Intermediate PROC SQL

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

ABSTRACT

This tutorial presentation will provide a practical explanation of some advanced features of the SAS ® SQL Procedure, including the use of: summary functions, subqueries, complex joins, in-line views, indexes, macro variables, and the SQL Pass-Through Facility. Also included will be a discussion of several useful tips for improving the performance of PROC SQL queries. This paper is for SAS programmers who already know how to code simple SELECT … FROM … WHERE … ORDER BY … query statements for PROC SQL, and who want to be able to do more.

This presentation does *not* explain SAS/ACCESS concepts, nor the ACCESS, nor DBLOAD procedures. SAS/ACCESS software provides interfaces for many popular database products.

EXAMPLES

The illustrative examples used in this paper are based upon a simplified, hypothetical personnel data base for a fictitious company. It should *not* be identified with any data base which actually is used by the State of Texas, or any of its agencies. The imaginary data base, PERSLIB (which could be a SAS Data Library), contains numerous tables (possibly SAS data sets); however, we shall be concerned with only four of them: DIVISION, EMPLOYEE, JOBV, and ACTION.

DIVISION includes current and historical information pertaining to the company's organizational structure. Each of the operational entities, which are called divisions, are identified by their number and name, DIVNUM and DIVNAME, respectively. Each division has a lifespan which begins on its DIVBEGDT, and ends on its DIVENDDT. The value of DIVENDDT would be null for current divisions.

EMPLOYEE includes current and selected historical information pertaining to company employees. There is only one row (observation) for each person ever employed by the company. It includes personal information, such as EMPNUM, and EMPNAME, RACE, SEX, BIRTHDAT, the number of years of formal education completed (EDUYRS), as well as the most recent home address information and home telephone number. It also contains certain items which are of particular concern to the company: the date on which the employee was hired by the company (HIREDATE), the date on which the employee may have terminated his/her employment with the company (TERMDATE), and the unique number associated with the employee's most recent job assignment (POSITION).

JOBV includes current and selected historical information pertaining to POSITIONs (and *not* directly with EMPNUMs) which were established in each DIVISION. JOB includes such information as job classification code (CLASSCD), job category code (EMPLTYPE), and current salary amount (MOSALRY). Each job position also has a life span, which begins on its POSBEGDT and ends on its POSENDDT.

ACTION contains current and historical information regarding changes in job assignment or salary, associated with each employee ever employed by the company. Each record includes EMPNUM, DIVNUM, POSITION, MOSALRY, and the effective date of the change, EFFDATE.

A BRIEF REVIEW OF FUNDAMENTAL IDEAS

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. Each RDBMS may use its own particular "dialect" of SQL.

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. PROC SQL includes several enhancements, which *exceed* the ANSI specifications, for greater compatibility with other elements of the SAS System.

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 be used on SAS files, flat files, VSAM files, database tables, and combinations of these to do query operations. PROC SQL also can perform many ordinary data manipulation and reporting operations customarily accomplished using DATA step programming, and the PRINT, SORT, MEANS and SUMMARY procedures. (See the SUGI 22 paper by Winn)

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;
```

For a complete explanation of these components, please consult the PROC SQL reference manual (see references).

Queries, Views, Joins, 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 information gathered from several tables.

The most common way of combining data from several tables is through a *join* operation. Joins combine information from multiple tables by matching rows that have common values in key columns which relate the tables. Tables do not have to be sorted before they are joined.

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

Elementary PROC SQL Syntax for Queries

The most frequently-encountered usage 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.

A basic form for the SELECT statement is:

```
SELECT column-1, column-2, …
    FROM table-a, table-b, …
    WHERE expression
    ORDER BY column-r, column-s, … ;
```

The SELECT statement specifies the column-names in a particular table (the FROM clause) from which the data are to be chosen, it further subsets these data according to a certain value contained in some of the rows (the WHERE clause), and then it identifies the column to be used as the basis for re-sequencing the extracted data for the printed report (the ORDER BY clause).

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

```
PROC SQL ;
   SELECT  A.DIVNUM,
           A.DIVNAME,
           B.EMPNUM,
           B.EMPNAME,
           B.HIREDATE,
           B.RACE,
           B.SEX,
           B.BIRTHDAT,
           B.HADDR,
           B.HCITY,
           B.HSTATE,
           B.HZIP,
           B.HPHONE,
           C.CLASSCD,
           C.EMPLTYPE,
           C.MOSALRY
      FROM PRSLIB.DIVISION     AS A,
           PRSLIB.EMPLOYEE AS B,
           PRSLIB.JOBV           AS C
      WHERE DIVNUM IN (14, 19)
            AND A.DIVENDDT IS NULL
            AND A.DIVNUM=C.DIVNUM
            AND B.POSITION=C.POSITION
            AND B.TERMDATE IS NULL
            AND C.JOBBEGDT>='01SEP1997'D
            AND C.JOBENDDT IS NULL
        ORDER BY DIVNUM, EMPNUM;
```

(This query is an inner join which would return selected information concerning all of the current employees in divisions 14 or 19 who are working in current job assignments which were established since September 1, 1997.)

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 PRSLIB.EMPLOYEE;
```

(The preceding query would return the entire EMPLOYEE table.)

The keyword DISTINCT is used before a column name to eliminate duplicate rows in the result set. When DISTINCT is used, only one row would be displayed for each unique combination of values returned from the query. For example,

```
PROC SQL;
    SELECT DISTINCT CLASSCD
        FROM  PRSLIB.JOBV;
```

(There may be several employees who occupy positions which are described by the same job classification code. The preceding query would return a list of all of the different values of CLASSCD which are, or have been, used.)

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

```
PROC SQL ;
   SELECT EMPNUM, POSITION,
          INT((TODAY()-HIREDATE)/365.25)
             AS EMPYRS
     FROM PRSLIB.EMPLOYEE;
```

(This query calculates the number of years which have elapsed since the initial date of employment, for all persons included in the EMPLOYEE table.)

The WHERE clause is used to specify one or more conditions that the data must satisfy in order to be selected.

```
PROC SQL ;
   SELECT EMPNUM, EMPNAME,
          HADDR, HZIP, HPHONE
     FROM PRSLIB.EMPLOYEE
     WHERE TERMDATE IS NULL
           AND  HCITY='AUSTIN' ;
```

(This query would return name, home address, and home telephone number for each current employee having an address in the city of Austin.)

```
PROC SQL ;
   SELECT DIVNUM, EMPNUM, EMPNAME,
          RACE, SEX, HIREDATE, BIRTHDAT
     FROM PRSLIB.EMPLOYEE
     WHERE TERMDATE IS NULL
           AND HIREDATE BETWEEN
               '01SEP94'D AND '31AUG97'D ;
```

(This query would return selected information regarding current employees who were hired on or after Sept. 1, 1994 and on or before Aug. 31, 1997.)

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

```
PROC SQL ;
   SELECT EMPNUM, POSITION,
          INT((TODAY()-HIREDATE)/365.25)
             AS EMPYRS
     FROM PRSLIB.EMPLOYEE
     WHERE  TERMDATE IS NULL
            AND CALCULATED EMPYRS > 35;
```

(This query returns a listing of all current employees with at least 36 years of service, together with their

job assignment number and calculated duration of employment.)

We 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 A.POSITION,
          A.CLASSCD,
          A.MOSALRY,
          B.EMPNUM,
          B.EMPNAME
      FROM PRSLIB.JOBV      AS A
           PRSLIB.EMPLOYEE  AS B,
      WHERE A.DIVNUM = 19
            AND A.JOBENDDT IS NULL
            AND A.POSITION=B.POSITION
            AND B.TERMDATE IS NULL
       ORDER BY EMPNAME;
```

(This query returns a roster of all current employees in division 19, including their job assignment number, job classification code, monthly salary, employee number, and name. The listing would be displayed in alphabetical order of employee name.)

```
PROC SQL ;
   SELECT A.DIVNUM,
          A.POSITION,
          A.CLASSCD,
          A.MOSALRY,
          B.EMPNUM,
          B.EMPNAME
      FROM PRSLIB.JOBV      AS A
           PRSLIB.EMPLOYEE  AS B,
      WHERE A.JOBENDDT IS NULL
            AND A.POSITION=B.POSITION
            AND B.TERMDATE IS NULL
     ORDER BY DIVNUM, MOSALRY DESC;
```

(This query returns a roster of all current employees, including their division number, job assignment number, job classification code, monthly salary, employee number, and name. The listing would be displayed in ascending order of division number, and, within each division, in descending order of salary amount.)

SUMMARY FUNCTIONS

The SQL procedure provides summary functions which calculate statistics from the data values in an entire table, or for each of several classification groups. The summary functions include such useful tools as: MEAN, FREQ, MAX, MIN, RANGE, STD, VAR, and SUM, among others.

```
PROC SQL ;
   SELECT FREQ(EMPNUM) AS COUNT,
      FROM PRSLIB.EMPLOYEE
      WHERE TERMDATE IS NULL;
```

(This query determines the total number of current employees.)

Whenever more than one column is specified in a summary function, then the summary function acts like a DATA step function. The calculation would be carried out for each row. Used appropriately, this property can be very useful. However, in the absence of clear thinking, it could lead to some unintended results.

```
PROC SQL ;
   SELECT A.DIVNUM,
          AVG(A.MOSALRY, B.EDUYRS)
              AS AVERAGE,
      FROM PRSLIB.JOBV      AS A
           PRSLIB.EMPLOYEE  AS B,
      WHERE A.JOBENDDT IS NULL
            AND A.POSITION = B.POSITION
            AND B.TERMDATE IS NULL;
```

(This query determines the *row-wise* average of the values for monthly salary and years of education, for each current employees. Obtaining the result of this calculation for each row should not be regarded as useful information.)

If the SELECT clause specifies more than one column then, after the calculation is performed, the result is re-merged with the individual rows of the table.

```
PROC SQL ;
   SELECT A.DIVNUM,
          SUM(A.MOSALRY) AS TOTAL,
      FROM PRSLIB.JOBV      AS A
           PRSLIB.EMPLOYEE  AS B,
      WHERE A.JOBENDDT IS NULL
            AND A.POSITION = B.POSITION
            AND B.TERMDATE IS NULL;
```

(This query calculates the total amount of monthly salary payable to current employees. This total, which includes *all* divisions, is then returned with the individual indicators for *each* division.)

The GROUP BY clause is used to separate the data into groups, based upon the distinct values in the column which is specified.

```
PROC SQL ;
   SELECT A.DIVNUM,
          SUM(A.MOSALRY) AS TOTAL,
      FROM PRSLIB.JOBV      AS A
           PRSLIB.EMPLOYEE  AS B,
      WHERE A.JOBENDDT IS NULL
            AND A.POSITION = B.POSITION
            AND B.TERMDATE IS NULL
     GROUP BY DIVNUM;
```

(This query calculates the total amount of monthly salary payable to current employees, by division.)

```
PROC SQL ;
   SELECT A.DIVNUM,
          FREQ(B.EMPNUM) AS COUNT,
      FROM PRSLIB.JOBV      AS A
           PRSLIB.EMPLOYEE  AS B,
      WHERE A.JOBENDDT IS NULL
            AND A.POSITION = B.POSITION
            AND B.TERMDATE IS NULL
     GROUP BY DIVNUM;
```

(This query determines the total number of current employees in each division.)

SUBQUERIES

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  EMPNUM,
            EMPNAME,
            POSITION,
            HIREDATE
        FROM PRSLIB.EMPLOYEE
        WHERE  TERMDATE IS NULL
            AND  POSITION  IN
                ( SELECT POSITION
                    FROM PRSLIB.JOBV
                    WHERE JOBENDDT IS NULL
                            AND EMPLTYPE='RF' )
        ORDER BY  EMPNUM ;
```

(This query returns a listing all current employees working in positions for which the job category code is "regular full-time".)

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

```
PROC SQL ;
    SELECT  A.EMPNUM,
            A.EFFDATE AS NEWDATE,
            A.DIVNUM AS NEWDIV,
            A.POSITION AS NEWPOS,
            A.MOSALRY AS NEWSAL,
            B.EFFDATE AS OLDDATE,
            B.DIVNUM AS OLDDIV,
            B.POSITION AS OLDPOS,
            B.MOSALRY AS OLDSAL,
            C.EMPNAME
        FROM PRSLIB.ACTION      AS  A,
             PRSLIB.ACTION      AS  B,
             PRSLIB.EMPLOYEE AS  C
        WHERE  A.EMPNUM=B.EMPNUM
               AND A.EMPNUM=C.EMPNUM
               AND A.EFFDATE >= '01JAN1997'D
               AND B.EFFDATE =
              ( SELECT MAX(EFFDATE)
                FROM PRSLIB.ACTION   AS D
                WHERE  A.EMPNUM=D.EMPNUM
                 AND D.EFFDATE < A.EFFDATE )
    ORDER BY EMPNUM ;
```

(This query returns a listing of all changes in job assignment or salary since Jan. 1, 1997 for all employees. The listing includes the date of every change, the job assignment and salary both before, and after, each change, and the date of the preceding change.)

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. Conceptually, correlated subqueries are pretty tricky.

CREATING TABLES (and SAS Data Files and Views)

PROC SQL can be used to create new tables (or SAS data files) and views (virtual tables) in several ways. As in SAS DATA step programming, using a one-level name in a PROC SQL CREATE statement would create a temporary entity, whereas usage of a two-level name (using a previously-defined *libref*) would create a permanent entity.

One way to create a new table with PROC SQL would be, first, to define the columns and, afterward, to fill-in the rows of data.

Here is the general syntax used for creating new tables (SAS data files) *without rows*::

```
CREATE TABLE table-name
      (column-1  type
          <(length) informat=… format=…  label='…' >,
       column-2   type
          <(length) informat=… format=…  label='…' >,
        … )
or
CREATE TABLE table-b LIKE table-a;
```

After the table exists, one may load the rows of data values by using the INSERT statement.

```
INSERT INTO table-name
  SET column-name-1=expression-1 ,
        column-name-2=expression-2, …;
or
INSERT INTO table-name
  VALUES (value-1, value-2, …);
  VALUES (value-a, value-b, …);  …
```

The data values for each column are specified positionally in a single row, one row at a time.

For example, I could create a permanently-stored table of job descriptions, for use in conjunction with my other PRSLIB tables, as follows:

```
PROC SQL;
   CREATE TABLE MYLIB.JOBCLASS
        (CLASSCD CHAR(4), CLASSTL CHAR(25));
   INSERT INTO MYLIB.JOBCLASS
        VALUES('A001', 'Clerk 1')
        VALUES('A002', 'Clerk 2')
        VALUES('B050', 'Equipment Operator')
        VALUES('C022', 'Technician')
         … ;
```

The most common method of creating new tables or views is by defining the rows and columns as the result set of a query of one or more already-existing tables or views.

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-name-1, column-name-2, …
    FROM table-name-a, table-name-b, …
    WHERE expression
    ORDER BY column-name-r,
                column-name-s, …
```

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

```
PROC SQL ;
   CREATE VIEW  RECENT AS
      SELECT  A.EMPNUM,
              A.EFFDATE,
              A.DIVNUM,
              A.POSITION,
              A.MOSALRY,
              B.EMPNAME,
```

```
            B.HIREDATE
        FROM PRSLIB.ACTION       AS  A,
             PRSLIB.EMPLOYEE AS  B
        WHERE  A.EMPNUM=B.EMPNUM
               AND A.EFFDATE >=
                          '01SEP1997'D
        ORDER BY EMPNUM, EFFDATE ;
```

(This query returns a view, named RECENT, which includes all changes in job assignment or salary since Sept. 1, 1997 for all employees, together with the employee's name and date of employment. The view includes the date of every change, as well as the job assignment and salary.)

The preceding join would create a temporary view which combines assignment history and selected general employee information, matching rows from the two tables according to employee number. It could just as easily have been for a temporary table (SAS data file). Since RECENT was created as a view, it actually contains no data values but, instead, it is a definition for a virtual table. However, subsequent steps in the program could refer to RECENT just as if it were an ordinary SAS data file.

If I had wanted to create a permanently stored table or view, then I would have used a *libref* for a SAS data library in a two-level name in the CREATE statement. For example, I might have substituted the following code-fragment:
…CREATE VIEW MYLIB.RECENT AS …

IN-LINE VIEWS

Consider the following coding situation.

```
PROC SQL ;
   CREATE VIEW  JOBS AS
      SELECT  A.DIVNUM,
              A.DIVNAME,
              B.POSITION,
              B.CLASSCD,
              B.EMPLTYPE,
              B.MOSALRY
         FROM PRSLIB.DIVISION  AS  A,
              PRSLIB.JOBV       AS  B
         WHERE A.DIVNUM IN (14, 19)
               AND A.DIVNUM=B.DIVNUM
               AND A.DIVENDDT IS NULL
               AND B.JOBBEGDT >=
                          '01SEP1997'D
               AND B.JOBENDDT IS NULL
         ORDER BY POSITION ;
   CREATE VIEW  PEOPLE AS
      SELECT  EMPNUM,
              POSITION,
              EMPNAME,
              HIREDATE,
              RACE,
              SEX,
              BIRTHDAT,
              HADDR,
              HCITY,
              HSTATE,
              HZIP,
              HPHONE
         FROM PRSLIB.EMPLOYEE
              WHERE TERMDATE IS NULL
         ORDER BY POSITION ;
   CREATE TABLE ASSIGNS AS
      SELECT *
         FROM JOBS       AS  C,
              PEOPLE  AS  D
         WHERE C.POSITION=D.POSITION
         ORDER BY DIVNUM, EMPNUM ;
```

The preceding code used three steps: the first step combined the DIVISION and JOBV data, the second step extracted certain information from EMPLOYEE, and the third step matched the work assignments with the associated people by matching according to the value of POSITION.

We have seen nested query-expressions in which the WHERE clause contains a query-expression (a subquery). An *in-line view* is when a query-expression is used in the place of a table name in a FROM clause. In certain cases, this may save some coding steps. Let me illustrate.

```
PROC SQL ;
   SELECT *
      FROM  (SELECT  A.DIVNUM,
                     A.DIVNAME,
                     B.POSITION,
                     B.CLASSCD,
                     B.EMPLTYPE,
                     B.MOSALRY
                FROM PRSLIB.DIVISION  AS A,
                     PRSLIB.JOBV       AS B
                WHERE A.DIVNUM IN (14, 19)
                     AND A.DIVNUM=B.DIVNUM
                     AND A.DIVENDDT IS NULL
                     AND B.JOBBEGDT >=
                                '01SEP1997'D
                     AND B.JOBENDDT IS NULL
                ORDER BY POSITION )       AS  C,
             (SELECT  EMPNUM,
                     POSITION,
                     EMPNAME,
                     HIREDATE,
                     RACE,
                     SEX,
                     BIRTHDAT,
                     HADDR,
                     HCITY,
                     HSTATE,
                     HZIP,
                     HPHONE
                FROM PRSLIB.EMPLOYEE
                WHERE TERMDATE IS NULL
                ORDER BY POSITION )       AS  D
      WHERE C.POSITION=D.POSITION
      ORDER BY DIVNUM, EMPNUM
```

(This query returns selected information concerning all current employees in divisions 14 or 19, who are working in current job assignments which were established since September 1, 1997.)

The preceding in-line query could have been coded in the following way.

```
PROC SQL ;
   CREATE VIEW  JOBS AS
      SELECT  A.DIVNUM,
              A.DIVNAME,
              B.POSITION,
              B.CLASSCD,
              B.EMPLTYPE,
              B.MOSALRY
         FROM PRSLIB.DIVISION  AS  A,
              PRSLIB.JOBV       AS  B
         WHERE A.DIVNUM IN (14, 19)
               AND A.DIVNUM=B.DIVNUM
               AND A.DIVENDDT IS NULL
               AND B.JOBBEGDT >=
                          '01SEP1997'D
```

```
                    AND B.JOBENDDT IS NULL
            ORDER BY POSITION ;
    CREATE VIEW  PEOPLE AS
        SELECT  EMPNUM,
                POSITION,
                EMPNAME,
                HIREDATE,
                RACE,
                SEX,
                BIRTHDAT,
                HADDR,
                HCITY,
                HSTATE,
                HZIP,
                HPHONE
            FROM PRSLIB.EMPLOYEE
                 WHERE TERMDATE IS NULL
            ORDER BY POSITION ;
    CREATE TABLE ASSIGNS AS
        SELECT *
            FROM JOBS       AS  C,
                 PEOPLE  AS  D
            WHERE C.POSITION=D.POSITION
            ORDER BY DIVNUM, EMPNUM ;
```

CREATING INDEXES

The SQL procedure can be used to create an index for tables which may, under some circumstances, facilitate more efficient retrieval of the rows in the table. However, it should be noted that Indexes are not recommended for small tables, and the number of indexes should be kept to a minimum.

```
PROC SQL ;
    CREATE INDEX DIVJOBS
        ON PRSLIB.JOBV(DIVNUM)

PROC SQL ;
    CREATE UNIQUE INDEX EMPNUM
        ON PRSLIB.EMPLOYEE(EMPNUM)
```

(Since each EMPNUM in the table EMPLOYEE is unique, a unique index can be defined for that column. This would not be the case if there were multiple instances for any value of EMPNUM; for example, in the event that EMPNUM contains some null values.)

The SAS System will determine the most efficient way to process a query. Therefore, PROC SQL will decide whether or not to use an index in executing the query.

MACRO VARIABLES

Macro variables are used to facilitate symbolic substitution of strings of text. They are particularly useful in repetitive coding situations. The customary way of defining a macro variable is with a %LET statement. Here is an example:

```
%LET CURRDT = '11NOV97'D;
%LET DIV = 14;
PROC SQL ;
    CREATE VIEW  TRANS AS
        SELECT  EMPNUM,
                EFFDATE,
                POSITION,
                MOSALRY,
            FROM PRSLIB.ACTION
            WHERE  DIVNUM=&DIV
                   AND EFFDATE = &CURRDT
            ORDER BY EMPNUM, POSITION ;
```

The preceding would resolve to:

```
PROC SQL ;
    CREATE VIEW  TRANS AS
        SELECT  EMPNUM,
                EFFDATE,
                POSITION,
                MOSALRY,
            FROM PRSLIB.ACTION
            WHERE  DIVNUM=14
                   AND EFFDATE='11NOV97'D
            ORDER BY EMPNUM, POSITION ;
```

This might not look like a very big deal, but if your programs include multiple statements involving the same date, for example, and if you have to run it regularly with revised date values each time, then using a macro variable can save you a lot of line-by-line editing.

PROC SQL provides another method for creating macro variables, where the value is determined by a query.

The syntax is

```
PROC SQL NOPRINT;
    SELECT  sql-expression-1,
            sql-expression-2,
            sql-expression-3,
            …
        INTO  :macro-variable-1,
              :macro-variable-2,
              :macro-variable-3,
              …
    FROM  . . .
    WHERE . . . ;
```

Here is an example:

```
PROC SQL ;
    SELECT FREQ(EMPNUM) INTO :COUNT,
        FROM PRSLIB.EMPLOYEE
        WHERE TERMDATE IS NULL;

%PUT There are &COUNT current employees. ;
```

If the total number of employees turns out to be, say, 147, then the preceding code would result in the following line in the SAS Log:

```
There are 147 current employees.
```

Some of the references noted at the end of this paper contain interesting examples which illustrate how useful this interface to the macro facility can be for application development purposes.

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 the general syntax for SQL Pass-Through:

```
PROC  SQL ;
  CONNECT TO dbms-name <AS alias>
                <(dbms-argument-1 = value …
                 <dbms-argument-n = value>)> ;
  EXECUTE
     (dbms-specific-SQL-statement)
     BY dbms-name/alias;
  SELECT *
     FROM CONNECTION TO dbms-name/alias
     (dbms-specific-SQL-query-statement) ;
  %PUT &SQLXRC;
  %PUT &SQLXMSG;
  DISCONNECT FROM dbms-name/alias ;
QUIT ;
```

The CONNECT statement establishes a connection with a DBMS which is supported by SAS/ACCESS software. If the DBMS supports multiple connections, then an *alias* can be specified to refer to a particular connection. The connection remains in effect until the DISCONNECT statement is encountered. The EXECUTE statement can be used, for example, for creating or modifying tables or views or indexes. The SELECT … FROM CONNECTION TO … statement is used for retrieving DBMS data in a PROC SQL query or view. SQLXRC and SQLXMSG are two SAS macro variables which are used for capturing DBMS-generated return codes and descriptive messages resulting from error conditions.

This is a DB2 query example of SQL Pass-Through:

```
PROC  SQL ;
  CONNECT TO DB2  (SSID=DSNQ) ;
  CREATE  VIEW  LOCALS  AS
    SELECT * FROM CONNECTION TO DB2
      (SELECT *
         FROM  PERS.VCOMPANY_EMPLOYEE01
         WHERE TERMINATION_DATE IS NULL
                 AND  HOME_CITY='AUSTIN' ) ;
  %PUT &SQLXMSG ;
  DISCONNECT FROM DB2 ;
QUIT ;
PROC  PRINT  DATA=LOCALS ;
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.

It also is permissible to incorporate macro variables in the SQL Pass-Through statements. For example:

```
%LET CURRDT='11/11/1997';
%LET DIV=14
 PROC SQL ;
  CONNECT TO DB2  (SSID=DSNQ) ;
  CREATE  VIEW  DIV&DIV  AS
    SELECT * FROM CONNECTION TO DB2
      (SELECT *  FROM
              PERS.VPERS_TRANSACTION01
              WHERE  DIVISION_NUM=&DIV
                      AND EFFECTIVE_DATE =
                              &CURRDT ) ;
%PUT &SQLXMSG ;
DISCONNECT FROM DB2 ;
QUIT ;
PROC  PRINT  DATA=DIV&DIV ;
RUN ;
```

In the preceding example, even though the DB2 SQL expression inside the parenthesis includes SAS macro references, these will be resolved before the text is sent to the RDBMS.

CONCLUSION

PROC SQL processes SQL statements that read and update tables. It includes many advanced features which are worth investigating. Incorporating some of these methods into the SAS code you write may result in less programming time and greater computer efficiency.

REFERENCES:

- SAS Guide to the SQL Procedure, Usage and Reference, Version 6, First Edition
- "The SQL Procedure", Chapter 5 of SAS Technical Report P-221, SAS/ACCESS Software: Changes and Enhancements, Release 6.07.
- "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, ***and*** Proceedings of the Twenty-First Annual SAS Users Group International Conference (1996), pp. 227-236.
- Paul Kent, "An SQL Tutorial – Some Random Tips", Proceedings of the Twenty-First Annual SAS Users Group International Conference (1996), pp. 237-241.
- Thomas J. Winn Jr., "Introduction to Using PROC SQL", Proceedings of the Twenty-Second Annual SAS Users Group International Conference (1997), pp. 383-389.

AUTHOR INFORMATION:

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

Telephone: (512) 463-4907
E-Mail: tom.winn@cpa.state.tx.us