ADVANTAGES AND PITFALLS OF THE SQL PROCEDURE

Patricia L. Gerend, Genentech, Inc., South San Francisco, California

ABSTRACT

SQL and traditional SAS® software are both capable of defining, manipulating, and printing data. While the use of SQL can result in more concise program statements when merging, renaming, sorting, printing, and obtaining descriptive statistics, traditional SAS software is a much better choice for more powerful statistical computing.

Reasons for learning the SQL procedure are to simplify data manipulation, to have an independent software tool for the validation of traditional SAS code, and to learn a method of accessing data in relational database management systems.

A comparison of SQL to traditional SAS code in performing data manipulation is the focus of this paper. In addition, pitfalls involved in switching from SAS to SQL programming are highlighted.

INTRODUCTION

In institutions where SAS software is used for statistical analysis, relational database management systems (rDBMSs) such as Oracle®, Informix, and DB2® are often used to store and manage data. The language used by most of these rDBMSs is the ANSI standard Structured Query Language, or SQL.

In keeping with the increasing popularity of SQL, version 6.06 of the SAS system introduced a new procedure -- PROC SQL. This procedure provides SAS programmers the opportunity to choose between SQL and traditional SAS statements for a given data processing task.

There are several reasons for learning SQL:

- SQL simplifies the process of merging, sorting, and printing, leading to more compact code, which is especially nice when working with the SAS Display Manager System.
- SQL code used independently of and in addition to traditional SAS code serves as an excellent software validation tool.
- PROC SQL provides a mechanism for porting data from an rDBMS to the SAS system.
- Learning the SAS software version of SQL assists in learning an rDBMS version of SQL, enhancing professional development.

Numerous factors can be taken into account when selecting a traditional SAS or a SQL algorithm, as will be demonstrated in this paper. While both methods provide capabilities for data definition (creating, defining, and altering files), data manipulation (combining, changing, and operating on values), and printing, their approaches differ in that traditional SAS software employs procedural logic while SQL uses set theory.

In this paper simple pharmaceutical data sets are used as sample data to demonstrate the use of both SQL and traditional SAS code to answer data queries. The two types of code are shown side-by-side to enhance assessment of the differing approaches. Both advantages and pitfalls of the SQL procedure are discussed.

SAMPLE DATA

Three sample data sets are listed below. DEMOG contains basic demographic information on pharmaceutical patients, ADV has the adverse events experienced by these patients, and THES acts as a thesaurus that translates patients' adverse events into standard terms that can be summarized.

DEMOG

patient	age	sex	race	birthdt	diagnos
101	67	f	Asian	01MAR26	the heebie-jeebies
201	52	m	Asian	12AUG41	cooties

301	38	m		25SEP55	ufo sightings syndrome
401	25	f	Black	16OCT68	the creepy crawlies

ADV

patient	adv	advstd	startdt	enddt
101	none			
201	redness		15JAN93	17JAN93
201	felt sick	nausea	10JAN93	20JAN93
201	head splitting	headache	21JAN93	21JAN93
201	felt nauseous	nausea	25JAN93	28JAN93
401	sick stomach	nausea	03FEB93	10FEB93

THES

verbatim	standard
head splitting	headache
felt nauseous	nausea
felt sick	nausea
sick stomach	nausea
unable to get to sleep on time	insomnia

REQUESTS / DATA QUERIES

Several examples of data queries are shown below involving data definition, manipulation, and printing of the sample data. The SQL algorithm is shown on the left, with the corresponding traditional SAS code on the right.

1. *Create a table (data set) of female patients.*

```
proc sql;
     create table females as
     select patient
          from demog
          where sex='f';
```

```
data females (keep=patient);
          set demog;
          if sex='f';
run;
```

The SQL procedure uses the CREATE clause to create a table from existing data and the WHERE clause to subset the original data. As with the traditional SAS code, you can create a saved file with

the SQL procedure simply by putting a lib ref before the table (data set) name. Data sets and tables are virtually identical except that tables have no observation numbers and thus cannot process the POINT= command. Note that RUN statements are extraneous with the SQL procedure.

2. *Assign a race of 'White' to patient 301.*

```
proc sql;
      update demog
      set race='White'
            where patient=301;
```

```
data demog;
      set demog;
      if patient=301 then race='White';
run;
```

In the SQL procedure, a value change is made with the UPDATE, SET, and WHERE clauses; traditional SAS code uses a DATA step, a SET statement, and an IF statement.

3. *List all patients who had an adverse event.*

```
proc sql;
      title 'Patients Having an AE';
      select distinct patient
            from adv
            where adv ne 'none'
            order by patient;
```

```
proc sort data=adv (keep=patient)
                  nodupkey   out=adv2;
      by patient;
      where adv ne 'none';
run;

proc print data=adv2;
      title 'Patients Having an AE';
run;
```

```
      ----------------
```

```
Patients Having an AE

      PATIENT
      -------------
      201
      401
```

```
Patients Having an AE

      OBS    PATIENT

      1      201
      2      401
```

Here we see a sneak preview of examples to come showing the relative compactness of SQL code. Only one SQL procedure is required to subset, compact, sort, and print, whereas traditional SAS code requires one procedure to subset, compact, and sort, and a second to print. In the SQL procedure compacting records is accomplished by using the word DISTINCT, with the SORT procedure using the words NODUP or NODUPKEY. Note that SQL listings have variable headings underlined and do not contain observation numbers, although you can obtain Row numbers by using the NUMBER option in the PROC SQL statement.

4. *List the number of patients by sex.*

```
proc sql;
      title 'Number Patients by Sex';
      select sex, count(patient) "NUMBER"
            from demog
            group by sex
```

```
      proc freq data=demog noprint;
            tables sex / out=demofreq;
      run;

      proc print data=demofreq
```

```
        order by sex;
```

Number Patients by Sex

SEX	NUMBER
f	2
m	2

```
        (rename=(count=number));
        title 'Number Patients by Sex';
        var sex number;
run;
```

Number Patients by Sex

OBS	SEX	NUMBER
1	f	2
2	m	2

Frequency counts are obtained in the SQL procedure with a COUNT function and a GROUP BY clause. Numerous other functions exist, such as AVG, STD, and SUM, but more powerful statistics are not available. With traditional SAS code, summary counts can be generated with a FREQ or MEANS procedure, but additional work is needed to produce a quality listing.

5. *List patients who are not Asian.*

```
proc sql;
      title 'Non-Asian Patients';
      select patient
            from demog
            where race ne 'Asian'
               or  race is null
            order by patient;
```

Non-Asian Patients

PATIENT
301
401

```
proc sort data=demog (keep=patient)
            out=demog2;
            by patient;
      where race ne 'Asian';
run;

proc print data=demog2;
      title 'Non-Asian Patients';
run;
```

Non-Asian Patients

OBS	PATIENT
1	301
2	401

This example illustrates an extremely important difference between SQL and traditional SAS code. SQL does not see missing values; they are treated as non-existent entities. SAS software, however, sees missing values as values that happen to be null. Thus, in SQL, the meaning of "not equal to 'Asian'" is limited to values that exist and are not equal to "Asian"; to include missing values in the result, an additional clause specifying the null condition is required. In SAS code,"not equal to 'Asian'" means either a value exists that is something other than "Asian" or that the value is missing.

6. *List all adverse events according to age.*

```
proc sql;
      title 'AEs According to Age';
      select d.age, a.adv
            from  demog d,
```

```
proc sort data=demog (keep=patient age)
                  out=demog2;
      by patient age;
run;
```

```
        adv a
    where d.patient=a.patient
    order by d.age, a.adv;
```

```
proc sort data=adv (keep=patient adv)
                    out=adv2;
     by patient adv;
run;

data advage (keep=age adv);
     merge demog2 (in=d) adv2 (in=a);
          by patient;
     if d and a;
run;

proc sort data=advage;
     by age adv;
run;

proc print data=advage;
     title 'AEs According to Age';
run;
```

```
AEs According to Age

AGE   ADV
--------------------
25    sick stomach
52    felt nauseous
52    felt sick
52    head splitting
52    redness
67    none
```

```
AEs According to AGE

OBS   AGE   ADV
1     25    sick stomach
2     52    felt nauseous
3     52    felt sick
4     52    head splitting
5     52    redness
6     67    none
```

To answer this question, both the DEMOG and the ADV files must be combined, sorted, and printed. In SQL, a WHERE clause connects key values to join the tables (refer to example 9. for complex joins). Variables are given a second-level name to distinguish their source tables. These tables are given a table alias ("d" for DEMOG, "a" for ADV) in the FROM clause so that the originating table can be referred to easily rather than having to precede each variable name with its full table name.

The SAS DATA step uses MERGE and BY statements to combine the files, then sorts and prints in separate steps. The compactness of SQL code is demonstrated convincingly here in that only 7 lines are required, whereas traditional SAS code uses 4 procedures and 1 DATA step with more than double the number of lines.

7. *List all patients entered so far into the database.*

```
proc sql;
     title 'All Patients Entered';
     select d.patient "PATIENT"
          from demog d
     union
     select a.patient "PATIENT"
          from adv a
     order by 1;
```

```
proc sort data=demog (keep=patient)
                    out=demog2;
     by patient;
run;

proc sort data=adv (keep=patient)
                    out=adv2;
     by patient;
```

```
run;

data allpts;
      set demog2  adv2;
            by patient;
      if (first.patient);
run;

proc print data=allpts;
      title 'All Patients Entered';
run;
```

```
----------------
```

All Patients Entered

PATIENT
101
201
301
401

```
----------------
```

All Patients Entered

OBS	PATIENT
1	101
2	201
3	301
4	401

Answering this query involves looking for all distinct patient numbers in both DEMOG and ADV. This is done easily by using the UNION operator in SQL. Items in each SELECT clause need not have identical names but must be of the same type. They are renamed identically with the column alias "PATIENT". (A rename could also have been achieved by saying "select a.patient as patient"; this rename method is recommended when creating a table to avoid potential problems caused by assigning a variable name in lower case letters.) Due to the possibility of non-identical source variable names in the tables being unioned, any sort conducted after an operation must be done by the order of the variable in the SELECT clause ("ORDER BY 1") instead of by a variable name. SQL operators automatically compact identical records unless explicitly instructed otherwise. The traditional SAS approach involves 3 PROCs and 1 DATA step and is somewhat more complicated.

8. *List patient adverse events that are unmapped in the thesaurus.*

Correlated Sub-Query

```
proc sql;
      title 'Unmapped Patient AEs';
      select a.adv
            from adv a
            where not exists
                  (select t.verbatim
                  from thes t
                  where t.verbatim=a.adv)
            order by a.adv;
```

Except Operator

```
proc sql;
      title 'Unmapped Patient AEs';
```

```
proc sort data=adv (keep=adv)
                        out=adv2;
      by adv;
run;

proc sort data=thes (keep=verbatim)
            out=thes2;
      by verbatim;
run;

data unmap;
      merge adv2 (in=a)
            thes2 (in=t
            rename=verbatim=adv));
            by adv;
```

```
select adv "ADV"
      from adv
except
select verbatim "ADV"
      from thes
order by 1;
----------------
```

Unmapped Patient AEs

```
ADV
-------------------
none
redness
```

```
      if a and not t;
run;

proc print data=unmap;
      title 'Unmapped Patient AEs';
run;
      ----------------
```

Unmapped Patient AEs

OBS	ADV
1	none
2	redness

There are 2 SQL approaches to this problem: the correlated subquery and the EXCEPT operator. In the correlated subquery, a second SELECT statement is placed in a WHERE clause and linked back to the main SELECT statement with a join in the subquery WHERE clause. Note that variables being compared in a WHERE clause need not have the same name. The second method is similar to the use of UNION in the prior example. The EXCEPT operator connects two query results and keeps only records from the first query that do not also exist in the second query. As before, selected items need not have the same name, just the same type. Traditional SAS code requires 3 procedures and 1 DATA step, and also requires a rename since merges can only be conducted on variables with identical names.

9. *List out all the variables in the database.*

```
proc sql;
      create table temp as
      select *
            from demog d left join adv a
            on d.patient = a.patient;

      title 'All Database Variables';
      select *
            from temp p full join thes t
            on p.adv=t.verbatim
            order by p.patient;
```

```
proc sort data=demog;
      by patient;
run;

proc sort data=adv(rename=(adv=verbatim));
      by patient startdt;
run;

proc sort data=thes;
      by verbatim;
run;

data both;
      merge demog adv;
            by patient;
run;

proc sort data=both;
      by verbatim;
run;

data all;
      merge both (in=b)  thes;
            by verbatim;
      if (b) then adv=verbatim;
```

```
run;

proc sort data=all;
     by patient;
run;

proc print data=all;
     title 'All Database Variables';
run;
```

PROC SQL:

All Database Variables

PATIENT / ENDDT	AGE	SEX	RACE	BIRTHDT	DIAGNOS / STANDARD	ADV	ADVSID	STARTDT
.	.			.				.
.	unable to get to sleep on time				insomnia			
101	67	f	Asian	01MAR26	the heebie-jeebies	none		.
.								
201	52	m	Asian	12AUG41	cooties	redness		15JAN93
17JAN93								
201	52	m	Asian	12AUG41	cooties	head splitting	headache	21JAN93
21JAN93	head splitting				headache			
201	52	m	Asian	12AUG41	cooties	felt nauseous	nausea	25JAN93
28JAN93	felt nauseous				nausea			
201	52	m	Asian	12AUG41	cooties	felt sick	nausea	10JAN93
20JAN93	felt sick				nausea			
301	38	m		25SEP55	ufo sightings syndrome			.
.								
401	25	f	Black	16OCT68	the creepy crawlies	sick stomach	nausea	03FEB93
10FEB93	sick stomach				nausea			

Traditional SAS:

All Database Variables

OBS	PATIENT	AGE	SEX	RACE	BIRTHDT	DIAGNOS	VERBATIM
1	.	.			.		unable to get to sleep on time
2	101	67	f	Asian	01MAR26	the heebie-jeebies	none
3	201	52	m	Asian	12AUG41	cooties	felt nauseous
4	201	52	m	Asian	12AUG41	cooties	felt sick
5	201	52	m	Asian	12AUG41	cooties	head splitting
6	201	52	m	Asian	12AUG41	cooties	redness
7	301	38	m		25SEP55	ufo sightings syndrome	
8	401	25	f	Black	16OCT68	the creepy crawlies	sick stomach

OBS	ADVSID	STARTDT	ENDDT	STANDARD	ADV
1		.	.	insomnia	
2		.	.		none
3	nausea	25JAN93	28JAN93	nausea	felt nauseous
4	nausea	10JAN93	20JAN93	nausea	felt sick

5	headache	21JAN93	21JAN93	headache	head splitting
6		15JAN93	17JAN93		redness
7		.	.		
8	nausea	03FEB93	10FEB93	nausea	sick stomach

This example illustrates several things: 1) the use of the asterisk to select all variables from a table(s), 2) the use of an outer join to connect tables with non-matching key values, 3) the use of temporary tables to perform an outer join between more than two tables, 4) the conciseness of SQL code, and 5) the poor legibility of SQL output when all listed variables do not fit on one line.

The outer joins performed here demonstrate a fundamental difference between DATA step merges and SQL joins. In a DATA step merge of two files, if a record in one file contains a BY value for which no match is found in the second file, that record is still included in the resulting merge. In the SQL procedure, however, if a join between two files uses only a WHERE clause to connect the keys in the two files, and if a key value in one file has no match in the second file, the record with the unmatched key value will be dropped from the resulting join. This is the reason for the above example's use of outer joins ("...FROM...LEFT JOIN...ON..." and "...FROM...FULL JOIN...ON..."). The word "LEFT" indicates that some records in the table left-most in the SQL statement (e.g., DEMOG), may contain key values that are not matched in the right-most table ADV. If the reverse situation were true, the word "RIGHT" would be used. When both tables contain unmatched key values, the term "FULL" is used. Since only two tables can be reliably connected by an outer join in one statement, a temporary table is used to connect the first two tables, then that temporary table is connected to the third table. As with many prior examples, the SQL code is considerably more concise than the traditional SAS code, although in this case not necessarily more readable.

Creating a listing with numerous variables illustrates the fact that once the number of variables being listed exceeds one line, the variable names are wrapped around the page before any data values are listed. The output becomes less readable as the number of variables increases. In contrast, the SAS PRINT procedure, while not of the most exquisite report quality, gets high marks for legibility.

CONCLUSION

The most compelling advantage of PROC SQL over traditional DATA and PROC steps is the conciseness of the code. Grouping, summarizing, compacting, sorting, and printing are all performed in one statement; numerous separate DATA and PROC steps are not necessary. Disadvantages include the lack of procedural logic, the lack of powerful statistics, the awkwardness of complex joins, and the poor legibility of listings whose variables do not all fit on one line.

When implementing SQL algorithms, programmers need to keep in mind a major difference between the SQL and the traditional SAS approaches: unless instructed otherwise, SQL excludes missing information while traditional SAS includes it. In particular, if missing values are desired when selecting records not equal to a particular value, the inclusion of missing values must be explicitly specified when using SQL, whereas traditional SAS results include them automatically. Also, if records from two tables are being joined and some key values in the two tables do not match, the inclusion of records with non-matching keys must be specified with an outer join when using SQL, while traditional SAS software includes them automatically in the merge. Fortunately, both types of code offer the programmer the flexibility to include or exclude missing information; it is simply the default approach that differs. By keeping this difference in mind, the traditional SAS programmer would do well to include PROC SQL in their repertoire of programming techniques.

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. DB2 is a registered trademark or trademark of International Business Machines Corporation. Oracle is a registered trademark or trademark of Oracle Corporation. ® indicates USA registration. Other brand and product names are registred trademarks or trademarks of their respective companies.