

Fun with the SQL Procedure — An Advanced Tutorial

Lewis Church Jr., SAS Institute Inc.
Paul Kent, SAS Institute Inc.

Abstract

The SQL procedure is an implementation of ANSI Standard Structured Query Language (SQL) that facilitates extracting data from multiple sources with simple and concise queries. SQL is a versatile language for expressing these queries but there is often more than one way to pose the same question — some of these ways may be less than intuitive for someone used to a procedural language.

This paper assumes some basic knowledge of the SQL SELECT Statement, and the concepts of joins and set operators that are central to SQL. The paper is a tutorial on using the more advanced SQL constructs to solve problems in unique and concise ways. Where more than one way to address a problem exists, we analyse the alternatives with an eye towards selecting the more efficient solution.

Overview

The paper covers a somewhat random selection of interesting SQL usage. The examples have been distilled from programs submitted to Technical Support and from the SAS-L electronic mailing list. Topics discussed include:

- Conditional Logic in SQL
- Set Operators and DATA Step Alternatives
- Cartesian Joins
- SQL Join strategies
- Tie-breaker Joins
- Fuzzy (Scored) Joins
- Joins with tables in an External DBMS

Conditional Logic in SQL

The conventional WHERE clause is the mechanism for excluding rows from SQL processing. This does not handle problems whose formulation includes “do this or do that”. The CASE expression in SQL allows you to choose a value based on a series of logical tests. For example, suppose that a person’s commission rate depended on their grade:

```
select case grade
      when 1 then .20
      when 2 then .10
      else 0
      end as comm,

      sales * calculated comm

from ...;
```

It might help to think of the SQL syntax starting with CASE up through the matching END as a user-defined function. This function returns the value associated with the first WHEN expression that is true.

The use of a CASE expression can simplify your WHERE clause when the matching column must be decided by some test. This next code fragment shows alternative ways of expressing a join where we would like to match with a row’s common name if that field has a value and match with given name if it does not:

```
/* LESS EFFICIENT */
where
  ( b.comname
    and a.name = b.comname )
or
  ( (not b.comname)
    and a.name = b.givname )

/* MORE EFFICIENT */
where a.name
      = case
          when b.comname then b.comname
          else b.givname
        end
```

The latter form is more precise and it has an additional advantage — it is expressed in a form that allows the PROC SQL query optimizer a better chance at performing the join efficiently. The optimizer can form a temporary column with the results of the case expression and then use any of the equi-join strategies discussed later in this paper to satisfy the query.

Another situation where the CASE expression is useful is when a table has values whose interpretation depends on the value of some other column in the table. Consider a table of transactions with a column for trantype (debit or credit) and a separate column for the amount of the transaction. Compare the usage of the CASE expression to compute the net change for each account with an alternative approach that unions the two distinct sets. Those of you comfortable with the SAS convention that logical expressions evaluate to 1 or 0 will understand the third solution

posed, but we are sure you'll agree that the CASE expression is the cleaner method.

```
select account,
       sum( case trantype
            when 'debit' then value
            when 'credit' then -value
            end
           ) as net
  from xact
 group by account
       ;

select account,
       sum(value) as net
  from (select account, value
        from xact
        where trantype = 'debit'

        union all

        select sum(-value) as net
          from xact
         where trantype = 'credit'
        )
 group by account
       ;

select account,
       sum( sum(
            ( (trantype = 'debit') * value ),
            ( (trantype = 'credit') * -value )
            )
           ) as net
  from xact
 group by account
       ;
```

Set Operators and DATA Step Alternatives

The SQL set operators UNION, INTERSECT and EXCEPT come in two flavors:

- The UNION ALL flavor has multi-set semantics — duplicate rows are not eliminated from the result multi-sets.
- The unqualified UNION flavor has traditional mathematical set semantics — duplicate rows are eliminated from the result set in the case of UNION and from the contributing datasets in the case of INTERSECT and EXCEPT.

The elimination of duplicates may be a costly process. There are occasions when the SQL query optimizer notices that the rows are already unique (a unique index exists), but most often PROC SQL will schedule an internal sort to weed out the duplicate rows.

Don't forget that the DATA Step is well suited to these types of requests. Using the IN= data set options, the SQL INTERSECT operation can be coded as:

```
data out;
  merge a(in=ina) b(in=inb);
  by col1 col2;

  if first.col2 and ina and inb;
run;
```

Note that you have more flexibility in the DATA Step — you can list just the key variables on the BY statement. In SQL, all the variables participate in the grouping process and there is no language feature provided to override this.

Cartesian Joins

The conceptual model of an SQL join is easy to understand. Simply compute all combinations of the rows from the contributing datasets and then eliminate the rows that don't "match" the where clause. A subsequent section of this paper discusses how PROC SQL can avoid this brute force approach, but sometimes this "all combinations" property is useful.

This series of examples also demonstrates an advantage of a non-procedural language like SQL over traditional tools like the DATA Step. The problem at hand starts out as a request to form the set of all combinations of the rows from two tables — like many real world problems, the request is refined

- To request only one version of each pair — include AB but exclude BA
- To perform the matchings withing by groups (groups of the same value)

The initial code for SQL is a join request with no where clause. The DATA Step solution involves looping over the records in the inner table with SET POINT= logic.

```
/* SQL */
select a.v as av, b.v as bv
  from t a, t b;

/* DATASTEP */
set t(rename=(v=av)) nobs=n;
do i = 1 to n;
  set t(rename=(v=bv)) point=i;
  output;
end;
```

Query Version 1.

If you want to eliminate the redundant pairings like AB and BA, you can change you code to make sure the value in the first pair is less than that from the second. It is possible to translate the SQL $a.v > b.v$ into a DATA Step expression, but it is more efficient to just consider a subset of the dataset that contributes the second half of the pairing.

```

/* SQL */
select a.v as av, b.v as bv
  from t a, t b
 where a.v < b.v;

/* DATASTEP */
set t(rename=(v=av)) nobs=n;
do i = _N_ to n;
  set t(rename=(v=bv)) point=i;
  output;
end;

```

Query Version 2.

Satisfying the final requirement (process the matches in groups) is simple in SQL — add the requirement that the rows be from the same group to the WHERE clause. It is possible to take this same approach for the DATA Step and add an IF expression to output only those rows from a matching group.

```

/* SQL */
select a.grp, a.v as av, b.v as bv
  from t a, t b
 where a.v < b.v
       and a.grp = b.grp;

/* DATASTEP */
set t(rename=(v=av grp=agrp)) nobs=n;
do i = _N_ to n;
  set t(rename=(v=bv grp=bgrp)) point=i;
  if agrp = bgrp then output;
end;

```

Query Version 3.

The SQL query will take fewer resources to process — The SQL Query Optimizer will recognize that the incoming data can be sorted by grp and will not even consider interactions of a row with rows outside of its group. The DATA Step code can be changed to do this, but many people would have missed this optimisation opportunity.

```

proc sort data=t;
  by grp;
run;

data res;
set t(rename=(v=av grp=agrp)) nobs=n;
more = 1;
do i = _N_ to n while(more);
  set t(rename=(v=bv grp=bgrp)) point=i;
  if agrp = bgrp then output;
  else more = 0;
end;
drop more;

```

Query Version 4.

PROC SQL allows you to concentrate on specifying what results you desire, and leaves the details of the processing up to the Query Optimizer. It is possible to code a DATA Step equivalent, but it is our contention that many folks will overlook optimisation possibilities that SQL will not. Also worth noting is that the SQL code is one third as many lines of code as the optimized DATA Step solution — this is probably a good thing unless you get paid by the line!

SQL Join Strategies

The progression of queries shown above demonstrate that PROC SQL can employ different strategies for answering the query over and above the obvious “consider the where clause for each combination of rows from the input datasets”. This brute force approach would be very expensive for large datasets.

PROC SQL can solve queries that specify an *equals* match between variables from both tables in a number of optimized ways. The SQL Query Optimizer must choose between:

- Sorting the tables and performing a match merge,
- Accessing the rows of one table sequentially, and fetching the matching rows from the other table via an index defined on that table,
- Loading the rows of the smaller table into memory and processing the rows from the other table sequentially, using table-lookup techniques to locate matching rows.

The merits of these different strategies are discussed below, followed by some hints on how you might find out which strategy PROC SQL has chosen on your behalf.

Sort-Merge Joins

The input datasets are processed in sort-group sized chunks. SQL does not need to consider each combination of record matches, just the matches with rows in the same sort-group. This method requires that the input data be sorted in sort-group order — or be marked as sorted in the SAS Dataset header information. Sort Merge Joins are good performers when the bulk of the dataset is being joined.

Indexed Joins

If one of the input datasets has an index on the variables used as the join key, it is possible to access the matching records directly via that index. This method requires that you define an index and performs well when accessing a small fraction of the total number of records in a dataset.

In-memory Joins

The smaller input dataset may fit into memory. In this case, PROC SQL can load the rows into an in memory table that provides fast access to the matching row given the join key values. A disadvantage of this method is that it requires that you have enough memory to store the smaller dataset.

So Many Choices...

How does PROC SQL choose from all these methods? We calculate the relative sizes of the input datasets and make an "educated guess" using these heuristics. We choose:

Indexed	If there are any candidate indexes
Merge	If the datasets are already sorted in a convenient sort order
Hash	If one of the datasets will fit into memory
otherwise	PROC SQL will choose to sort the incoming datasets and perform the Sort Merge algorithm to process the join.

What did SQL choose?

There is no simple way to tell. Other Data Base Management Software has features that explain the query strategy chosen by the DBMS. We hope to implement this in PROC SQL in a future release.

The SAS System option `msglevel=i` will display an informatory note when an index is selected for where clause or join processing. There is also an undocumented `_method` option on the PROC SQL statement that will display an internal form of the query plan, by showing the hierarchy of processing that will be performed. The module codes used in this display are:

<code>sqxslct</code>	Select
<code>sqxjm</code>	Merge Join
<code>sqxjndx</code>	Index Join
<code>sqxjhsh</code>	Hash Join
<code>sqxsort</code>	Sort
<code>sqxsrc</code>	Source Rows from table
<code>sqxfl</code>	Filter Rows
<code>sqxsumg</code>	Grouped Summary Statistics
<code>sqxsumn</code>	Summary Statistics (not grouped)

For example, if one of the contributing datasets is sorted on key then the SQL:

```
proc sql _method;
  select *
    from a,b
   where a.key = b.key;
```

would display the query plan below in the SAS Log file. The SELECT module (`sqxslct`) gets its input records from the Merge Join Module (`sqxjm`), which gets its input from two sources. The first source is passed through a sorting process, while the second is not — it is already sorted in a useful order.

NOTE: SQL execution methods...

```
sqxslct
  sqxjm
    sqxsort
      sqxsrc
    sqxsrc
```

If neither contributing dataset is sorted, but one of them does fit into memory, PROC SQL will choose to process the query with a Hash Join (`sqxjhsh`), as shown below:

NOTE: SQL execution methods...

```
sqxslct
  sqxjhsh
    sqxsrc
  sqxsrc
```

Fuzzy (Scored) Joins

Suppose that you are required to match up records that have been captured independently — match public records stored in one format with company records stored in another. You would like to select a short list of possible matches, recognizing that it is unlikely that the records will match on all criteria. As long as records match on some criteria, you would like them included.

One approach to a problem of this nature is to assign a score to each kind of match, and select records that meet a certain score. An example could be "Select records that have the same value for KEY and match on two (or more) of these four variables W, X, Y, Z.

```
select *
  from a, b
 where a.key = b.key
    and sum ( a.w=b.w,
              a.x=b.x,
              a.y=b.y,
              a.z=b.z
            )
        >= 2;
```

This logic relies on the SAS convention that a boolean expression has the value 1 if the expression is true, and 0 otherwise. If the sum of the four boolean expressions is greater than or equal to 2, then 2 or more of those expressions must have been true.

Tie Breaker Joins

A query related to the fuzzy join shown above is one where you would like to select the "best" match from a pool of potential matches. In this example, we would like to select records with the closest Y value. If there are ties for the closest Y, then choose the record with the closest Z — If there are ties in this category then pick one at random.

One can define an expression that computes a distance (or penalty) measure, and then select the combination with the smallest distance measure. In this example, we bias differences in Y by a factor of 1000, and differences in Z by a factor of 10. The "proper" value for these bias factors depends on the range of your data values.

```
select *,
       1000 * abs(A.Y - B.Y)
     + 10 * abs(A.Z - B.Z)
     + uniform(0)
       as penalty
  from A left join B
    on A.X = B.X
 group by A.X
 having penalty = min(penalty)
 ;
```

Joins with tables in an External DBMS

SAS/ACCESS[®] software allows a user to create views of tables in external DBMSs. When they use one or more of these access descriptors as table names in a PROC SQL query, many people make the (incorrect) leap of faith that lulls them into thinking that the entire SQL statement will be passed to the DBMS. This is not the case — PROC SQL must process the query just as if the tables are SAS data sets (and even worse, as SAS data sets that have no indexes).

A typical problem that a user might face is:

- I have a large data set in the DBMS called A. It is indexed in the DBMS on the key value A.X
- I have a smaller data set (usually a SAS data set) called B. I want to extract the rows from A that have A.X = B.X

This problem can be solved with the simple PROC SQL join shown below. In many cases this is an adequate solution as the data volumes are not large enough to warrant the extra effort required to perform this join more efficiently.

```
proc sql;
  select * from A,B
  where A.X = B.X;
```

The DBMS knows that A.X is an indexed column, but this information is not surfaced through the access descriptor, so PROC SQL must resort to another strategy for processing the join.

It will most likely choose to do a hash join. It will load all the rows of the small table into an in-memory lookup data structure, retrieve each row from the database and check to see if a match is found in the small table. This process is expensive, as we must extract *all* the rows from the large DBMS table.

There are a number of ways to attack this problem that avoid transferring all the rows from A in the DBMS into the SAS system before subsetting them.

- Move the small table to the DBMS and perform the join there.
- Construct a list of the key-values in the small table, and use that to construct a where clause to be passed into the DBMS.
- Use a side effect of the MODIFY statement in the DATA step.

Move the small table to the DBMS

This code shows how one can load a SAS data set into the DBMS and then use the SQL Passthru feature to execute the join in that DBMS. This example uses DB2[®] from IBM, but the principle applies to all SQL-based data bases supported by SAS/ACCESS software.

```
data b;
  do x = 1 to 100000 by 1000;
    y = 'data';
    z = 'other data';
    output;
  end;
run;

proc dbload dbms=db2 data=b;
  limit = 200000;
  commit = 20000;
  table = b;
  type x = 'int';
  load;
run;

proc sql;
  create table result as
  select * from connection to db2
  (
    select a.x,a.y,a.z, b.x,b.y,b.z
      from a,b
     where a.x = b.x
  )
  as r(ax,ay,az,bx,by,bz);

%put sqlxrc=&sqlxrc;
%put sqlxmsg=&sqlxmsg;
quit;
```

This strategy permits the DBMS the most freedom in choosing a better join strategy — if the size of B, relative to A, is small enough then there should be no full pass of A. You should experiment with this method and use the DBMSs explain facilities as you may just be moving the locus of work from your SAS execution to the DBMS execution for no net gain (unless someone else pays for the DBMS cycles!)

Creating new tables in the DBMS is often a point of contention between users, their DBA's, and the DBMS system catalogs. An alternative is to have a predefined, but empty table for the keys of B in the DBMS that you populate with PROC APPEND, perform the join, and then remove the rows from B in preparation for the next time.

Construct a list of the keys in the small table

The SAS Sample Library has a member that demonstrates this approach. Ask your SAS Software Consultant how to find the SQLJMAC member of the SAS Sample Library. Using this approach requires two steps:

- Compute the unique keys and store them in macro variables using the %PREJOIN macro.
- Form the keys computed above into a valid where clause to be sent to the DBMS using the %KEYJOIN macro.

```
/*-----*/
/*- load in the macros -*/
/*-----*/
%inc 'SAS.SAMPLES(SQLJMAC)';

/*-----*/
/*- compute the keys in B -*/
/*-----*/
%prejoin(table=B, key=X, keytype=N);

/*-----*/
/*- and use those keys to limit -*/
/*- the rows returned by the -*/
/*- DBMS. -*/
/*-----*/
proc sql;
  select * from A.B
  where A.X = B.X
  and %keyjoin(table=A);
```

You can also use a variation of the %KEYJOIN macro with the SQL Passthru facility, but you would have to change the macro as it emits character constants surrounded by double quotes, and many DBMSs interpret that as a column name.

This scheme becomes expensive when there are many different key values in the smaller data set as it creates a new macro variable for each one.

Use a side effect of the MODIFY statement

The MODIFY statement is a new statement in the DATA step for Release 6.07 TS301 of the SAS System. It processes each row of the transaction data set and dynamically constructs a where clause of the form KEY=<value> for the master data set. Usually one modifies the record from the master data set in some way and replaces it, but you don't have to do that — This example simply saves the master record with OUTPUT;

```
data a wanted;
  modify a b;
  by x;

  if _iorc_ = 0 or _iorc_ = %sysrc(_DSEMR)
  then do;
    output wanted;
    _error_ = 0;
  end;

/*-----*/
/*-- this cancels the implicit REPLACE --*/
/*-- that happens at the end of the --*/
/*-- data step loop. --*/
/*-----*/
return;
run;
```

Trademarks

SAS and SAS/ACCESS Software are registered trademarks of SAS Institute, Inc.

DB2 is a registered trademark of I.B.M Corporation.

Your Turn

The authors can be reached care of SAS Institute Inc., or at the Internet E-mail Addresses: *kent@unx.sas.com* and *saslcj@unx.sas.com*

Comments on this tutorial and PROC SQL in general are always welcome.