Fun with the SQL Procedure — An Advanced Tutorial

Paul Kent, SAS Institute Inc., Cary, NC
<kent@unx.sas.com>

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.

This tutorial attempts to provide the reader with some insight into the SQL query optimiser and formulating queries that can be executed most efficiently. We hope some of the examples are sufficiently different from conventional solutions that you will gain increased awareness for the range of queries SQL can address.

OVERVIEW

The tutorial is presented in the following sections:

- helping joins go faster
- tolerating an external DBMS
- helping SET operators go faster
- helping SQL views go faster
- lateral thinking and SQL.

Many of the examples use rather synthetic tables as a basis for discussion. Tables are named `A, B, C,` and `D`. Columns are named `X, Y` and `Z` with `Xs (Xi)` used to signify that the table is sorted (indexed) on that column.

Some of the examples also give the DATA step code necessary to produce representative tables for the scenario being considered.

HELPING JOINS GO FASTER

There are two different profiles of join queries.

On one hand, you use a join to match up records from two or more files on a common key. Most records match, and the resulting table is as big as (or bigger than) the sum of the sizes of the base tables. These types of joins do not really benefit from indexes and other random access techniques — they perform best when left to operate in a sequential access mode. Your best chance at helping these joins go faster includes:

- large buffer sizes and many buffers
- accurate sort-order information
- other tools like the DATA step

On the other hand, joins are used to subset a large file, extracting records whose keys exist in some smaller file. These types of joins do benefit from random access techniques as avoiding processing the entire large file can produce significant savings.

Sorted data sets

Information describing the sort order is stored in data sets created with Release 6.07 TS301 of the SAS® System. PROC SQL will extract this information and use it to avoid further sorting. Consider two 100,000 row tables extracted from flat files stored by some traditional application at your site.

```
data A B;
  do X = 1 to 100000;
     Y='data';
     output;
     end;
  run;

data As(sortedby=Xs) Bs;
  do Xs = 1 to 100000;
     Y='data';
     output;
     end;
  run;

proc sort data=Bs;
  by Xs;
  run;

proc sql stimer;

  create table _null_ as
  select * from A,B
   where A.X = B.X;

  create table _null_ as
  select * from As,B
   where As.Xs = B.X;

  create table _null_ as
  select * from A,Bs
   where A.X = Bs.Xs;

  create table _null_ as
  select * from As,Bs
   where As.Xs = B.Xs;
```

The execution times for this (silly) example are tabulated below and show that telling SAS about the sort order of data sets that you read in from external files can pay back handsomely. There is no

need to specify the SORTEDBY= option for data sets that you plan to sort with the SORT procedure as that procedure sets the sort information automatically.

In fact PROC SQL honors a SORTEDBY= option set by PROC SORT without validating that the records are indeed sorted as it processes them. The (A, Bs) join saved time with respect to the (As, B) join, representing the cost of validating the assertion that the data are in fact sorted. This data was collected on an HP720 workstation running an un-released version of SAS software.

The most telling statistic is the one for the DATA step, which is well suited for sequential processing of large files. Your mileage may vary (these were very simple input data sets and there was no output data set), but you should always try the DATA step as an alternative when processing large sequential files.

Join Case	Real CPU	User CPU
A,B	29.9	25.3
As,B	22.9	20.5
A,Bs	22.2	19.9
As,Bs	15.6	15.2
DATA step	7.3	6.8

Tell the whole truth in your WHERE clause

The bulk of the SQL query optimiser is concerned with joins where the columns must compare equal. Equijoins (as this flavor of join is known) can be processed with a variety of strategies that perform better than the conceptual Cartesian Product that all joins are defined in terms of. These other strategies are detailed in "Inside the SQL Procedure's Query Optimiser" [Kent,1991].

PROC SQL uses all the equijoin predicates to evaluate alternative orders for processing a query. Consider these two queries:

```
select * from A, B, C
 where A.X = B.X
   and B.X = C.X

select * from A, B, C
 where A.X = B.X
   and B.X = C.X
   and C.X = A.X
```

In the former query PROC SQL considers two alternatives:

- join A and B on A.X = B.X into a temporary result, and join that temporary result with C
- join B and C on B.X = C.X into a temporary result, and join that temporary result with A.

The latter query allows PROC SQL to evaluate a third strategy:

- join A and C on A.X = C.X into a temporary result, and join that temporary result with B.

If A.X and B.X are indexed and A is significantly larger than B with C being the smallest table, then the last strategy is the optimal one — it minimises the size of the intermediate results.

Specifying the join predicate completely in this fashion becomes harder as you add more tables to the mix or the keys are hierarchical in nature. Consider the following data sets that model a geographical hierarchy:

Table(Alias)	Key(s)
STATE(A)	STATE
COUNTY(B)	STATE COUNTY
CITY(C)	STATE COUNTY CITY
DISTRICT(D)	STATE COUNTY CITY DISTRICT

The minimal WHERE clause needed to effect this join is much shorter than the fully specified one. The extra predicates added to allow SQL a complete chance at considering all alternatives are shown with the comment /* redundant */.

```
where A.STATE = B.STATE
  and A.STATE = C.STATE    /* redundant */
  and A.STATE = D.STATE    /* redundant */

  and B.STATE  = C.STATE
  and B.COUNTY = C.COUNTY
  and B.STATE  = D.STATE   /* redundant */
  and B.COUNTY = D.COUNTY  /* redundant */

  and C.STATE  = D.STATE
  and C.COUNTY = D.COUNTY
  and C.CITY   = D.CITY
```

It can be tedious work typing all that out, but the SAS macro language is very useful for reducing tedium. This macro produces the fully specified join clause for N tables with a hierarchical key like this example has. Ideally PROC SQL would support the NATURAL JOIN syntax of the proposed ANSI SQL2 standard (which implies this predicate) but we will not be able to deliver that support until a later release of SAS software.

```
%macro hjoin(tables=, keys=);

  %local i j k n needand;

  %let i=1;

  /*-----------------------------------------*/
  /*- load the values into a convenient    -*/
  /*- array format                         -*/
  /*-----------------------------------------*/
  %do %while( %scan(&tables, &i) ^= %str() );

      %local tab&i key&i;
      %let   tab&i = %scan(&tables, &i);
      %let   key&i = %scan(&keys, &i);

      %let   i = %eval(&i+1);
      %end;
```

```
%let n = %eval(&i-1);
%let no_and = 1;

/*-------------------------------------*/
/*- do over all but the last table    -*/
/*-------------------------------------*/
%do i=1 %to %eval(&n-1);

    /*-----------------------------------*/
    /*- for all the keys that apply to  -*/
    /*- this table                      -*/
    /*-----------------------------------*/
    %do j = 1 %to &i;

        /*--------------------------------*/
        /*- for the remaining tables     -*/
        /*--------------------------------*/
        %do k = %eval(&i+1) %to &n;


            /*----------------------------*/
            /*- add AND if needed        -*/
            /*----------------------------*/
            %if (&no_and) %then
                %let no_and=0;
            %else and ;

&&&tab&i...&&&key&j = &&&tab&k...&&&key&j

            %end;
        %end;
    %end;

%mend;
```

In an internal application tracking all the products that SAS Institute licenses to its customers, fully specifying the join clause for a hierarchy of four tables showed performance improvements of an order of magnitude when the query restricted one of the tables in the middle of the hierarchy by some other variable in that table.

Suppose the CITY table has a flag indicating whether the city has a smoking ban in public areas and you include that qualification in your where clause.

Instead of forming the join in the natural order — A to B and then to C and then to D, it becomes advantageous to start with the C table and restrict that to those cities that have a smoking ban. A complete specification of the join predicate gives PROC SQL the most latitude in considering whether to join the restricted C with A, B, or D as the next phase of solving the query.

Index the entire join path

When using SAS data sets to model a hierarchy, it is more effective to make composite keys for the lower levels of the hierarchy than multiple simple indexes. When nsing the geographical data sets described previously you should create a composite index on STATE, COUNTY for the COUNTY table, and another on STATE, COUNTY, CITY for the CITY table.

Individual indexes on STATE are not as useful to the join process since they are not as selective as indexes for STATE, CITY. However, some part of the key may bc almost as selective as the complete key itself. If CITY were mostly unique across all STATE it may be as cost effective to index just that part of the key. The index will be smaller and it will not qualify too many unnecessary records for any given query.

Abhor OR!

An OR operator in the join clause invalidates almost all the strategies that PROC SQL considers to optimise a join. Simple OR predicates like X=1 or X=2 are optimised into X IN (1,2), but predicates that refer to more than one variable will result in the most expensive join strategy of all — the Cartesian Product. When forced into this strategy PROC SQL must compare each row from one table with every row from the other table and evaluate the WHERE clause to decide whether or not to include this combination of rows in the output.

TOLERATING THE 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.
- Vote for the SAS/ACCESS folks to surface the index information on a DBMS table through their Access descriptors next time you see a SASware Ballot !

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 alteruative 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.SAS607.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(_DSENMR)
     then do;
     output wanted;
     _error_ = 0;
     end;

  /*-------------------------------*/
```

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

Use a custom SCL program

Screen Control Language(SCL) is the programming language of SAS/AF® Software and it includes a WHERE() function that facilitates constructing where clauses dynamically. The source code of this program is not a pretty affair, but it is available on request.

Consider the alternatives

SAS Institute staff in Australia experimented with each of these strategies. Each technique has advantages and disadvantages. This table summarises the results of an experiment in which we created a 100,000 row DB2 table and created a unique index on a randomly ordered field. We then tried to retrieve 10, 100, 1000 and 10000 rows from the table based on the values in a SAS data set. This is a somewhat simple and artificial test, but the results may inspire you to try these strategies on your data sets.

Rows Extracted	10	100	1,000	10,000
SQL	22.7	23.0	23.2	25.9
MACRO	.3	18.9	189.7	memory exceeded
DBLOAD	19.6	20.5	21.3	28.9
SCL	.2	.5	3.2	27.9
MODIFY	.2	1.3	11.6	110.9

The different strategies may have different paybacks with your data. You are encouraged to perform a similar experiment before committing to any one course of action here. In this example the vanilla SQL and DBLOAD cases are the safest. They do not perform poorly for any scenario. If you are doing lots of 10 row extracts from 100,000 row tables, then some other strategy may suit you better.

HELPING SET OPERATORS GO FASTER

People often use the SQL UNION operator when simple concatenation is all they really want. The set-oriented operators SQL provides model the more formal semantics of sets that include eliminating duplicate rows from the result — an expensive undertaking usually involving a sort.

You should use the UNION ALL operator when you do not need to eliminate duplicate rows (or you know in advance that there are not going to be any). The DATA step is still a better way to go if all you want to do is concatenate two data sets. PROC SQL needs some work in this area and we are pursuing performance improvements for a subsequent release of the SAS System.

HELPING SQL VIEWS GO FASTER

SQL views are convenient devices for encapsulating the joins between related tables. They allow end users to think of their data as a single logical table. If you use SQL views, you should be aware of these limitations and consider extracting the view into a temporary WORK data set before making repeated accesses to it.

- All columns specified by a view are extracted. If a procedure that references an SQL view uses DROP or KEEP options, that information does not reduce the amount of processing done by the view. We still materialise all the columns and then return a subset of those columns to the procedure.
- The view is a barrier as far as WHERE processing is concerned. WHERE statements and options applied to a view are not passed through to the underlying tables. The view materialises all the rows it would normally, and the extra WHERE filtering is done after that. This also means that WHERE statements applied to views are never passed to the underlying DBMS.
- The view must spool a copy of all records returned to many procedures as they make more than one pass over the data.

There is an exception to the first two points above. If a PROC SQL statement refers to a SQL view, then only the requested columns are extracted from the view. If a PROC SQL statement with a WHERE clause refers to a SQL view, then the WHERE clause is merged into the view. This merging happens only for PROC SQL, and not for other procedures that support WHERE clause processing. Consider the following code:

```
proc sql;

   create view a as
   select w,x,y,z from table
    where w between 10 and 20;

proc <anyproc> data=a;
   var x y;
   where w between 15 and 25;
   run;

proc sql;
```

```
create view b as
select x,y from a
 where w between 15 and 25;

proc <anyproc>
   run;
```

The first execution of PROC ANYPROC materialises a subset of three columns of data with values of W that range between 10 and 20. A subsequent stage of processing ignores column Z and performs a second restriction on W. The stand-alone WHERE statement and the columns requested by the PROC are not factored into the view A.

Creating a second view B that references the first provides PROC SQL with more information. The two view definitions are merged so we extract fewer columns and we also collapse the WHERE clause into W between 15 and 20.

```
                then b.x
                else b.y
                end;
```

This code was run on MVS. The trick to the faster solution is reformulating the join clause so that it has the form A.something = B.something. It relies on the fact that the determination of the variable to compare with A.X can be computed from variables in B only. The PROC SQL query optimiser recognises the latter query as an equijoin, computes the result of the case expression as a temporary and proceeds as if the join clause were as simple as A.X = B.temp. In the former query we are unable to select anything better than the Cartesian Product — the strategy of last resort!

Strategy	CPU	Elapsed	EXCP
Simple SQL	4.54	26.72	2
Smart SQL	.10	.17	2

THINK LATERALLY

This section of the tutorial presents small case studies showing SQL used in interesting (albeit not immediately obvious) applications. We hope they inspire a SQL solution to some sticky problem in your applications.

Not-quite-sure-of-the-key joining

A user wanted to merge two files with an interesting twist to the merge. He wanted to match A.X with B.X if B.X wasn't missing, otherwise he wanted to match A.X with B.Y. We consider the SQL join strategies for the obvious SQL formulation as well as hand tuned where clause that avoids the OR operator.

```
data a;
  do x = 1 to 1000; output; end; run;

data b;
  do i = 1 to 1000;
    if mod(i,3) then x = i;
                else x = .;
    if mod(i,5) then y = i;
                else y = .;
    output;
    end;
  drop i;
  run;

proc sql;

  create table _null_ as
  select * from a,b
   where (b.x is not null and a.x = b.x)
      or (b.x is null and a.x = b.y);

  create table _null_ as
  select * from a,b
   where a.x = case when b.x is not null
```

Recursive joining

The CPE group at SAS Institute wanted to merge data on processes run on our VAX mini-computer. On the VAX, each process may spawn children, who may spawn grandchildren and so on. We wanted to attach all such offspring to the parent process for book-keeping purposes.

The sample data has 3 parent processes: parent 1 has no children, 2 has many and 8 has only one child.

```
/*
 * the data shows three processes,
 * with subprocesses like this.
 *
 *        1
 *        ---
 *        2
 *          3
 *          4
 *            5
 *              6
 *          7
 *        ---
 *        8
 *          9
 *
 * you can recognise a "top level"
 * process as its parent id is 0
 * in the data below.
 */


data rj;
 input type $1. id pid;
 length stuff $5;
 stuff = type || put(id,1.) || put(pid,1.);
 cards;
1 0
2 0
3 2
```

```
4 2
5 4
6 5
7 4
8 0
9 8
run;

%macro rj(data=);

  %local level nlevel;
  %let   level = 1;

  proc sql;

  * get first level (parents) as table l1;
  create table work.l1 as
  select id as sid,
         id, pid,
         1 as level
    from &data
  where  pid = 0;


 /*
  * recursively get successive levels.
  * sql sets macro variable SQLOBS
  * to the number of rows it processed -- we
  * can use this to know when to stop...
  */
  %do %while(&SQLOBS > 0);

      %let nlevel = %eval(&level + 1);

      create table work.l&nlevel as
      select sid, p.id, p.pid,
             &nlevel as level
        from work.l&level s, &data p
       where s.id =  p.pid;

      %let level = &nlevel;
      %end;


 /*
  * now, concatenate all levels together
  * SID is the ultimate parent.
  */
  data work.leaf;
    set %do i = 1 %to %eval(&level - 1);
          work.l&i
          %end;
        ;
    run;

  proc sort;
    by sid id;

  %mend;


%rj(data=rj);

proc print; run;
```

Multi-criteria tie-breaker joining

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) are 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 recoding) 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(clinrace-normrace)) and similarly for SES.

Another user posted this reply:

The problem can be restated 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. Here is my solution. It incorporates my own interpretations and other specifics, but the general approach should work in any case.

```
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) +
             input(norm.id,10.10)
               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 the ID of the potential match as a fraction, to serve as the ultimate tie-breaker.

Exploiting the Cartesian product

The following example shows how the explosion effect of SQL

joins can be put to good use: SAS Consulting Services had a customer who wanted to: *do an "all combinations interleave" where data set ONE has vars A B C, data set TWO has vars B C, and we want to insert all rows from data set TWO after each by group (BY A) of data set ONE.*

We suggested: `SELECT UNIQUE A FROM ONE`, then join the list of unique A's with data set TWO to get the combination of each A with every B C. This resulting data set can be combined with the original data set ONE.

The customer's original program ran four DATA steps and two PROC SORT steps for each value of their key field (docket), so it probably could have been a little more efficient even solved as a DATA step problem. Another glitch is that there were 213 dockets in the smallest run, so they were running their six steps 213 times. SAS Consulting Services replaced all that code with four PROC SQL statements — approximately 30 lines of SQL code, including comments!

There were other minor changes to the rest of the job. The job went from 1 hour and 26 minutes of CPU time to 32.5 minutes. Direct I/O and elapsed time also went way down. The job ran on a VAX cluster, and the data files have *many* variables (the output data file had over 1000 vars).

Exploiting the Logical expression

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 seperate column of the report. The TABULATE procedure can do this, but the user also wanted to perform some computations with the counts obtained.

The data was collected for a number of questions `Q` and the responses were stored in the variable `V`. This SQL produces columns of `0,1` values and then counts the true values with the SUM function. Contrast this solution with one that uses the SUMMARY and TRANSPOSE procedures to get the individual counts as columns in the output.

```
data test;
input q$ v$;
cards;
a y
a y
a n
b y
b b
b n
b n
b n
;


proc sql;

select q label='Question',
       sum(v='y') as yea,
       sum(v='n') as nay,
       sum(v not in('y', 'n'))
          as other,
       calculated yea / calculated nay
          as margin
  from test
  group q;
```

CONCLUSIONS

The flexibilty of SQL is a two-edged sword: for many queries the underlying tables are small enough that any solution that works is the best solution. There are situations where an understanding of the SQL query optimiser can help you express a query so that PROC SQL has the best chance of finding an optimal method to access your data.

The best advice is to approach the problems with an open mind and be prepared to benchmark several different solutions. Many of the trade-offs are very data dependant and no matter what we suggest your mileage will vary!

REFERENCES

Stranieri, Mark [1991] "Differences and Interrelationships between the SQL Procedure and SAS/ACCESS Software" pp554-560 *SUGI 16 Proceedings*

Van Wyk, Jana [1991] "Efficient use of SAS/ACCESS Interfaces to INGRES and SYBASE" pp561-566 *SUGI 16 Proceedings*

Kent, Paul [1991] "Inside the SQL Procedure's Query Optimiser" pp567-571 *SUGI 16 Proceedings*

SAS Institute, Inc. [1989] "SAS Guide to the SQL Procedure: Usage and Reference"

SAS Institute, Inc. (1992) "SAS Technical Report P-122, Changes and Enhancements to Base SAS Software, Release 6.07"

ACKNOWLEDGEMENTS

Bill Gibson, David Ford and Michael Matthews of SAS Institute Inc (Australia) benchmarked the examples in the external DBMS section, as well as inventing the `MODIFY` and `SCL` solutions.

Ann Soeder of Glaxo Incorporated (USA) endured many experiments joining SAS data sets with large DB2 tables.

Ginny Dineley of SAS Institute Inc (USA) posed the recursive join challenge.

Howard Schreier of the U.S. Dept. of Commerce is a valiant defender of PROC SQL's honor on the SAS-L electronic users

group forum and an "SQL Lateral Thinker"

Merry Rabb of SAS Consulting Services posed the all combinations interleave challenge.

TRADEMARKS

DB2 is a registered trademark of IBM Corporation.

VAX is a registered trademark of Digital Equipment Corporation.

SAS, SAS/AF and SAS/ACCESS are registered trademarks of SAS Institute Inc., Cary, NC, USA.

AND, OR, and BUT have not been trademarked as far as the author is aware.

Your Turn

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

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