

Efficient Programming with PROC SQL

Kim L. Kolbe Ritzow, Systems Seminar Consultants

Abstract

PROC SQL is a powerful new Base SAS® PROC which combines the functionality of the DATA Step and many of the Base SAS PROCs such as SUMMARY, MEANS, and SORT into a single PROC step.

This paper is designed for those users with little or no prior SQL knowledge and who are interested in learning SQL as a means to more efficient and effective SAS programming. This paper will not discuss the use of SQL as an interface language to other database products, but will concentrate on familiarizing you with SQL's syntax and the efficiencies gained by using PROC SQL versus other SAS PROCs to access SAS data sets.

Introduction

Much like SAS, SQL (Structure Query Language) is commonly used to retrieve, update and report on information. Whereas SAS operates on SAS data sets, SQL normally operates on data stored in relational tables and databases.

Starting with Version 6.06 of Base SAS software there is a SQL procedure available. The SAS System's PROC SQL is very similar to the ANSI standard for SQL with a few key differences.¹

One of the reasons for adding an SQL procedure to the Base SAS software was to make the learning of the SAS language easier for those individuals who already knew SQL. Someone who already knew SQL could now, just as easily, know SAS and double their programming power. Another important reason is that an SQL procedure can often obtain the same results as DATA and PROC steps, but with fewer and shorter statements. In some cases it uses fewer resources than the conventional SAS DATA and PROC Steps.

For those of you who have never seen, nor never used SQL, its syntax may look very foreign to you. It is, however, a very easy language to learn and use. There are several

advantages to learning PROC SQL's syntax. First, the knowledge you gain is transferable. It will help you not only code your SAS programs, but SQL programs as well. Secondly, because of PROC SQL's dense language structure you can accomplish a lot of work with a small amount of syntax. And finally, because PROC SQL combines the functionalities of so many other SAS PROCs into a single PROC step your programs will often times be more efficient.

SELECT Statement

PROC SQL combines some of the best features of the DATA step and several PROC steps into one single procedure. This ability to condense the number of steps within our SAS programs will provide us with an enormous amount of power in relatively few statements. One of the primary statements in PROC SQL is the **SELECT** statement. The **SELECT** statement describes how the report will look.

PROC SQL options;
SELECT SQL-clauses;

Through SQL clauses on the **SELECT** statement we can select columns, compute new columns, select records meeting certain conditions, group records together for summarization, and order the records. The **SELECT** statement's syntax is as follows:

SELECT column

FROM table-name | view-name

WHERE where-expression

ORDER BY column

GROUP BY column;

FROM Clause

The **FROM** clause specifies the table- or view-name(s) from which the information is being read. Think of it being like the **INFILE/INPUT** or the **SET** statement in the DATA step which names where the information is coming from. The very simplest of **SELECT** statements using

an asterisk (*), will select off all the variables from the specified data set on the FROM clause:

```
PROC SQL;
  SELECT *
  FROM USSALES;
```

The above statement is exactly the same as the following PROC PRINT:

```
PROC PRINT DATA=USSALES;
```

We can also limit the amount of information on the report by delimiting the desired variable(s) on the SELECT statement with commas:

```
PROC SQL;
  SELECT STATE, SALES
  FROM USSALES;
```

This, however, only controls the variables that will appear on the report, in the VIEW, or in the TABLE. It DOES NOT limit the numbers of variables read; all the variables are still read by the FROM clause before the SELECT statement is processed. Since the FROM clause supports SAS DATA set options, such as DROP= and KEEP=, the use of either of these options on the FROM clause can drastically improve performance by reading only the needed variables:

```
PROC SQL;
  SELECT *
  FROM USSALES(DROP=SALESID
  CUSTID);
```

Variations on the SELECT Clause

The SELECT statement has the unique ability to dynamically compute or create new variables:

```
PROC SQL;
  SELECT STATE, SALES,
  (SALES * .05)
  FROM USSALES;
```

This newly calculated variable will appear as an unlabeled column on the report. Therefore, it is sometimes useful to specify an alias on a newly created or computed variable:

```
PROC SQL;
  SELECT STATE, SALES,
  (SALES * .05) AS TAX
  FROM USSALES;
```

We can even associate default formats with the variables on our SELECT statement:

```
PROC SQL;
  SELECT STATE, SALES
  FORMAT=DOLLAR10.2,
  (SALES * .05) AS TAX
  FORMAT=DOLLAR7.2
  FROM USSALES;
```

It is also possible to use summary functions (also known as aggregate functions) on the SELECT statement. It's only useful to use these summary functions in conjunction with the GROUP BY statement (if the GROUP BY statement is omitted the statistics are computed for each observation, which is virtually useless). The syntax and purpose of the GROUP BY statement will be discussed later on within this paper.

```
PROC SQL;
  SELECT STATE, SUM(SALES) AS
  TOTSALES
  FROM USSALES
  GROUP BY STATE;
```

It is useful to define an alias on summary functions to name the newly computed column. Some of the many other available summary functions are the avg/mean, count/freq/n, max, and min.

This ability to group information and report various statistics on grouped data provides PROC SQL with some of the very same functionalities as other SAS PROCs such as MEANS, SUMMARY, and FREQ.

Another feature of the SELECT statement is the ability to specify a CASE expression, which can be useful in conditionally assigning a value of a new variable on the SELECT statement. For instance:

```
PROC SQL;
  SELECT STATE,
  CASE
    WHEN SALES < 100 THEN 'LOW'
    WHEN SALES < 500 THEN 'AVG'
    WHEN SALES < 1000 THEN 'HIGH'
    ELSE 'VERY HIGH'
  END AS SALESCAT
  FROM USSALES;
```

WHERE Clause

The **WHERE** clause specifies the rows to select from the information being read. It gives us the ability to subset the data as it is being read in. Compare the syntax of PROC SQL to PROC PRINT which both accomplish the same thing:

```
PROC SQL;
  SELECT *
  FROM USSALES
  WHERE STATE IN
  ('OH','IN','IL');
```

versus:

```
PROC PRINT DATA=USSALES;
  WHERE STATE IN
  ('OH','IN','IL');
```

The **WHERE** statement supports the same logical and comparison operators that the **IF** statement does in addition to other unique features such as the **BETWEEN/AND**, **CONTAINS**, **LIKE**, **IS NULL/IS MISSING**, and **= *** (the sounds like operator).

Another unique feature to the **WHERE** statement is the ability to subquery data. Subquerying, or inner queries, allows us to select information in one table based on values in another table:

```
PROC SQL;
  SELECT STATE, SALES
  FROM USSALES
  WHERE STATE IN
  (SELECT STATE
   FROM INTLSALE
   WHERE GROSS > 100000);
```

The subquery first builds a virtual table internally which consists of states that have gross sales over \$100,000 in international sales. Rather than hardcoding the state names

with over \$100,000 in international gross sales on the **WHERE** statement, the subquery provides the ability for the result of the evaluation to vary based on the data in our INTLSALE file.

ORDER BY Clause

The **ORDER BY** specifies a sort order for the data, much like PROC SORT:

```
PROC SQL;
  SELECT STATE, SALES
  FROM USSALES
  ORDER BY STATE;
```

While PROC SORT and the **ORDER BY** clause on PROC SQL are nearly comparable in terms of efficiency, PROC SQL can sort on a newly computed variable more efficiently than PROC SORT. With PROC SORT, the variable must be created in a separate DATA Step prior to the SORT.

GROUP BY Clause

The **GROUP BY** processes the rows in specified groups, much like the **CLASS** statement on PROC SUMMARY. While the two are about the same in terms of efficiency, PROC SQL can group by a newly computed variable more efficiently than SUMMARY or MEANS.

```
PROC SQL;
  SELECT SUM(SALES) AS TOTSALES
  FROM USSALES
  WHERE STATE IN ('OH','IN','IL')
  GROUP BY STATE;
```

CREATE Statement

We have seen the **SELECT** statement's syntax and how PROC SQL has some of the same functionalities of PRINT, SORT, SUMMARY, FREQ, and MEANS. However, what if you want the ability to build a new SAS data set? This can be accomplished through the **CREATE** statement. The **CREATE** statement names the table or view being built. Its syntax looks like the following:

```
CREATE TABLE table-name;
CREATE VIEW view-name;
```

The **CREATE TABLE** statement will build an SQL table. An SQL table is identical to a SAS data set. Think of the **CREATE TABLE**

statement as being just like the DATA statement in the DATA step; they both name a SAS data set being built. The **CREATE VIEW** statement will build an SQL view of the data being read. The **CREATE VIEW** statement is commonly used when interfacing with another database product with SAS.

```
PROC SQL;  
  CREATE TABLE TEST AS  
  SELECT STATE, SALES  
  FROM USSALES  
  WHERE STATE IN ('IL', 'OH');
```

Additional Options

There are several options that can be specified on the PROC SQL statement. A few of the more useful options include the **INOBS=**, **FEEDBACK**, and the **NUMBER** options. The **INOBS=** option specifies the number of rows read from a single source (meaning if you were joining two tables and **INOBS=10**, there would be a total of 20 records, 10 from each data set specified). The **FEEDBACK** option expands the underlying query of a **VIEW**. The **NUMBER** options requests the **SELECT** statement to maintain a column called **ROW**, which contains the row number, or as we may think of it, the observation number. The syntax of these options are as follows:

```
PROC SQL INOBS=10 FEEDBACK NUMBER;  
  ...other sql statements...
```

Efficiency Factors

PROC SQL has some distinct advantages over traditional SAS code. Because of PROC SQL's dense language structure we can accomplish a lot with a small amount of syntax. In most cases, PROC SQL will be more efficient in terms of CPU utilization than traditionally SAS code. This is usually because of the ability to combine several SAS DATA and PROC Steps into a single PROC SQL Step. In most cases efficiency can be further improved by indexing the data sets.

Another advantage of PROC SQL is its ability to do three-way merges that are indirectly related to one another and fuzzy merges more efficiently than standard SAS code. PROC SQL can perform cartesian joins, which standard SAS code cannot.

In general, PROC SQL out performs traditional SAS code in terms of a reduction in CPU utilization when performing similar tasks. While we may feel more comfortable coding traditional SAS code, it is worthwhile to learn PROC SQL's syntax as a more efficient and effective means to coding our SAS programs.

References

'SAS' Guide to the SQL Procedure, Version 6, First Edition, SAS Institute, Inc., Cary, NC, USA, pages 189-193.

Trademark Notice

SAS and Base SAS are registered trademarks of the SAS Institute, Inc., Cary, NC, USA.

Any questions or comments regarding the paper may be direct to the author:

Kim Kolbe Ritzow
Systems Seminar Consultants
Kalamazoo Office
927 Lakeway Drive
Kalamazoo, MI 49009
Voice: (616) 345-5728
Fax: (616) 345-5793