Paper TU_09

Proc SQL Tips and Techniques - How to get the most out of your queries

Kevin McGowan, Constella Group , Durham, NC
Brian Spruell, Constella Group, Durham, NC

Abstract:

Proc SQL is a powerful part of SAS® when you have to work with large amounts of data. Proc SQL is very useful when working with relational databases but it can be hard to learn as there are some common mistakes that are not obvious to new users. This paper provides examples of ways to use Proc SQL efficiently while avoiding mistakes that can make software run much slower than it needs to be. Topics covered include joins, table aliases, common proc options and simple statistics that can be calculated with Proc SQL without having to use other procs such as freq or means. When to use Proc SQL vs. other parts of SAS will also be covered.

Introduction

SQL is the language of choice for many database programmers. One reason for this popularity is that SQL can make code smaller and easier to read by performing several different operations in 1 step. Proc SQL gives SAS programmers the power of SQL for use with both relational databases and SAS datasets. SQL should be part of every SAS programmers toolbag because of the power and flexibility it offers.

When to use SQL

There are many situations for which Proc SQL is the appropriate method of data manipulation and retrieval and should thus be included in a person's SAS code. Since Structured Query Language (SQL) is standardized and widely used, a programmer is able to gather and manipulate data stored in a variety of databases in an efficient manner. The SQL language itself is simple and yet extremely powerful. Clever coding can return complex queries of just about any data type.

What is a Relational Database?

Proc SQL is especially powerful in retrieving data stored within a relational database. A relational database is defined as follows:

> "A database system in which the database is organized and accessed according to the relationships between data items without the need for any consideration of physical

> orientation and relationship. Relationships between data items are expressed by means of tables." (www.orafaq.com/glossary/faqglosr.htm)

Proc SQL not only retrieves data, it can also manipulate data and change the values within the database itself. It has the ability to insert and/or delete rows within a database table. It can also be used when a programmer wishes to create macro variables which contain values from querying a database for later use.

Advantages/Disadvantages

Proc SQL has many advantages associated with it when compared to the SAS data step. Oftentimes the Proc SQL statement can accomplish the same task as multiple data steps. Therefore less code is required, which is generally easier to maintain in the long run. The code itself is easier to follow, which makes its understanding that much easier for a person less than familiar with SAS. SQL also uses less computer resources than traditional SAS Procs. Proc SQL has the ability to read in unsorted data and from such data produce a sorted table.

Proc SQL does contain several disadvantages when compared to the SAS data step/SAS procs. For instance, Proc SQL lacks the ability of creating a table from non relational database data (such as a spreadsheet). Data _Null_ programming within the SAS data step processing tends to be more flexible for the programmer than Proc SQL select statements.

It is left up to the individual programmer which manner of coding he/she prefers. As demonstrated above Proc SQL and SAS data steps have their own distinct advantages and disadvantages. The programmer should consider time constraints, the ultimate goal of the project, and even how comfortable he or she is with either method when deciding upon the appropriate method to use.

Common SQL Statements

The four most commonly used types of SQL statements are the select, insert, update and delete statements. The following SAS code examples show the syntax for each:

Select:

The select statement enables the SAS programmer to literally 'select' columns and rows from database tables or SAS datasets. The select statement is composed of two basic components: what you want to select and where you want to select it from. The result of this Proc SQL code will be a SAS dataset which is the same as a SAS dataset created by a data step or any other proc. The code which follows is a simple example of the select syntax:

```
Proc SQL;
        Create table doses as
        Select dose, sex, species
         from all_doses;
```

This statement creates a SAS dataset called doses that will contain only the variables dose, sex, and species.

If all the variables in a table need to be selected then the '*' syntax is used in the select statement which is simpler than listing every variable in the table or SAS dataset:

```
Proc SQL;
    Create table doses as
    Select *
     from all_doses;
```

Insert:

The insert statement enables the SAS programmer to add a row to a table or SAS dataset. The code which follows is a simple example of the insert syntax:

```
Proc SQL;
        Insert into animals
        Values(a, b, c, d, e);
```

In this example the number of variables listed in the values part of the SQL statement needs to equal the number of variables in the table or dataset. If less values are listed than the number of variables the variables not listed will be blank.

Update:

The update statement enables the SAS programmer to modify data in a table or SAS dataset. The update command can update a specific row within a table. The command also has the ability to update all the rows within a table. When using update it is very important to remember to include a 'where' clause, otherwise the database may be modified in a manner not desired by the programmer. Omission of a where clause will cause all the rows within the table to be effected. The code which follows is a simple example of the update syntax:

```
Proc SQL;
        Update protocols
        Set sex='M'
        Where type='Chronic';
```

Delete:

The delete statement enables the SAS programmer to remove one or more rows from a database table or SAS dataset. As is the cause with the update statement, it is extremely important to include a where clause with this code. An omission of a where clause would cause the removal of all the rows within a table. The code which follows is a simple example of the delete syntax:

```
Proc SQL;
        Delete from treatments
        Where treat=0;
```

SQL Example

The following SAS code is a real life example of Proc SQL. The database being queried contains micro-array data from a commercial client. Please pay particular attention to the where clause at the end of the program. This 'where' clause, similar to a SAS data step or proc where clauses, tells SAS to subset the query if certain conditions are met. :

```
proc sql;
create table received as
SELECT Lab As Lab,
        Protocol,
        SampleType As Type label='Sample Type',
        COUNT(ClientSamples.ClientSampleID) As Received label='# Samples
Received'
FROM LIMS.ClientSamples
Where ClientSamples.BatchID =Batches.BatchID
and ClientID = 'EA04049'
GROUP BY Lab, Protocol, SampleType;
quit;
```

The above query creates a SAS dataset called received. The dataset contains counts and grouping variables. The output of this SQL statement is very similar to the output that would be produced with a proc means statement with a by clause. The where clause causes the SAS code to only return those records the end user is interested in, rather than returning a plethora of data he or she may not need.

The SAS where clause performs a similar function as demonstrated in the following code below:

```
sub=where(table,'x=1 and y<26');
```

Variable ‘sub’ is given values only in those instances were x is equal to one and y is less than twenty six.

The WHERE clause is an essential part of any program involving SQL code. It can prevent the accidental over-writing of an entire database table. It has the ability to eliminate headache by only returning those pieces of information that a programmer/end user is interested in.

SQL Joins

Since relational databases are normally comprised of more than 1 table, in most cases programs use a technique that allows the programmer to connect the tables together. In SQL the method to connect tables is called a join. A join can connect 2 or more tables together so that the data is returned, deleted, or updated as 1 set of data. A join operates very similar to the merge statement in a SAS data step.

The most basic type of SQL join is called an equijoin or an inner join. An inner join uses the following syntax:

```
Proc SQL;
  create table merged as
 select treat, removal_date  from
   Treatments, remdates
    Where  treatments.tid=remdates.rid;
```

This select query returns 2 variables, treat and removal_date and it only returns records that have the column tid in the table treatments matching rid in the table remdates.

Almost every time a query involves more than 1 table a join is needed to relate the tables together. If the join is left off the query, much more data will be returned than is needed. When 2 tables are listed in a query and not joined or not joined properly, the result is called a Cartesian product. In the above example if the tables are not joined correctly and both tables have 10,000 rows then the query will return 10,000 x 10,000 rows or 100 million rows. Returning 100 million rows may possibly cause the program to crash because there may not be enough data or disk space to handle that large amount of data.

An outer join is used when the user needs to return results from more than 1 table and where rows should be returned even if there is not matching data in both tables. An example of an outer join is:

```
Proc SQL;
  create table merged as
```

```
select treat, removal_date  from
Treatments, remdates
 Where  treatments.tid*=remdates.rid;
```

The *= is the left outer join operator and it means that all rows should be returned from the treatments table even if there is not a matching row from the remdates table.

The =* is the right outer join operator and it is used to specify that all rows should be returned from the table on the right side of the statement , in this case the remdates tables.

Some databases require a slightly different syntax for an outer join. Oracle uses the following syntax for their outer joins:

```
Proc SQL;
 create table merged as
select treat, removal_date  from
 Treatments, remdates
  Where  treatments.tid (+) =remdates.rid;
```

Subqueries and joins

Another way to join tables is to use a subquery. A subquery is a query that is embedded in another query. A simple example is:

```
Proc SQL;
 create table animals as
select treat, removal_date  from
 Treatments
  Where  treatments.tid in
    (select rats from animals where species='Rats');
```

Subqueries work by processing the inner query first (the select rats part) and then returning that data to the overall query. One limitation of subqueries is that the inner query can only return one column of data – if more than one column is listed a syntax error will be returned. In many cases a subquery is slower than a query using a join but it can be faster depending on the design of the database or indexes used in the database. It is a good idea to test a subquery versus a join query to see which one is faster.

Column and table aliases

Another useful aspect of SQL is the use of aliases to change the name of a variable or table. This is normally done if the database name for the variable or table is confusing or conflicts with another name already in the program. It can also be used if the table or variable name needs to be used more than 1 time in the SQL statement.

```
Proc SQL;
  create table merged as
 select treat, removal_date  from
   Treatments as treat, remdates
    Where  treat.tid (+) =remdates.rid;
```

In the above example the database table treatments is aliased to treat. Note that in the where clause the alias name treat must be used rather than the database name. Oracle allows the user to leave out the "as" in the above SQL statement. To use an alias for a column or variable name the following syntax is used:

```
Proc SQL;
  create table merged as
   select treat as new_treat, removal_date  from
   Treatments as treat, remdates
    Where  treat.tid (+) =remdates.rid;
```

This example shows a case where an alias is used because the same table shows up in the SQL statement twice since it is joined to 2 different tables:

```
Proc SQL;
  create table merged as
   select treat as new_treat, removal_date  from
   Treatments as treat, treatments as orig_treat, remdates,stdates,
    Where  treat.tid (+) =remdates.rid and
    Orig_treat.tid=stdates.sid;
```

Summarizing and counting data

One advantage SQL has over a SAS data step is that SQL can retrieve data from a database, merge it, and perform mathematical operations on it in 1 step rather than having to use a SAS data step along with another proc such as proc means or proc freq. A basic example of counting the number of records in a table is:

```
Proc SQL;
  create table counts as
   select count(*) as obscount,  from
   Treatments ;
```

This example simply counts the number of records in the table treatments. The COUNT function counts the number of records and (*) is the syntax that tells SQL to count all the records.

A better example using a subsetting where clause would be:

```
Proc SQL;
  create table counts as
   select count(*) as obscount,  from
   Treatments
   where treat_text='Control' ;
```

This example counts the number of records in the table where the treatment is control. Both of the previous examples use the alias obscount to name the variable with the count. They both return only 1 record.

The following example uses another SQL function, AVG, to calculate the average weight of all animals that have a treat_text equal to control.

```
Proc SQL;
  create table average as
   select avg(weight) as an_weight,  from
   animals
   where treat_text='Control' ;
```

There are other mathematical functions in SQL that can be used the same way as the AVG and COUNT functions above. An important note is that when using a function such as AVG only records with nonblank values for the variable the function is being used on are counted in the average. This is similar to SAS procs that only process observations for a variable if the value of that variable is not equal to blank (or '.' in SAS terms.)

Calculated fields

The use of calculated fields in SQL is an area where SQL can lead to less coding than standard SAS code. The reason for this is that a new variable can be created at the same time data is summarized or subset or sorted. A basic example of a calculated field is:

```
Proc SQL;
  create table average as
   select weight*10 as new_weight,  from
   animals
   where treat_text='Control' ;
```

This example creates a new variable called new_weight that is 10 times weight. It is also possible to create a calculated field using text variables such as this example:

```
Proc SQL;
```

```
create table species_plus as
 select  species+' ' + sex as spec_sex   from
 animals;
```

The + operator in this example is for concatenation - the equivalent to || in a SAS data step. It's also possible to use functions to trim the length of variables within the SQL statement at the same time a calculated field is being created as is shown in this example:

```
Proc SQL;
 create table species_plus as
 select  species+' ' + TRIM(sex) as spec_sex   from
 animals;
```

The difference between this example and the previous one is that the variable sex is trimmed before it is concatenated to the variable species.

Grouping and Sorting Data

SQL has the ability to group and sort data at the same time it is being retrieved from the database or SAS dataset – there is no need to have a separate sorting step which some SAS procs require before processing data. A simple example of sorting data is:

```
Proc SQL;
 create table group_sorted as
    select  dose from
    animals
   order by dose;
```

In most cases the grouping is done along with calculations so that the result is similar to doing a SAS proc such as means with a by group statement

```
Proc SQL;
 create table group_counts as
    select  dose, count(*) as num_animals from
    animals
   group by dose;
```

The result of this query will be a count of the records by the variable dose. There is no need to perform a separate sorting step before the counts are calculated.

In order to filter the query the having syntax must be used because the where clause only works on individual records, not groups. An example of the above query with a filter for groups where the count is greater than 1 is:

```
Proc SQL;
  create table group_counts as
   select  dose, count(*) as num_animals from
    animals
    group by dose
    having count(*) > 1;
```

There where clause can be used along with the having clause as shown in the following example where the data is first subset to only include males and then counted by the variable dose:

```
Proc SQL;
  create table group_counts as
   select  dose, count(*) as num_animals from
    animals
    where sex='Male'
    group by dose
    having count(*) > 1;
```

Even though it's not required by standard SQL syntax rules, it's always a good idea to add the order by syntax to any query that has a group by statement. The above example with the added order by clause would be:

```
Proc SQL;
  create table group_counts as
   select  dose, count(*) as num_animals from
    animals
    where sex='Male'
    group by dose
    having count(*) > 1
    order by dose;
```

Conclusion

Knowing Proc SQL is a very valuable skill for any SAS programmer. SQL can simplify SAS code by performing multiple steps in 1 proc rather than using a combination of data steps and procs. Knowledge of SQL can also be used later if the programmer needs to write programs for a database system as well as SAS.

Acknowledgments

Kevin would like to thank Pat Crockett of Constella Group for advice on this project and Kevin would also like to thank his wife Lib for help with editing.

This research was supported by the National Institute of Environmental Health Sciences under contract number GS-10F-0351K

Contract Information

Kevin McGowan
Constella Group
2605 Meridian Parkway Durham, NC 27713
Phone: (919) 313-7554
Email: kmcgowan@constellagroup.com
Web: www.constellagroup.com

Brian Spruell
Constella Group
2605 Meridian Parkway Durham, NC 27713
Phone: (919) 313-7673
Email: bspruell@constellagroup.com
Web: www.constellagroup.com