Introduction to Using PROC SQL

Thomas J. Winn Jr., Texas State Comptroller's Office, Austin, Texas

ABSTRACT

This tutorial presentation will explain the basic syntax of the SQL Procedure. PROC SQL is the SAS ® System's implementation of Structured Query Language, which is used for retrieving and updating data in relational tables and databases. PROC SQL also has substantial data manipulation and summarization capabilities. PROC SQL statements could be used to replace much traditional SAS code (DATA step, PROC SORT, and PROC MEANS steps), resulting in less programming time and greater computer efficiency. Most of the illustrative examples in this presentation pertain to DB2 tables which are used for tax administration purposes at the author's agency. The principles underlying each example should transfer readily to other settings.

FIRST STEPS TOWARD UNDERSTANDING PROC SQL

Relational Database Management Systems

A *relational database* is an organized collection of information in which the data are arranged into a collection of two-dimensional *tables*, each containing one or more rows and columns, and conforming to certain rules. The data are *logically related*, based upon their *values*, and not according to some other data structure.

Information in a relational database management system is managed using a specific language comprised of a small but elegant set of operators. This language is called *Structured Query Language (SQL)*.

Sometimes, I may refer to SAS data sets (SAS data files and SAS data views) as "tables", to SAS variables as "columns", and to observations as "rows". Strictly speaking, these aren't exactly the same things (there exist tables which aren't SAS data sets, and views contain no data themselves) but the underlying concepts are certainly comparable, at least. If most of your work mostly involves SAS data files, then whenever I refer to a "table", you may substitute "SAS data file; and when I write "column", substitute "SAS variable"; and when I say "row", substitute "SAS observation".

Structured Query Language (SQL)

SQL is a language that talks to a relational database management system. It is a standard. There are many implementations of SQL For example, each RDBMS may use its own particular "dialect" of SQL. Many programmers admire SQL because it is compact and powerful. However, these desirable characteristics also make SQL a little bit tricky to use. SQL supports all of the standard operations one might need to manage a database. Among other provisions, for example, SQL allows such familiar actions as CREATE, SELECT, INSERT, UPDATE, DELETE, and DROP.

The theoretical foundation for relational database management systems is mathematical set theory. This well-defined foundation makes it possible to focus on the logical structure of the data, rather than on the underlying physical layout of the database.

Unlike many other data base programming languages, SQL is *non-procedural*. SQL is a high-level language. SQL is used to inform the DBMS about *what* data are needed (the desired end result), but not about a particular method (the *where* and the *how*) for dealing with the data. The access method specifics are internal to the DBMS. The query optimizer chooses an efficient strategy for achieving the specified objective.

The SAS SQL Procedure

SAS has an implementation of Structured Query Language called PROC SQL. PROC SQL follows most of the guidelines set by the American National Standards Institute (ANSI) in its implementation of SQL. However, it is not fully compliant with the ANSI-standard for SQL. PROC SQL includes several enhancements, which *exceed* the ANSI specifications, for greater compatibility with other elements of the SAS System.

The SQL Procedure processes SQL statements that read and update tables. PROC SQL uses Structured Query Language to:

- retrieve and manipulate SAS data sets,
- create and delete data sets,
- add or modify data values in a data set,
- add, modify, or drop columns in a data set,
- create and delete indexes on columns in a data set.

PROC SQL can be used on SAS files, flat files, VSAM files, database tables, and combinations of these to do query operations, and also to perform many of the ordinary data manipulation and reporting operations customarily accomplished using DATA and PROC steps.

PROC SQL processes SQL statements that read and update tables. PROC SQL uses SQL to create, modify, and retrieve data from tables and views (and SAS data sets). PROC SQL can be used in batch programs or during an interactive SAS session. PROC SQL can perform many of the operations provided by the DATA step, and the PRINT, SORT, MEANS and SUMMARY procedures.

Syntax for the SAS SQL Procedure

The SQL Procedure includes several statements, not all of which are always required. SQL itself is made up of modular components, and PROC SQL includes statements and clauses which reflect those components.

Here is the basic syntax:

```
PROC SQL < option < option > ...
    ALTER alter-statement;
    CREATE create-statement;
    DELETE delete-statement;
```

```
DESCRIBE VIEW view-name;
DROP drop-statement;
INSERT insert-statement;
RESET < option < option > ... >;
SELECT select-statement;
UPDATE update-statement;
VALIDATE query-expression;
```

We're planning to discuss only the most commonly-used statements for the SQL Procedure here. For a more complete explanation, please consult the appropriate reference manuals.

USING PROC SQL TO SELECT DATA

Queries, Views, and Result Sets

A *view* is a stored specification of a database request. A view is a description of selected data from one table, or from several tables. It may be helpful to regard a view as a *virtual table*.

A *query* is a request to retrieve some data from a database table or view. A query may be a simple question about the information which is in a single table, or it may be a complex question about the information from several tables.

A *result set* is what you get back when you query a database table or view. A result set also is a table.

PROC SQL Syntax for Simple Queries

One of the most common uses for PROC SQL is to provide a query to one or more SAS Data Files or SAS Data Views. This is accomplished by means of a SELECT statement. Here is a simple example:

```
PROC SQL ;
   SELECT AUDID,  ZIPSERVE
   FROM MYLIB.ZIPAUDOF
   WHERE AUDID='2I80'
   ORDER BY ZIPSERVE ;
```

In the preceding example, the SELECT statement specifies the column-names in a particular table from which the data are to be chosen, it further subsets these data according to a certain value contained in some of the rows, and then it identifies the column to be used as the basis for re-sequencing the extracted data for the printed report.

A general form for the SELECT statement is the following:

```
SELECT column-1, column-2, ...
   FROM table-a, table-b, ...
   WHERE expression
   GROUP BY column-i, column-j, ...
   HAVING expression
   ORDER BY column-r, column-s, ... ;
```

Here is another example of the use of the SELECT statement:

```
PROC SQL ;
SELECT TPNUM,  TAXID, CMPY_TY,
        CMPY_LIC
   FROM MYLIB.TAXRESP
   WHERE TAXID=71
   AND CMPY_TY IN (40,41,45,56)
   ORDER BY TPNUM ;
```

The VALIDATE statement can be used to check the validity of a query expression, without executing the query:

```
PROC SQL ;
  VALIDATE
     SELECT TPNUM, TAXID,
            CMPY_TY, CMPY_LIC
     FROM MYLIB.TAXRESP
     WHERE TAXID=71
        AND CMPY_TY IN (40,41,45,56)
     ORDER BY TPNUM ;
```

Notice that this example is just a SELECT expression preceded by the VALIDATE keyword. This would generate the printing of a message in the SAS Log regarding whether or not the PROC SQL SELECT statement has correct syntax.

An asterisk (*) in the SELECT statement of a query results in the selection of all of the columns in the specified table.

```
PROC SQL ;
   SELECT * FROM MASTER.TAXTYPE1;
```

Often, we need to create new variables (temporary columns) whose values are derived from existing columns. SAS DATA step functions can be used to calculate values for temporary columns. The AS keyword is used to specify a column *alias* for such new columns.

```
PROC SQL ;
  SELECT TPNUM, TAXID,
     INT((TODAY()-FRSBUSDT)/365.25)
        AS BUSYRS
   FROM MYLIB.PERMIT ;
```

Use a WHERE clause to specify a condition that the data must satisfy in order to be selected. As you might expect, one can use any of the customary comparison operators in a WHERE clause.

```
PROC SQL ;
   SELECT  TPNUM,  TAXID,  PDOENDDT,
           DTLTY, PYRNETPM
    FROM  MYLIB.PAYREV1
    WHERE  TAXID=21  AND  DTLTY='P' ;
```

In addition to the common comparison operators (LT, <, GT, >, EQ, =, LE, <=, GE, >=, NE, ^=, ~=) and the IN-operator, there also are some special operators that can be used in a WHERE clause:

- CONTAINS or ? -- selects rows that include a specified string,
- IS NULL or IS MISSING -- selects rows for which the value of a particular column is missing,
- BETWEEN - AND -- selects rows in which the value of the column falls within a range of values, *inclusive* of the end points,
- LIKE -- selects rows by comparing character values to specified patterns [a percent sign (%) replaces any number of characters, and an underscore (_) replaces only one character],
- =* -- selects rows that contain a spelling variation of the specified word (this is a "sounds like" operator).

```
PROC SQL ;
   SELECT  TPNUM,  TAXID,  PDOENDDT,
           DTLTY, PYRNETPM
   FROM  MYLIB.PAYREV1
   WHERE  TAXID=21  AND  DTLTY='P'
      AND PDOENDDT BETWEEN
          '01SEP95'D  AND '31AUG96'D ;
```

Consult a reference manual for additional examples of the use of operators in a WHERE clause.

To eliminate duplicate rows returned from a query, precede a column name with the keyword DISTINCT. Then, one row would be displayed for each unique combination of values. For example,

```
PROC SQL;
  SELECT DISTINCT TPNUM, TAXID
    FROM  MYLIB.PERMIT;
```

The CALCULATED keyword may be used to refer to a temporary column, which was previously specified by an expression in the SELECT clause.

```
PROC SQL ;
  SELECT  TPNUM,  TAXID,  PERIOD,
     GRSALS - DEDUCTS + PURCHS
        AS AMTSUBJ
  FROM  MYLIB.REPTAMT
  WHERE  TAXID=2
     AND CALCULATED AMTSUBJ > 5000;
```

Use an ORDER BY clause to return the results of a query in ascending (the default), or in descending order, relative to the values in specified columns.

```
PROC SQL ;
   SELECT TPNUM,  TAXID, CMPY_TY, CMPY_LIC
    FROM MYLIB.TAXRESP
    WHERE TAXID=71
    ORDER BY TPNUM ;

PROC SQL ;
   SELECT  AUDID, ZIPSERVE
   FROM  MYLIB.ZIPAUDOF
   WHERE  AUDID='2I80'
   ORDER BY  ZIPSERVE  DESC ;
```

You can customize the printed reports produced from queries by specifying SAS formats and/or labels to be used. Here is an example:

```
PROC SQL ;
  SELECT TPNUM
           LABEL='TAXPAYER NUMBER',
      TAXID  LABEL='TAX CODE',
      PDOENDDT
           LABEL='PERIOD END DATE'
           FORMAT=MMDDYY8.,
      DTLTY  LABEL='RECORD TYPE',
      PYRNETPM LABEL='NET PAYMENT'
                FORMAT=DOLLAR12.2
   FROM  MYLIB.PAYREV1
   WHERE  TAXID=21  AND  DTLTY='P' ;
```

The SQL Procedure provides several functions for summarizing the rows in a column. As an accommodation to programmers who are familiar with either SAS or SQL, some of these functions have multiple names.

- COUNT, FREQ, N -- number of (non-missing) values,
- NMISS -- number of missing values,
- MAX – maximum (largest) value,
- MIN – minimum (smallest) value,
- SUM -- sum of values,
- AVG, MEAN – arithmetic average value,
- STD -- standard deviation of values,
- VAR – variance of values,
- STDERR – standard error of the mean.

Summary functions calculate statistics based on the entire table.

```
PROC SQL ;
  SELECT  SUM(PYRNETPM) AS TOTAL
    FROM  MYLIB.PAYREV1
    WHERE  TAXID=71  AND  DTLTY='P'
       AND PDOENDDT BETWEEN
       '01SEP94'D AND '31AUG95'D ;
```

If more than one column-name is specified in a summary function, then the summary function operates like a DATA step function, in which the calculation is performed for each row. Whenever the SELECT clause includes a summary function, and at least one other column-name, then, after the calculation is performed, the result of the calculation is re-merged with each of the selected values from the table.

Let us suppose that a display or analysis needs to be performed, not on the table as a whole, but for each of the various classification groups defined by the values occurring in a particular column. In this case, a GROUP BY clause can be used to separate the data into groups based upon the distinct values in a column, or to generate summary function statistics for each of the distinct values in the grouping column.

```
PROC SQL ;
   SELECT  TAXID,
           SUM(PYRNETPM) AS TOTAL
      FROM  MYLIB.PAYREV1
      WHERE  DTLTY='P'
         AND PDOENDDT BETWEEN
             '01SEP94'D AND '31AUG95'D
      GROUP BY TAXID ;
```

We have previously discussed how to use a WHERE clause to select data based on values for individual rows. If one wanted to specify a condition (involving a summary function) that each group in a query would have to satisfy, then the HAVING clause would be used.

```
PROC SQL ;
   SELECT  TAXID, SUM(PYRNETPM) AS TOTAL
      FROM  MYLIB.PAYREV1
      WHERE  DTLTY='P'
         AND PDOENDDT BETWEEN
             '01SEP94'D AND '31AUG95'D
      GROUP BY TAXID
      HAVING SUM(PYRNETPM) > 500000 ;
```

The GROUP BY clause must precede the HAVING clause. The HAVING clause contains an expression which includes a summary function. The result set would include only those groups of data which satisfy the condition specified in the HAVING clause.

PROC SQL Syntax for Subqueries (Nested Queries)

It is possible to nest queries inside other queries. Nested queries, also called subqueries (or inner queries), select rows from one table based on values in another table. A subquery is a query-expression that is nested as part of another query-expression. A subquery (the inner query, which is enclosed in parentheses) is evaluated before the outer query. The result set from the inner query is used as the domain for the outer query. The subquery can be against a different table than the outer query. If more than one subquery is included, the innermost query is evaluated first, then the next innermost query, and so forth, moving outward through each level of nesting.

Subqueries usually involve a WHERE or HAVING clause which contains its own SELECT clause, and which is enclosed in parentheses. Here is an example of a subquery:

```
PROC SQL ;
   SELECT  DISTINCT  TPNUM, CMPY_TY,
                     CMPY_LIC
      FROM MYLIB.TAXRESP
      WHERE  TPNUM  IN
         (SELECT TPNUM
            FROM MYLIB.TAXSTAT
            WHERE TAXID=71
               AND ENDSTADT=. )
      ORDER BY TPNUM ;
```

A subquery that depends upon values returned by the outer query is called a correlated subquery. Here is an example:

```
PROC SQL ;
   SELECT  CMPY_LIC
      FROM MYLIB.TAXRESP AS T
      WHERE  71  IN
         (SELECT TAXID
            FROM MYLIB.TAXSTAT AS S
            WHERE  ENDSTADT=.
              AND  S.TPNUM=T.TPNUM)
      ORDER BY CMPY_LIC ;
```

Observe that in this type of subquery, the WHERE expression in the inner query refers to values in a table in the outer query. The correlated subquery Is evaluated for each row in the outer query. Fortunately, correlated subqueries are encountered much less frequently than ordinary nested queries. Often, one is able to find another way to code this type of data request.

USING PROC SQL TO COMBINE DATA

Combining Data from Tables

There are two major ways of combining data from tables: we use *join operations* to combine data from tables in a horizontal, or side-by-side, manner, using a key value; and we use *set operations* to combine data from tables vertically -- that is, concatenating the information by stacking the data from one table on top of the data from another table.

Join Operations

Joins combine information from multiple tables by matching rows that have common values in columns which relate the tables. Data from the tables are combined horizontally (i.e., in a side-by-side manner) using a key value. Tables do not have to be sorted before they are joined. Joining tables is similar to, though not the same as, merging SAS data sets. There are different kinds of joins.

- Inner Joins (conventional joins) retrieve rows with matching key values. Inner joins can be performed on up to 16 tables in one query.
- Outer Joins retrieve rows with matching key values, plus all non-matching rows from the left, both, or right tables. Outer joins can be performed on only two tables at a time.

PROC SQL Syntax for Joining Tables

The fundamental type of horizontal synthesis of data from two tables contains all combinations of the rows from both tables. This result set is called the *Cartesian Product* of two tables. It is obtained by combining each row of the first table with each row of the second table. Here is the syntax:

```
/* CARTESIAN PRODUCT */
PROC SQL;
  SELECT * FROM FIRST, SECOND;
```

In the preceding example, the FROM clause identifies more than one table name as sources of data for the query – this tells us that some type of a join operation is being performed. Since, in this case, there are no additional conditions to be satisfied, we recognize that the desired result set is the Cartesian Product.

Most of the time, we're not interested in obtaining *all* of the possible combinations of the rows. Usually, we want our result to include only those rows which have common values in certain columns (the keys) which relate the tables to each other, so we include a WHERE statement which specifies the key values where matches are sought. This situation describes an *inner join* (or a *conventional join*). The following example is for a conventional join result set, which includes only those rows from the Cartesian Product which have matching key values. The syntax for retrieving row combinations having matching key values is:

```
/* INNER JOIN OF TABLES */
PROC SQL;
  SELECT *
     FROM FIRST, SECOND
     WHERE FIRST.X=SECOND.X;
```

As mentioned previously, the result set for an *outer join* would include all rows from the Cartesian Product with matching key values, plus all non-matching rows from the left, both, or right tables. Typical syntax for a left outer join would be:

```
/* LEFT OUTER JOIN OF TABLES */
PROC SQL;
   SELECT *
      FROM FIRST LEFT JOIN SECOND
      ON FIRST.X=SECOND.X;
```

The preceding example, for a *left outer join*, would result in the retrieval of all of the rows which have matching values in the columns named X in both tables, plus all of the non-matching rows from the first-mentioned *(left)* table, which is named FIRST, in the FROM clause. Notice that the ON clause is used to specify the matching condition in an outer join, instead of a WHERE clause.

Here is a similar example for a *right outer join*, which would include all rows from the Cartesian Product with matching key values, plus all of the non-matching rows from the right table in the FROM clause:

```
/* RIGHT OUTER JOIN OF TABLES */
PROC SQL;
   SELECT *
      FROM FIRST RIGHT JOIN SECOND
      ON FIRST.X=SECOND.X;
```

What follows is the code for a *full outer join*, which would include all rows from the Cartesian Product with matching key values, plus all of the non-matching rows from both tables in the FROM clause:

```
/* FULL OUTER JOIN OF TABLES */
PROC SQL;
   SELECT *
      FROM FIRST FULL JOIN SECOND
      ON FIRST.X=SECOND.X;
```

Now, an experienced SAS programmer might think that a result set which includes all of the matching rows, plus all of the non-matching rows from both of the tables would be identical with the result set from an ordinary DATA step merge using the X column for a BY-variable, since that column is common to both tables. However, you must remember that join operations combine data by concatenating them in a side-by-side fashion. Therefore, the result set for a join operation would have matching-column values from each of the tables which were joined. In the preceding examples, the columns FIRST.X and SECOND.X would not be combined (overlaid) to form a single column in the result set.

If one desired to combine (overlay) the matching columns in a full outer join, so as to produce the same result as a DATA step merge, one would use the COALESCE function, as follows:

```
/* FULL OUTER JOIN
   (USING COALESCE FUNCTION) */
PROC SQL;
   SELECT
     COALESCE(FIRST.X, SECOND.X) AS X,
      Y1,Y2,Z1,Z2,Z3
   FROM FIRST FULL JOIN SECOND
   ON FIRST.X=SECOND.X;
```

In this example, the SELECT clause specifies each of the columns to be included in the result set, some from the table FIRST, others from the table SECOND. The column named X in the result set would be a composite of the columns having that name in both tables.

Set Operations

Set operations combine information from two tables by concatenating the information in a vertical manner. That is, the data from one table are arranged above the data from the other table in the result set.

There are four set operators:

- Intersect retrieves all unique rows which are common to both tables,
- Union retrieves all unique rows from both tables,
- Outer Union retrieves all rows from both tables, both unique and non-unique,
- Except retrieves all unique rows which are in the first-mentioned table, but which are not also contained in the second-mentioned table (this is a "difference" operator).

The default set operators line up the columns in the result set according to the ordinal positions of the columns in the tables which are being combined. Quite often, this will produce a result set which doesn't make any sense at all (when columns in the same ordinal position of the two tables represent dissimilar items). However, there is an optional keyword, CORRESPONDING, which is used in conjunction with the set operator keywords, and which remedies this situation. Whenever the word CORRESPONDING appears with a set operator, the columns in the tables are lined-up vertically according to column-name, and not according to their ordinal position.

Here are some examples of SAS code for various set operations:

```
/* INTERSECT SET OPERATION OF TABLES */
PROC SQL;
  SELECT * FROM FIRST
  INTERSECT CORRESPONDING
  SELECT * FROM SECOND;

/* OUTER UNION SET OPERATION OF TABLES */
PROC SQL;
  SELECT * FROM FIRST
  OUTER UNION CORRESPONDING
  SELECT * FROM SECOND;
```

Set operations originated in mathematical set theory, but they don't seem to be as useful as joins, in the context of data processing.

USING PROC SQL TO MANAGE DATA

Creating Tables (and SAS Data Files and Views)

There are several methods by which PROC SQL can be used to create new tables (or SAS data files) and views (virtual tables). The most common method is to create tables or views using already-existing tables or views, by defining the rows and columns from the result set of a query.

Here is a general form for the CREATE statement, using other tables or views:

```
CREATE VIEW view-name AS query-expression;
  or
CREATE TABLE table-name AS query-expression ;
  where query-expression is of the form:
       SELECT column-1, column-2, ...
          FROM table-a, table-b, ...
          WHERE expression
          ORDER BY column-r, column-s, ...
```

Here is a typical example of an inner join for a SAS view:

```
PROC SQL ;
  CREATE VIEW TPOUTL AS
     SELECT S.TPNUM, S.OUTLET,
            S.O_NAME, S.O_ADDR,
            S.O_CITY, S.O_STATE,
            S.O_ZIP, S.O_SIC,
            T.TAXCD, T.OPSTATCD,
            T.O_OOBDAT, T.O_PRMDAT,
            T.O_FSLDAT, T.PERMSTAT
       FROM MYLIB.TPOUTLET AS S,
            MYLIB.OUTLPERM AS T
       WHERE S.TPNUM=T.TPNUM
             AND S.OUTLET=T.OUTLET
       ORDER BY TPNUM;
```

This join would create a temporary view which combines sales taxpayer outlet general information with outlet permit information, matching rows from the two tables according to taxpayer number and outlet number. It could just as easily have been for a temporary table (SAS data file).

Another way to create a new table with PROC SQL would be to define the columns and then to fill in the rows of data later, using the INSERT statement. Consult a reference manual to learn the details of this approach.

Modifying Tables and Views

PROC SQL has several statements which can be used to modify tables and views which were created previously.

- The ALTER statement can be used to change one or more of the columns of an already existing table. Using ALTER,

you can add new columns, you can change the formats which are used to display column values, and you can remove columns.

- The DELETE statement is used to remove one or more rows from a table. You must use a WHERE statement to specify a defining condition for the rows to be eliminated; if you don't, all of the rows will be deleted.
- To delete a table, the DROP TABLE statement is used.
- The INSERT statement inserts a new row into an existing table.
- The UPDATE statement modifies the values of columns in existing rows of a table.

Data Manipulation Using PROC SQL

We have discussed how to write PROC SQL statements which create tables, how to create temporary columns from existing columns, how to sort the rows which are displayed in the result set, and how to associate labels and formats with columns. We have seen the usefulness of summary functions to calculate statistics for the entire table, and also for classification groups. It is easy to see how knowledge of the use of these techniques is important for programmers who spend much of their time working with RDBMS data. These methods also could be very useful for working with non-RDBMS data.

PROC SQL's substantial data manipulation and summarization capabilities could be used to replace many DATA step, PROC SORT, and PROC MEANS steps in traditional SAS code. Coding a PROC SQL step may require fewer lines, and the PROC SQL code generally will execute in less time, than the corresponding traditional SAS code.

The following are simple examples of SAS code which produces reports, from a SAS data file which contained advance registration information for the 1996 South-Central Regional SAS Users' Group Conference.

Here is the traditional SAS code:

```
DATA A;
  SET MYLIB.SC96REG;
  AMTDUE = TOTAL - AMTPAID;
  KEEP CITY COMPANY STATUS AMTPAID;
PROC SORT  DATA=A;
  BY CITY COMPANY PAID;
PROC MEANS  DATA=A  N  NOPRINT;
  VAR AMTDUE;
  BY CITY COMPANY STATUS;
  OUTPUT OUT=STATS SUM(AMTDUE)=  ;
PROC PRINT  DATA=STATS;
RUN;
```

The following PROC SQL step produces essentially the same report as the preceding traditional code:

```
PROC SQL;
  SELECT CITY, COMPANY, STATUS,
      FREQ(STATUS) AS NUMTYP,
      SUM(TOTAL - AMTPAID) AS AMTDUE
  FROM MYLIB.SC96REG
  GROUP BY CITY, COMPANY, STATUS
  ORDER BY CITY, COMPANY;
```

In the preceding examples, the traditional SAS program contains more lines of code, and requires 2.6 times as much CPU time to execute, than the PROC SQL step. Replacing customary SAS code with PROC SQL lines generally will result in less programming time and greater computer efficiency.

The SQL Pass-Through Facility

Another method for accessing DBMS tables from a SAS session is the Pass-Through Facility of the SQL Procedure. The SQL Pass-Through Facility allows programmers to embed DBMS code within SAS SQL expressions. The program can specify exactly what processing should take effect on the DBMS side and make use of any special features that the DBMS offers.

The *SQL Pass-Through Facility* uses a SAS/ACCESS® interface to establish connection to the DBMS, and it sends native SQL statements to the DBMS. The idea behind SQL Pass-Through is to push as much work as possible into the DBMS, since its query optimizer knows all about the storage structure of the tables that are in the DBMS. Because of this, when you are selecting data from multiple tables, the DBMS generally is able to return the result set more quickly than if you did everything from within the SAS System, using SAS view descriptors.

Here is an DB2 example of SQL Pass-Through:

```
PROC  SQL ;
  CONNECT TO DB2  (SSID=DSNT) ;
  CREATE  VIEW  COUNTY  AS
    SELECT * FROM CONNECTION TO DB2
      (SELECT  *  FROM
          COMBCOD.VCOUNTY01) ;
  %PUT &SQLXMSG ;
  DISCONNECT FROM DB2 ;
QUIT ;
PROC  PRINT  DATA=COUNTY ;
RUN ;
```

Observe the SELECT clause which is enclosed in parentheses, following the "SELECT * FROM CONNECTION TO DB2" clause. It is written using the DB2 version of SQL. Using SAS table names and SAS column-names would not produce the desired result.

CONCLUSION

PROC SQL processes SQL statements that read and update tables. Besides being used for retrieving and updating data in relational tables and databases, PROC SQL also has substantial data manipulation and summarization capabilities. PROC SQL statements could be used to replace much traditional SAS code (DATA step, PROC SORT, and PROC MEANS steps), resulting in less programming time and greater computer efficiency.

REFERENCES:

- SAS Guide to the SQL Procedure, Usage and Reference, Version 6, First Edition
- Getting Started with the SQL Procedure, Version 6, First Edition
- "The SQL Procedure", Chapter 37 of SAS Technical Report P-222, Changes and Enhancements to Base SAS Software, Release 6.07.
- Alan Dickson, and Ray Pass, "Select Items from PROC.SQL Where Items > Basics", Proceedings of the Nineteenth Annual SAS Users Group International

Conference (1994), pp. 1440-1449; and Proceedings of the Twentieth Annual SAS Users Group International Conference (1995), pp. 432-441.
- Kim L. Kolbe Ritzow, "An Introduction to PROC SQL", Proceedings of the Twenty-First Annual SAS Users Group International Conference (1996), pp. 327-335.
- Kirk Paul Lafler, "Using the SQL Procedure", Proceedings of the Seventeenth Annual SAS Users Group International Conference (1992), pp. 555-560.
- Kirk Paul Lafler, "Diving Into SAS Software With the SQL Procedure", Proceedings of the Twentieth Annual SAS Users Group International Conference (1995), pp. 1076-1081.

AUTHOR INFORMATION:

Thomas J. Winn, Jr.
Audit HQ, Comptroller of Public Accounts
L.B.J. State Office Building
111 E 17th Street
Austin, TX 78774

Telephone: (512) 463-4907
E-Mail: twin504@cpa.state.tx.us
0007095870@mcimail.com