occasionally a table requires one or more changes. When a table is changed or altered, the structure also changes. Columns can be added, renamed, deleted, or modified. Should one or more columns be dropped, any data within that column is also lost.

Columns definitions (length, informat, format, and length) can be modified using the MODIFY clause. The RENAME= SAS data set option must be used to rename column names (no other method exists). Altering the attributes of a column that contains an associated index (simple or composite), does NOT prohibit the values in the altered column from using the index. But, should a column that contains an index be dropped, then consequently the index is also dropped.

The following example illustrates how a new column called SALARY can be added to an existing table.

```
PROC SQL ;
  ALTER TABLE libref.PATIENTS
    ADD SALARY NUM FORMAT=8.
            LABEL='Patient"s Salary' ;
QUIT ;
```

The ALTER TABLE statement tells the SQL processor which table is to be changed. Once a column exists in the table structure, it can then be modified using the ALTER TABLE statement with MODIFY as illustrated in the next example.

```
PROC SQL ;
  ALTER TABLE  libref.PATIENTS
    MODIFY SALARY NUM
              FORMAT = DOLLAR12.2 ;
QUIT ;
```

In this example, the column format related to SALARY is modified to display values using the DOLLAR12.2 template.

Deleting one or more rows of data is accomplished with the DELETE statement. Use extreme care when using the DELETE statement to delete rows of data, since a DELETE statement without a corresponding WHERE clause will delete all rows of data in the table.

```
PROC SQL ;
  DELETE  FROM  libref.PATIENTS
     WHERE  SALARY = . ;
QUIT ;
```

 # 5. Using Summary Functions

Summary functions produce a value from a group of values. Summaries can be specified for all the values in each row or column in a table. Groups can be specified by using a GROUP BY clause. In the event this clause is not specified, each and every row in the table is treated as a single group.

Several popular summary functions are AVG, COUNT, MIN, MAX, RANGE, and SUM. The following example illustrates how the MAX function is used. This query will determine the maximum value from the group.

```
PROC SQL ;
  SELECT SSN, GENDER, MAX(DOB)
    FROM libref.PATIENTS ;
QUIT ;
```

The next example illustrates combining a summary function with a GROUP BY clause to form a group.

```
PROC SQL ;
  SELECT SSN, GENDER, MAX(DOB)
    FROM libref.PATIENTS
      GROUP BY GENDER ;
QUIT ;
```

 # 4. Producing Reports

The SQL procedure produces ***"quick and dirty"*** reports similar to the PRINT procedure, except column headings are automatically underlined. A maximum of 10 titles and/or footnotes can be added as with other output producing procedures. User-defined formats can also be specified to enhance the output's appearance.

```
PROC SQL ;
  TITLE1  'Sample SQL Report' ;
  SELECT LASTNAME, GENDER
    FROM libref.PATIENTS
      ORDER BY LASTNAME ;
QUIT ;
```

Output from this query is displayed below.

Sample SQL Report

LASTNAME	GENDER
ADAMS	M
CRANBERRY	M
JONES	F
PETERSON	M
SMITH	F

 # 3. Creating and Using Views

Views behave in many ways as a table, even though they are not a table. A table stores data while a view stores only instructions. Since a view is only a set of instruction, they cannot be updated as tables can. Consequently, views are "read-only".

An internal table is constructed whenever a view is referenced by another SQL statement, DATA step, or PROC step. Once the internal table is constructed, it is then processed by the SQL processor.

A view always retrieves data from the most current underlying table(s). Consequently, views don't suffer from out-of-date information as other methods do. They provide a way to maintain a high level of data integrity while reducing data redundancy.

Creating and using views are accomplished in a two-step process. First, the view is created. Once it's created, it can be used. The following example illustrates creating a view with the CREATE VIEW statement.

```
PROC SQL ;
  CREATE VIEW libref.SENIOR  AS
    SELECT LASTNAME, AGE
      FROM libref.PATIENTS
        WHERE AGE > 65 ;
QUIT ;
```

In this example, the view is created as a stored set of instructions. No data is extracted, processed, or displayed. Once a view is successfully created, a message appears on the SAS System log informing you that the view was successfully created.

```
Note:  View libref.SENIOR  has been defined.
```

Using a view to display data is accomplished by referencing the view by its store name. Views can be referenced in many ways. The most common is through an SQL query. Other methods are available and can be used, such as in a PROC or DATA step.

The next example illustrates using a view within an SQL query. The view name is libref.SENIOR in this example.

```
PROC SQL ;
  SELECT  *  FROM libref.SENIOR ;
QUIT ;
```

Views can also be referenced or used in other SAS Procedures. The next example illustrates calling a view using PROC MEANS.

```
PROC MEANS DATA=libref.SENIOR ;
RUN ;
```

The next example shows a view being used by the CONTENTS procedure.

```
PROC CONTENTS DATA=libref.SENIOR;
RUN ;
```

The output generated from executing the CONTENTS procedure under an MVS environment is illustrated next.

```
                        CONTENTS PROCEDURE

Data Set Name:  libref.SENIOR          Observations:          .
Member Type:    VIEW                   Variables:             2
Engine:         SQLVIEW                Indexes:               0
Created:        07SEP97:07:07:07       Observation Len:       18
Last Modified:  07SEP97:07:07:07       Deleted Obs:           0
Data Set Type:                         Compressed:            NO
Label:

-----Alphabetic List of Variables and Attributes-----
#   Variable        Type      Len       Pos       Format
-----------------------------------------------------------------------
2   AGE             Num         3       15
1   LASTNAME        Char       15        0
```

Occasionally, the statements of a view definition may need to be seen. The DESCRIBE statement provides a way to see the view as it was originally defined. The next example illustrates describing a view definition.

```
PROC SQL ;
  DESCRIBE VIEW  libref.SENIOR ;
QUIT ;
```

The next example shows how a view definition can be deleted when no longer needed.

```
PROC SQL ;
  DROP VIEW  libref.SENIOR ;
QUIT ;
```

 # 2. Joining Tables of Data

A join of two or more tables provides a means of gathering and manipulating data in a single SELECT statement. A "JOIN" statement does not exist in the SQL language. The way two or more tables are joined is to specify the tables names in a WHERE clause of a SELECT statement. A maximum of 16 tables can be joined.

Joins are specified on a minimum of two tables at a time, where a column from each table is used for the purpose of connecting the two tables. Connecting columns should have ***"like"*** values and the same datatype attributes since the join's success is dependent on these values.

The next example illustrates a simple two-way join.

```
PROC SQL ;
  SELECT *
    FROM libref.PATIENTS, libref.COSTS
      WHERE PATIENTS.SSN =
            COSTS.SSN ;
QUIT ;
```

In this example, tables PATIENTS and COSTS are used. Each table has a common column, SSN which is used to connect rows together from the tables when the value of SSN are equal (match), as specified in the WHERE clause. Note the use of a comma to separate each table in the FROM clause.

It is worth noting, that if the WHERE clause in the preceding example was not specified, all possible combinations of rows from each table would have been joined. This is known as the ***Cartesian Product***. Say for example you joined two tables, table A containing 100 rows and table B also containing 100 rows. The result of this combination would be the Cartesian product of 10,000 rows. Very rarely is there a need to perform a join operation in SQL when a WHERE clause isn't specified.

 # 1. Using the Pass-Through Facility

The SQL Pass-Through facility is a SAS software product that allows DBMS-specific SQL statements to be routed to a database management system (of choice) to enable direct data retrieval. It accomplishes this by using the appropriate SAS/ACCESS interface.

The advantages of using SQL Pass-Through are:

- Ability to retrieve data directly from the host database management system
- Uses the services of the host database management system
- The host DBMS optimize queries to achieve maximum performance.

The following statements are used with the SQL Pass-Through Facility:

- CONNECT - establishes a connection with host DBMS
- EXECUTE - sends dynamic host DBMS-specific statements to DBMS
- DISCONNECT - terminates the connection with the host DBMS.

A few examples will illustrate how easy it is to extract information from a host DBMS. The next example connects to a SYBASE system and assigns the alias SYB1 to it. Since SYBASE is a case sensitive database, values (e.g., SYB1) are typed in uppercase.

```
PROC SQL;
  CONNECT TO SYBASE AS SYB1
    (SERVER=serve1
      DATABASE= employee
      USER=userid
      PASSWORD=password) ;
  %PUT &SQLXMSG ;
QUIT ;
```

The next example sends an ORACLE SQL query to an ORACLE database for processing. MYORAC is used as the connection alias.

```
PROC SQL ;
  CONNECT TO ORACLE AS MYORAC
    (USER=ABC
     ORAPW=PASSWORD
     PATH="@ORAC77") ;
  %PUT &SQLXMSG ;
  SELECT *
    FROM CONNECTION TO MYORAC
        (SELECT SSN, EMPID,
                LASTNAME, SEX, AGE
            FROM EMPLOYEES
                WHERE AGE > 65) ;
  %PUT &SQLXMSG ;
  DISCONNECT FROM MYORAC ;
QUIT ;
```

CONCLUSION

The SQL procedure provides a powerful and exciting way to retrieve, analyze, manipulate, and display data. It's easy to learn and use. More importantly, since the SQL procedure follows ANSI (American National Standards Institute) guidelines, your knowledge is portable to other platforms and vendor's SQL implementations.

The SQL procedure offers a comprehensive and powerful set of tools. Learning and using SQL could be your ticket to an enjoyable and rewarding experience. Set aside two hours a week to learn and use SQL. I think you'll be surprised at how effortlessly you'll become a pro. Work hard, enjoy, and be rewarded!

ACKNOWLEDGEMENTS

The author would like to thank SUGI 23 Conference leaders including Conference Chair Sally Goostrey, and Section Chairs Frank Fry and David Wilhite for inviting me to present and for their support and encouragement during the development of this paper.

CONTACT INFORMATION

The author welcomes any and all comments and suggestions. He can be reached at:

Kirk Paul Lafler
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, CA 91979-1390
Tel: (619) 670-SOFT or (619) 670-7638
Email: KirkLafler@CompuServe.com