ABSTRACT

This paper will present a comparison of typical SAS programming techniques to the equivalent code needed to perform similar functionality using PROC SQL. The SQL procedure implements the Structured Query Language (SQL) for the SAS System. SQL is a standardized, open language originally developed to work in conjunction with Relational Database Management Systems (RDBMS) such as Oracle® and DB2®. SQL was primarily designed to perform Direct Data Access (DDA) functions, such as retrieving and updating data in tables (data sets), however its functionality is much more advanced in today’s software.

INTRODUCTION

The language was first created by IBM® in 1975 and was called SEQUEL for “Structured English Query Language.” SQL was adopted as a standard by the American National Standards Institute (ANSI) in 1986 and the International Organization for Standardization (ISO) in 1987. The SAS System’s SQL procedure, introduced in 1990, enables the creation or retrieval of data stored in tables or views. It can create SAS macro variables that contain values from rows of a query’s result. The table at the right displays the equivalent nomenclature for SAS and SQL terminology. Throughout this paper SAS and SQL code will be presented in uppercase letters while supplied variable names and values in lowercase letters.

CREATING TABLES

The examples in figures 1 and 2 are meant to show how data sets can be created using the DATA step as well as the SQL procedure. One major difference is the use of the keyword QUIT in lieu of RUN. Multiple statements can be used within the SQL procedure as indicated with the CREATE TABLE and INSERT INTO statements.

SUB-SETTING TABLES

Limiting a data set to desired records can be done with the WHERE clause in both a DATA step and SQL. To capture values within a given range the SQL statement can use the BETWEEN condition as shown in Figure 4. The greater than, less than, and equal to operators must be used to obtain the same result within a DATA step where PROC SQL supports both comparison operations. Note the asterisk in the SQL tells the system to output all variables.
SORTING

PROC SORT is typically used to sort data within the SAS system as compared to the ORDER BY clause of SQL. Within the SAS system, SQL will sort data in the exact order as PROC SORT, though implementations of SQL from other vendors can vary. In some cases character variables that begin with numeric values show up in the beginning of the table and others at the end. This has serious implications when joining or merging data from other vendors. Note that tables created using PROC SQL can be later sorted by PROC SORT as the results are a SAS data set.

PROC SORT DATA=table1a;
  BY var1;
RUN;

PROC SQL;
  CREATE TABLE table1b AS
  SELECT *
  FROM table1b
  ORDER BY var1;
QUIT;

Figure 5: Sorting Data using PROC SORT

Figure 6: Sorting Data using SQL

ELIMINATING DUPLICATES

Obtaining unique values within a data set can be done in the SAS system by using the NODUPS option within a PROC SORT. The original table can be preserved by including the OUT= option as shown in Figure 7. In SQL unique values can be obtained by using the keyword DISTINCT. In comparison, the original table can be preserved by specifying a different table name on the CREATE TABLE statement.

PROC SORT DATA=table1a
  NODUPS OUT=table2a;
  BY var1;
RUN;

PROC SQL;
  CREATE TABLE table2b AS
  SELECT DISTINCT *
  FROM table1b
  ORDER BY var1;
QUIT;

Figure 7: Eliminating Duplicates using PROC SORT

Figure 8: Eliminating Duplicates using SQL

ELIMINATING DUPLICATE KEYS

By using the NODUPKEY option PROC SORT keeps the FIRST record of the BY group similar to the FIRST (dot) notation in a DATA step. This means the first time SAS encounters a new BY group it keeps the first record it encounters regardless of the values within other variables in the data set. An example of FIRST (dot) notation is shown later in this paper.

Although there is not a true equivalent in the SAS implementation of SQL to keep the first record of a BY group as compared to the NODUPKEY option, other functionality exists that can be used to eliminate duplicates with a more clearly defined logic. The example in Figure 10 will only keep the records with the minimum values of all variables.

PROC SORT DATA=table1a
  NODUPKEY OUT=table3a;
  BY var1;
RUN;

PROC SQL;
  CREATE TABLE table3b AS
  SELECT MIN(var1) AS var1,
       MIN(var2) AS var2,
       MIN(var3) AS var3,
       MIN(numvar) AS numvar
  FROM table1b
  GROUP BY var1
  ORDER BY var1;
QUIT;

Figure 9: Eliminating Duplicates Keys PROC SORT

Figure 10: Eliminating Duplicates Keys using SQL
GROUPING

The grouping of data allows for the accumulation of summary variables. Using the DATA step to group and summarize information, the data must first be sorted. This is not the case when using SQL, though it can help efficiency tremendously. In Figure 11, the FIRST (dot) notation is used in the DATA step to set the value of the first record in the BY group to zero. Accumulation of values occurs and the LAST record of the group is OUTPUT. SQL may be easier to follow as the COUNT and SUM functions, accompanied by the GROUP BY, perform this function.

```sas
PROC SORT DATA=table1a;
   BY var1;
RUN;

DATA table5a ( DROP= numvar );
   SET table1a ( KEEP= var1 numvar);
   BY var1;
   IF FIRST.var1 THEN DO;
      cntvar = 0;
      sumvar = 0;
   END;
   cntvar + 1;
   sumvar + numvar;
   IF LAST.var1 THEN OUTPUT;
RUN;
```

Figure 11: Grouping Data Using the DATA Step

```sql
PROC SQL;
   CREATE TABLE table5b AS
      SELECT var1,
             COUNT(*) AS cntvar,
             SUM(numvar) AS sumvar
      FROM table1b
      GROUP BY var1;
   QUIT;
```

Figure 12: Grouping Data Using SQL

```sas
PROC SUMMARY DATA=table1b NWAY MISSING;
   CLASS var1;
   VAR numvar;
   OUTPUT OUT=table5c
      N=cntvar
      SUM=sumvar;
RUN;
```

Figure 13: Grouping Data Using PROC SUMMARY

COMBINING DATA

The combining of data sets is typically referred to as a “merge” in SAS terminology and as a “join” in SQL. The figures below help describe the types of joins discussed within this paper. The Inner join, and Left, Right, and Full Outer join.

INNER JOIN
An inner join returns a result table for all of the rows in a table that has one or more matching rows in another table. This is often called an Equi-Join as the values within the joining variables must have values that are equal to each other in both tables.

FULL OUTER JOIN
A Full Outer join has all the rows for which the sql-expression is true, plus rows from each table that do not match any row in the other table. Outer joins are inner joins that have been augmented with rows that did not match with any row from the other table.

LEFT OUTER JOIN
A left outer join, specified with the keywords LEFT JOIN and ON, has all the data for which the sql-expression is true, plus observations from the first (LEFT) table that do not match any observation in the second (RIGHT) table.

RIGHT OUTER JOIN
A left outer join, specified with the keywords RIGHT JOIN and ON, has all the data for which the sql-expression is true, plus observations from the second (RIGHT) table that do not match any observation in the first (LEFT) table.
INNER JOIN

An inner join is basically a query where the values of the columns being compared are equal. All other records are discarded. In the DATA step, an inner join can be performed by executing the MERGE statement. The IN= data set option creates a variable that indicates which table the data came from. The IF statement tells the SAS System to keep all records that are in both the “left” and “right” tables. The FROM clause in the SQL statement contains two tables separated by a comma indicating a join and the WHERE clause indicates the values to be equal in both tables.

LEFT JOIN

A left join indicates that all records should be kept from the first or left hand table and only those that match from the right hand table. Note the slight difference in the IF statement in Figure 16 as compared to Figure 14. The SQL procedure is quite different. In order to perform a left join the keywords LEFT JOIN must now appear. The ON clause tells SAS which variables to match on. Another difference, though not required, is the use of table aliases; see T1 and T2 in Figure 17. Note the syntax of the RIGHT join is the same as the LEFT with the use of the syntax RIGHT JOIN.

FULL JOIN

The full join tells the system to keep all records from both tables and match values with equal BY or ON variables. Note the exclusion of the IF statement in the DATA step and the use of the keyword FULL in PROC SQL.
APPENDING DATA

Appending data together in a DATA step can be done by adding the additional tables to the SET statement as in Figure 20. To set data together using SQL the keywords UNION ALL are used. Note the use of the word ALL tells the system not to eliminate duplicate rows.

```sas
DATA table1j;
  SET table1j
  table2j;
RUN;
```

Figure 20: Appending Data Using the Step

```sas
PROC SQL;
  CREATE TABLE table1k AS
  SELECT * FROM table1k
  UNION ALL
  SELECT * FROM table2k;
QUIT;
```

Figure 21: Appending Data Using SQL

APPENDING DATA – NO DUPLICATES

The PROC APPEND is shown in Figure 22 to identify another means of appending data. This method is much more efficient as it only needs to read the incoming data to be appended. The subsequent SORT with the NODUPS option will eliminate the duplicate records. The use of the keyword UNION without the word ALL, see Figure 23, eliminates duplicate records in the resulting output, which may or may not be required.

```sas
PROC APPEND
  DATA=table2l
  OUT=table1l;
RUN;
```

Figure 22: Appending Data No Dupes Using PROC

```sas
PROC SQL;
  CREATE TABLE table1m AS
  SELECT * FROM table1m
  UNION
  SELECT * FROM table2m;
QUIT;
```

Figure 23: Appending Data No Dupes Using SQL

IDENTIFYING DUPLICATES

Below are two means of identifying duplicate records within a data set. Figure 24 uses a PROC SORT and the DATA step to count records within a BY group and outputs only those observations where last record in the group is greater than one, hence a duplicate. The SQL procedure shown here introduces the use of the HAVING clause, which occurs after the ‘GROUP BY’ clause has completed. Once the count of the variables within the by group is accomplished, the HAVING clause requests only those records where the count is again greater than one.

```sas
PROC SQL;
  CREATE TABLE find_dups_using_sql AS
  SELECT var1,
       COUNT(*) AS cntvar
  FROM table1
  GROUP BY var1
  HAVING cntvar > 1;
QUIT;
```

Figure 24: Identify Duplicates Using the Step

```sas
PROC SORT DATA=table1
  BY var1;
RUN;
```

```sas
DATA find_dups_using_datastep;
  SET table1;
  BY var1;
  IF FIRST.var1 THEN cntvar = 0;
  cntvar + 1;
  IF LAST.var1 AND cntvar > 1;
RUN;
```

Figure 25: Identify Duplicates Using SQL
CASE vs IF

The following is an example of how to set the values of a new variable based on the values of an existing variable. In Figure 26, the DATA step uses an IF statement and the SQL procedure uses the CASE statement as in Figure 27.

```sas
DATA table1n;
  SET table1a;
  IF var1 = 'SaS'
  THEN life = 'Good';
  ELSE life = 'Ok';
RUN;
```

**Figure 26: IF Statement Example Using the DATA Step**

```sql
PROC SQL;
  CREATE TABLE table2n AS
  SELECT *,
    CASE
      WHEN var1 = 'SaS'
      THEN 'Good'
      ELSE 'Ok'
    END AS life
  FROM table1b;
QUIT;
```

**Figure 27: CASE Statement Example Using SQL**

SUB-QUERY

A sub-query, sometimes called a nested table expression, is in fact a temporary table. An example of using a sub-query would be to identify all beneficiaries who have had a particular type of claim and then go back to get all of their claims. The first query or sub-query would identify the beneficiary number for use in the second query. In the examples listed below, think of “var3” as a filter for a type of surgical procedure and “var1” as a beneficiary number. The DATA step in Figure 28 is shown here to illustrate the equivalent process of the SQL examples.

```sas
DATA temp1;
  SET table1a;
  WHERE var3 = 'Good';
  KEEP var1;
RUN;

PROC SORT DATA=temp1 NODUPS; BY var1;
DATA table1o;
  MERGE temp1 ( IN= left )
    table1a;
  BY var1;
  IF left;
RUN;
```

**Figure 28: Sub-Query Using the DATA Step**

```sas
PROC SORT DATA=table1a; BY var1;
PROC SORT DATA=temp1 NODUPS; BY var1;
DATA table1o;
  MERGE temp1 ( IN= left )
    table1a;
  BY var1;
  IF left;
RUN;
```

**Figure 29: Sub-Query Using INNER JOIN**

```sql
PROC SQL;
  CREATE TABLE table2o AS
  SELECT *
  FROM table1b T1
  INNER JOIN
  (SELECT DISTINCT var1
   FROM table1b
   WHERE var3 = 'Good') T2
  ON T1.var1 = T2.var1;
QUIT;
```

**Figure 30: Sub-Query Using WHERE Clause**

```sql
PROC SQL;
  CREATE TABLE table2o AS
  SELECT *
  FROM table1b
  WHERE var1 IN (
    SELECT DISTINCT var1
    FROM table1b
    WHERE var3 = 'Good');
QUIT;
```

DATA STEP

The SAS code in Figure 28 creates a temporary data set, sorts the results and merges it back to the original data set to capture all relevant information based on the value of var1.

INNER JOIN

The example displayed in Figure 29 is a sub-query within an INNER JOIN. Note the alias of T2 for the temporary table. The ON clause can then be rendered as T1.var1 = T2.var1.

WHERE CLAUSE

Figure 30 demonstrates the use of a sub-query within a WHERE clause. The outcome is once more a temporary table though it is used similar to an IN list such as ("abc","def","ghi").
SAS ACCESS

The SAS/ACCESS products are data access engines that translate read and write requests from SAS into the appropriate call for a specific DBMS or file structure. This means that SAS code can be submitted against data sources such as Oracle®, DB2®, and Teradata® without having to know the engines specific SQL implementation. Please note, each Figure shown here requires a licensed version of SAS Access to Oracle and the Oracle client must also be installed on the remote machine.

LIBNAME enroll ORACLE SCHEMA=enroll PATH='hcis' USER="&uid" PASSWORD="&pwd";
LIBNAME ref ORACLE SCHEMA=ref PATH='hcis' USER="&uid" PASSWORD="&pwd";

DATA STEP

The LIBNAME statements shown here are specific to the Oracle database, Health Care Information System (HCIS) Legacy, at CMS. They are used to identify the location of data for use by SAS DATA steps and procedures. The SAS Access engine will convert the SAS code into the relative SQL code needed for the host database. Note that table and variable names have been shortened for clarity. Variable names need to be the same in a DATA step MERGE hence the RENAME option.

**PROC SQL**

As with the DATA step option, the SQL procedure uses the LIBNAME engine to access the Oracle data. The SAS SQL will be converted to Oracle SQL and passed to the database. The syntax used to access Oracle data via the LIBNAME engine is exactly the same as if accessing a SAS data set. Native Oracle SQL is a bit different as shown in the next example. Keep in mind that if a SAS function is used that does not have an Oracle equivalent all rows will be returned to the remote machine!

PASS-THROUGH

The pass-through facility will allow the SAS system to send host specific SQL to the Oracle database typically running on a much faster computer such as UNIX. The syntax for the pass-through is quite different than in the previous SQL example. There is the CONNECT statement with values similar to the LIBNAME shown above. The CREATE TABLE statement also contains the syntax SELECT * FROM CONNECTION TO ORACLE, which tells SAS the data will be retrieved from the Oracle database.

The code that follows is simply a subquery passing Oracle SQL. Note the syntax of the LEFT JOIN within Oracle is the plus sign (+) surrounded by the parenthesis, which is located within the WHERE clause. This language rule instructs Oracle to keep all of the observations from the table on the side of the equal sign where the plus sign is present. This identifies an obvious example of the varying implementations of SQL.

**Figure 31: SAS Access to Oracle Using the DATA Step**

**PROC SQL**

CONNECT TO ORACLE ( PATH='hcisprd.world' USER="&uid" PASSWORD="&pwd");
CREATE TABLE hcis3 AS
SELECT * FROM CONNECTION TO ORACLE
(SELECT T1.bene_state, T2.state_name,
SUM(T1.bene_cnt) as bene_cnt_tot
FROM enroll.bene_smry T1,
ref.state_tbl T2
ON T1.bene_state = T2.state_cd (+)
WHERE T2.state_cd = T1.bene_state
AND T1.year = '2004'
GROUP BY T1.bene_state, T2.state_name);
DISCONNECT FROM ORACLE;
QUIT;

**Figure 32: SAS Access to Oracle Using SQL**

**PROC SQL**

CONNECT TO ORACLE ( PATH='hcisprd.world' USER="&uid" PASSWORD="&pwd");
CREATE TABLE hcis3 AS
SELECT * FROM CONNECTION TO ORACLE
(SELECT T1.bene_state, T2.state_name,
SUM(T1.bene_cnt) as bene_cnt_tot
FROM enroll.bene_smry T1,
ref.state_tbl T2
ON T1.bene_state = T2.state_cd (+)
WHERE T2.state_cd = T1.bene_state
AND T1.year = '2004'
GROUP BY T1.bene_state, T2.state_name)
QUIT;

**Figure 33: Oracle Pass-Through Using SQL**
SIMPLE INDEX

Indexing large data sets can often result in much faster query times, especially if the result of the query is less than 15% of the total number of observations within the table. The method of creating a simple index in traditional SAS code is by using a PROC DATASETS as in Figure 34. The SQL counterpart is shown opposite.

<table>
<thead>
<tr>
<th>PROC DATASETS lib=work; MODIFY table1a; INDEX CREATE var1; QUIT;</th>
<th>PROC SQL; CREATE INDEX var1 ON table1b (var1); QUIT;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 34: Simple Index Using PROC DATASETS</td>
<td>Figure 35: Simple Index Using SQL</td>
</tr>
</tbody>
</table>

COMPOSITE INDEX

A composite index is one where more than one variable is used. Figure 36 demonstrates the name of the index is set equal to the variables requested surrounded by parenthesis. The SQL example identifies the name of the index within the CREATE INDEX statement. Notice the variables in the SQL example are separated by commas.

<table>
<thead>
<tr>
<th>PROC DATASETS lib=work; MODIFY table1a; INDEX CREATE idxlab=(var1 var2); QUIT;</th>
<th>PROC SQL; CREATE INDEX idxlab ON table1b (var1, var2); QUIT;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 36: Composite Index Using PROC DATASETS</td>
<td>Figure 37: Composite Index Using SQL</td>
</tr>
</tbody>
</table>

MACRO VARIABLES

Another useful mechanism of the DATA step and SQL within the SAS system is the ability to dynamically create macro variables using data within a table. In the examples below, the DATA Step will grab the FIRST record in the table where the PROC SQL captures the LAST record. The _NULL_ statement instructs SAS not to create a table.

<table>
<thead>
<tr>
<th>DATA <em>NULL</em>; SET table1a; CALL SYMPUT('macvar1', numvar); RUN;</th>
<th>PROC SQL NOPRINT; SELECT numvar INTO : macvar2 FROM table1b; QUIT;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 38: Creating Macro Variables Using the DATA Step</td>
<td>Figure 39: Creating Macro Variables Using SQL</td>
</tr>
</tbody>
</table>

DYNAMIC IN (LIST)

PROC SQL can also be used to string all of the values together. This is particularly useful if an IN list is needed for use within a WHERE clause. Consider the data set table2, where a list of CPT codes has been stored. This list can be placed into a macro variable and used as such: WHERE cptcode IN ( &mylist ).

PROC SQL NOPRINT; SELECT '' || cptcode || '' INTO : mylist SEPARATED BY ',' FROM table2; QUIT;

%PUT &mylist;
*RESULT = '21081','21082','21083','21084','21085';

Figure 40: Dynamic IN List Using SQL
CONCLUSION

There is almost no limit to what can be done with the Base SAS system in regards to manipulating data. Since the advent of SQL the SAS system has even more data processing power. The ability to reach out to other database vendors and often simplify coding schemes is immeasurable. In addition, using SQL within SAS provides new flexibilities with respect to multiple data sets and disparate data sources within a single program and/or data step. Moreover, a new level of acceptance is achieved by using the ability to load, manipulate, and analyze data using SQL to transcribe that knowledge to others who do not know SAS, but who are fluent in the ways of SQL. A basic knowledge of SQL can go a very long way.

REFERENCES


ACKNOWLEDGMENTS

The authors wish to thank Charles Waldron and Ann Marie Chesney for their technical and editorial assistance.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Rick Andrews  
Centers for Medicare and Medicaid Services  
Office of Research, Development, and Information  
7500 Security Boulevard  
Baltimore, MD 21244  
Phone: (410) 786-4088  
E-mail: Richard.Andrews@cms.hhs.gov

Tom Kress  
Centers for Medicare and Medicaid Services  
Center for Medicaid and State Operations  
7500 Security Boulevard  
Baltimore, MD 21244  
Phone: (410) 786-2252  
E-mail: Thomas.Kress@cms.hhs.gov

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.