**Abstract**

This presentation will clarify some of the less obvious details of PROC SQL, and identify situations where SQL is useful. Real-world examples will be drawn from the Technical Support staff at SAS, and the electronic SAS-L forum. A basic knowledge of "SELECT * FROM TABLE" is assumed but anyone who uses SAS should find something of value in this tutorial.

It is presented as a series of unrelated "short stories"—I am hoping that one or more of them will alert you to a pitfall of SQL or trigger some response to potential SQL solutions for problems faced by SAS programmers every day.

The sections in the paper are:

* Towards A Better Union
* Outer Joins - more than 2 too.
* Joins whose criteria are Fuzzy
* Logical Expressions in SAS Software
* Case Free Sorting
* Let the DBMS do the work
* Data about Data

**Towards A Better Union**

Experienced DATA STEP users want to take two tables and concatenate them just like "DATA A; SET B C; RUN;" - the knee jerk SQL formulation is often:

```sql
create table a as
select * from b
union
select * from c;
```

This is usually not what was wanted. The SQL UNION has more strict set-oriented semantics than most problems call for.

Firstly, the UNION is formed on a column by column basis—the columns are not matched by name as a SAS programmer would expect. This behavior is probably a hangover from the early SQL implementations, but is now required by the SQL Standard. SQL has the additional CORRESPONDING keyword used after one of the set operators [UNION, INTERSECT, EXCEPT] that triggers matching by column name instead of column position.

Secondly, the UNION operator requires that the result set be processed to return only unique rows. In PROC SQL this process of returning unique rows is implemented with an internal sort using all the columns in the result set as the sort key. Unless you need this duplicate row elimination it seems a shame to pay for it.

You probably meant to say:

```sql
create table a as
select * from b
union all corresponding
select * from c;
```

The ALL allows PROC SQL to skip the inevitable sort needed to eliminate duplicate rows, and the CORRESPONDING tells it to line the columns up by name and not by position.

This is a good time to caution you not to abandon the DATA STEP. Unless the UNION is part of a larger SQL expression such that there may be some advantage to keeping the problem all in SQL, you will be better off using the DATA STEP who has been honed over the years to be the host it can be for problems of this nature.

**Outer Joins - more than 2 too.**

The normal SQL join sometimes comes as a surprise to the experienced SAS programmer. SQL by default will "drop" records that have no match during a join - The SAS programmer is used to the data step behavior that keeps these non matched rows and supplies missing values where needed. SQL can do this too - the feature is called Outer Joins and is now implemented in more and more systems that support SQL.

Consider a system that tracks People in one data set, their Payroll information in another and information on their investments in a third. There is an inherent Master-Detail relationship between these data sets. Normally there will be no payroll or investment records that do not have a matching people record. The knee jerk join of these tables is:

```sql
select *
from people, payroll, invest
where people.person = payroll.person
and people.person = invest.person;
```

This is usually wrong. It loses information for people that have either no payroll record, or no investment record - An outer join is called for.

```sql
select *
from people left join payroll
on people.person = payroll.person
left join invest
on people.person = invest.person;
```

This query retains the people that have no payroll or investment data in the result set. Some folks are not aware that they can string more than one outer join together in this fashion and resort to more complicated nested SQL queries to achieve the same effect.

The Select clause often needs attention in these kinds of queries as it is not necessary to record the key value three times, once from each table. What we really want is the persons name from the people table, and can do without it from the other tables — we already know they must be the same for a match to occur. Unfortunately, it is not easy to "drop" payroll.person and invest.person after the join has happened, so you are forced to code something like:
ANSI SQL does define a syntax that reduces this burden called a Natural Outer Join. Look for PROC SQL to implement this in a later release.

**Full joins vs. Left/Right joins.**

Sometimes the Master Detail relationship between the tables is not as well defined. Imagine a "messy" database where we had payroll and investment records, but no person record to match. A left join is no longer appropriate, as it only retains rows from the left-hand table that have no match in the right hand one.

```sql
select coalesce(people.person, payroll.person, invest.person) as person,
       people.var1, people.var2, payroll.var1, payroll.var2, invest.var1, invest.var2
from people full join payroll
     on people.person = payroll.person
     full join invest
     on coalesce(people.person, payroll.person) = invest.person;
```

Now! That got complicated fast. One cannot be sure whether the person code came from the people, the payroll or the investment dataset and have to use the coalesce function to select it from the proper contributor. I suggest using a temporary view to make this join clearer to the poor programmer that will have to maintain this code when you move on to a management job.

Create view peo_pay as

```sql
select coalesce(people.person, payroll.person)
       as person,
       people.var1, people.var2, payroll.var1, payroll.var2
from people full join payroll
     on people.person = payroll.person
```

```sql
select coalesce(people.person, payroll.person, invest.person)
       as person,
       peo_pay.var1, peo_pay.var2, invest.var1, invest.var2
from peo_pay full join payroll
     on peo_pay.person = invest.person;
```

Some join requests are couched rather vaguely. I'm sure you have all had a boss that said "I just want the best match". A SAS software user posed the following problem to the SAS-L BITNET electronic discussion forum:

What I have been trying to do is match kids referred for behavior problems with those who haven't on the basis of sex, age, SES and race.

The first two (sex age) to be exact matches. This is easy: I pick subsets of the data sets which are the right age and sex. What next? If clinical kid 1 has an SES of 1 (a three-level recording) I try to find a non-clinical kid who also has an SES of 1 and a race that matches the race of the clinical kid (1 = white 2=black 3=other).

If I can't find an exact match, I back off the tolerance from 0 (abs(clin.race-norm.race)) and similarly for SES.

Another user posted as follows:

Consider all possible matches for each referred kid; discard those matches which do not meet the mandatory criteria; then, if there is more than one match for a kid, pick the best one according to the other criteria. The join operation, which is central to SQL, operates (at least conceptually) in a way which lends itself to this type of problem.

```sql
proc sql;
   select clin.id as clin_id,
          norm_id as norm_id,
          1000*abs(clin.ses-norm.ses) +
          1000*(clin.race ne norm.race) +
          abs(clin.dob-norm.dob) +
          uniform(123) as penalty
   from clin left join norm
        on clin.sex=norm.sex
        and abs(clin.dob-norm.dob)<365
        group by clin.id
        having penalty=min(penalty)
;
```

The first two columns selected are the IDs of the paired kids; the third (PENALTY) is explained below.

The FROM clause uses a LEFT JOIN rather than an inner join so that a row will appear for each referred kid for whom no match can be found.

The ON clause implements the mandatory criteria. Kids are considered to be the same age if their birth dates are less than a year apart.

The GROUP BY and HAVING clauses implement the non-mandatory criteria to choose the single best match for each referred kid. This is done by quantifying the tradeoffs in the PENALTY column; the lower the value, the better the match. Each PENALTY value also includes two levels of tie-breakers to ensure unique results: these are separated from each other and from the criteria by orders of magnitude.
The first term of the PENALTY formula adds 1,000 “points” for each level of SES difference. The second adds 1,000 points if the races are different. The third term is the basic tie-breaker; other things being equal, a smaller age difference might make a better match, so one point is added for each day. The final term adds a random value to serve as the ultimate tie-breaker.

Score those joins

Perhaps your joins can be solved with this scoring technique. If you can invent the Sal expression that reflects the quality of the match as a “score” then you may be able to use this technique to select the match with the “best score.”

Logical Expressions in SAS Software

Logical Expressions in SAS have an interesting property that can be exploited by SQL. Be sure and leave plenty of comments in your source code if you do something like this so that the poor person that follows will have a clue as to your intent in SAS. Logical expressions are guaranteed to return a result that is numerically ONE or ZERO.

Logical Expressions as Join Criteria

An excellent paper was presented at SUGI 18 [Corelle...]. The problem the authors of this paper were faced with was that they were trying to join data files obtained from two different sources. The data in question were infant mortality data — one table was captured by the local state authorities and the other by the hospitals in the area.

It was decided that records from the two tables were a match if:

• The sex matched
• The birth dates were within a year of each other
• The pair of records scored 6 or more points from this table
  • 2 for a matched month
  • 2 for a matched day
  • 2 for a matched year
  • 1 for a match on surname
  • 1 for a match on first name
  • 1 for a match on initial

This logic is implemented in the WHERE clause of an SQL Join.

```sql
WHERE a.sex = b.sex
  and abs(a.birth - b.birth) <= 365
  and ( 2 * (a.month = b.month) )
  + ( 2 * (a.day = b.day) )
  + ( 2 * (a.year = b.year) )
  + ( a.surname = b.surname )
  + ( a.firname = b.firname )
  + ( a.initial = b.initial )
  >= 6
```

Be warned that these forms of joins are usually more “expensive” in terms of computer resources than traditional equi-joins. They often involve a Cartesian product — forming all combinations of rows from the contributing tables, and then evaluating the where clause against that. Each exact criteria specified helps to trim the problem space down.

Logical Expressions as Counters

A SAS user in Chicago wanted to tally the values of a variable in a SAS data set, presenting the tally for each value as a separate column of the report. The twist on the problem was that each value could be tallied into more than one category. They collected data on people and their pets — how many cats and dogs in their household, and wanted to tally cat owners, cat lovers [more than 2 cats], dog owners and dog lovers [more than 2 dogs].

```sql
data pets;
  input person $ cats dogs;
cards;
paul 5 1
linda 0 2
chris 0 2
pat 0 0
kelsey 1 0
thais 0 4
run;
```

We can use a SAS expression to decide if a person falls into any of the 4 categories.

```sql
proc sql;
  select person,
    cats > 0 as cat_own,
    cats > 2 as cat_love,
    dogs > 0 as dog_own,
    dogs > 2 as dog_love
  from pets;
```

```sql
  PERSON CAT_OWN CAT_LOVE DOG_OWN DOG_LOVE
  paul 1 1 1 0
  linda 0 0 1 0
  chris 0 0 1 0
  pat 0 0 0 0
  kelsey 1 0 0 0
  thais 0 0 1 1
```

The answers have now been categorized with the one/zero values. Since all the “hits” are now 1 and “misses” are 0, we can count the hits by adding up the columns using the SQL SUM function. Some people’s responses are a hit on more than one category — this was the tricky part of the original question that rules out using a format and PROC MEANS to solve this problem.

```sql
proc sql;
  select sum(cats > 0) as cat_own,
    sum(cats > 2) as cat_love,
    sum(dogs > 0) as dog_own,
    sum(dogs > 2) as dog_love
  from pets;
```

```sql
  CAT_OWN CAT_LOVE DOG_OWN DOG_LOVE
  2 1 4 1
```

239
Advanced Tutorials

This is not a technique that you'll use often, but it does have applications — I'm hoping you'll recognize this when you encounter a problem that could use this solution.

**Case-Fixed Sorting**

Technical Support here at SAS received several calls from people who wanted to sort their data on some criteria that was not reflected in the data directly. The caller usually wanted to sort by formatted values, but some other variations included wanting to sort in a case-insensitive fashion, and wanting to sort "special" values to the beginning or end.

Proc SQL facilitates these requests as it allows the ORDER BY clause to contain expressions like:

```sql
order by upcase(name)
order by put(variable, format.)
order by case
  when account = 999 then .
  else account
end
```

**The SQL Passthru Syntax**

The SQL Passthru syntax was added to PROC SQL after its initial release in Version 6.06 of the SAS System and I still meet people that have not been exposed to it. If you have two tables that reside in a DBMS system like DB2® or Oracle® then you have more than one option for joining those tables. You could

1. Create SAS ACCESS descriptors of the tables and then use PROC SQL to join those descriptors.
2. Create a View of the join in the DBMS, and then create a SAS ACCESS descriptor of that DBMS view
3. Use SQL Passthru to materialize the result directly into an SQL from clause

The objective is to push as much work as possible into the DBMS as its query optimizer has a much better insight into the storage structure of the tables in question, and will almost always be able to materialize the result set more quickly than transferring the individual tables into SAS and forming the result set there as is done for option 1 above.

If you would be accessing the results of this join often (or from many different programs) it makes sense to centralize the definition of the join in the DBMS (i.e. use option 2 above). On the other hand, if the usage is more casual, then coding the join inside the SAS job is often easier and keeps all the logic in one place. Here is an example of a DBMS join coded using SQL Passthru.

```sql
PROC SQL;
   ...
connect to oracle(user=scott
    password=tiger);
create table work.result as
    select * from connection to oracle
/* Oracle syntax inside the parens */

   ...
from dictionary.tables
where libname='STOCK';
```

Interesting things to note in this example are:

- one uses the Oracle dialect of SQL inside the parenthesis — the syntax is passed literally to the DBMS for interpretation (after SAS Macro references have been expanded out)
- the result of the query passed to Oracle is processed in the SAS environment just like any other component on the from clause... In our example we name this result set ORA_DATA and supply alternate column names of KEY and SUM

**Data about Data**

A recent addition to PROC SQL are the "dictionary tables". These are result sets of information about the current SAS environment. There is a table [dictionary.tables] that details all the datasets visible to the current session. This can be used to automate processing all members in a SAS data library, for example. Other dictionary tables include variable level information on SAS data sets; the current values of SAS options; the current values of SAS macro variables; the titles and footnotes in effect as well as information on external files allocated to the session.

Suppose we have a libname STOCK that points to a series of SAS data sets. Each dataset tracks the daily stock price of one company - the name of the dataset is the stock symbol for that company. We'd like to process the collection of datasets and find the stocks that have appreciated in value the most between two dates.

This example also demonstrates the newer features of the into clause. You can now place each row in the result set into a separate macro variable which is very useful when trying to process lists of things. The SQLDBMS macro variable is used to query how many rows were returned (and thus how many elements are in the list)

```sql
options print;
%macro winners(start='30dec94'd,
                   stop='29dec95'd);

First we need a list of datasets. We can ask the "dictionary.tables" virtual table for all the tables in our STOCK Library.

proc sql;
    reset noprınt;
    select memname
        into :mem1 - :mem99999
        from dictionary.tables
        where libname='STOCK';
```
%let memcount=&sqlobs;
%put NOTE: There are &memcount stocks;

Now we know how many stocks we have to process, let's go and do each one in turn. We'll define a work table "growth" to accumulate the results as we go:

create table growth
   ( symbol char(8), growth num );

Process each of the members we found in the "STOCK" library...

%do i = 1 %to &memcount;
%let symbol=&&&mem&i;

Find the price as of the start and stop dates and insert the symbol name and its growth factor into our temporary table that is accumulating the results:

select &symbol
   into : sstart
   from STOCK.&symbol
   where date=&start;
%let gotstart=&sqlobs;
select &symbol
   into : sstop
   from STOCK.&symbol
   where date=&stop;
%let gotstop=&sqlobs;

We are a little cautious to ensure that we got valid data for both the start and stop dates before recording the growth figures for a stock.

%if &gotstart & gotstop %then %do;
   insert into growth
      set symbol="&symbol";
      growth= 100 * ( &sstop - &sstart ) / &sstop
   ;
%end;
%end;

OK, now that we have all the results let us go and print those stocks whose growth came within 20% of the best one.

reset print;

'Growth Stocks';
select *
   from growth
   where growth >= ( select .8 * max(growth)
                   from growth
                  )
   order by growth desc, symbol;
%mend;
%winners;

References:

Corell, MacLaughlin, Drew and Longacre: Death Matching: an Algorithm and comparison of its implementation in PROC SQL vs. PROC SQL plus a DATA STEP [SUGI 18 Proceedings]


SAS Institute: Combining and Modifying SAS Datasets - Examples

SAS Institute: SAS Guide to the SQL Procedure - Usage and Reference