Paper 51

Performance Enhancements to PROC SQL in Version 7 of the SAS® System

Lewis Church, Jr., SAS Institute Inc.

<saslcj@wnt.sas.com>

ABSTRACT

This document describes three performance improvements made to PROC SQL for Version 7 of the SAS System.

These improvements are:

- development of an implicit, or automatic, form of PassThrough to enable better execution of PROC SQL queries that referenced one or more data sets via SAS/ACCESS® software,
- enabling any WHERE clause specified by a procedure statement to be pushed down into a view referenced in the DATA= option of that procedure, and
- making the most common use of an alias much less expensive.

IMPLICIT PASSTHROUGH

Implicit PassThrough (IP) is the result of a collaborative effort between the PROC SQL and SAS/ACCESS groups. The goal of this effort is to provide you with an automatic and implicit mechanism that provides many of the benefits of the PassThrough facility in PROC SQL when your PROC SQL query references one or more DBMS tables associated with a single libname engine. These benefits will occur without any need for you to use any special syntax or even be aware that DBMS tables are involved in the query — PROC SQL will simply take care of matters behind the scene, so to speak.

This new mechanism is intended to complement, rather than replace, the existing PassThrough mechanism. There is still very much a need for PassThrough, e.g., you may want to ask the DBMS to do a natural join. PROC SQL won't support natural join syntax until Version 9. In the meantime, you will need to use PassThrough in order to get the syntax past the PROC SQL parser.

Before delving more into the workings of IP, let us briefly consider how, in Version 6, you would do an inner/outer join of two tables from the same DBMS.

In Version 6, if a query involved tables from a DBMS such as DB2®, Informix®, Oracle®, or Sybase® there were basically three choices [Kent 1996] w.r.t. result materialization of an inner/outer join:

- create one or more SAS/ACCESS descriptors of the tables to assist in bringing the data over to the SAS side and do all the work there using PROC SQL, or
- use the PROC SQL PassThrough facility [Kent 1996] to pass the whole query or parts of the query to the DBMS, or
- create a view of the join in the DBMS, then create a SAS/ACCESS descriptor of that DBMS view, and have the DBMS do the work.

Each approach would achieve a similar result, with the second and third choices generally being the better approaches as:

- the DBMS materializes the inner- or outer-join using its native tables, and
- the DBMS typically possesses information (that PROC SQL is not privy to) which probably allows it to do a better job of query optimization and materialization.

In Version 7, PROC SQL has code that tries to offload as much of the work as possible for the DBMS to do. Basically, the IP code inspects a query to see if it looks profitable to attempt to send the whole query or just parts of it to the DBMS. It's essentially trying to automatically do what users in Version 6 had to explicitly use the PROC SQL PassThrough facility to do. The big differences between this approach and the Version 6 approach are that:

- you do not have to know about, or use, PassThrough to get some nice benefits, and
- PROC SQL may notice some offloading situations that the user may have missed.

A Detailed Look

PROC SQL provides the following IP components:

- the inspector/ferreter and
- the special query textualizer that generates ISO/ANSI SQL2 [ANSI 1992] Standard syntax (with engine-Specific modifications) for the SAS/ACCESS engines.

The SAS/ACCESS engines are responsible for components that textualize the following:

- table names,
- variable names,

- formatted SAS date values,
- formatted SAS time values, and
- formatted SAS date time values.

In addition to the above, the SAS/ACCESS engines are responsible for communicating to PROC SQL, at textualization-time, whether the DBMS underlying that engine uses nonstandard syntax for either (a) the SQL concatenation operator , (b) outer joins, or (c) both. If nonstandard syntax is being used, the SAS/ACCESS engine must also provide some indication of how to handle it.

The task of the inspector/ferreter is to examine the query to see if it, or parts of it, should be passed off to the DBMS. Any part(s) determined to be passable to the DBMS are extracted and sent to the DBMS for materialization.

The job of the textualizer is to generate a textual version of the query to hand off to the DBMS. This textual version must be syntactically acceptable to the DBMS and also must be formulated in such a way that it has the same semantics as the original query.

Ideally, what this would mean is that the textualizer would be primarily concerned with the generation of SQL2 syntax. Unfortunately, if the textualizer did this, many legal queries would fail, as quite a few of the DBMSes either (a) do not support features like full joins, intersection, difference, etc., (b) support them in a rather limited way, (c) support some of them but with nonstandard syntax, or (d) a combination of both (b) and (c).

To be fair, there are three conformance levels to the SQL2 Standard, each with a different set of features. Level One is basically ISO/ANSI SQL-89 [ANSI 89] with a few additional features and corrections, Level Two is Level One plus more features, and Level Three is Level Two plus some more features. The DBMS vendors are free to support any level they wish. Many of the DBMSes do not seem to conform to any particular level; instead, they vary considerably, as far as support (or nonsupport) of certain features like the set operators, outer joins, column aliases, correlated subqueries, etc. are concerned.

The PROC SQL Query Tree

At parsing time, the textual version of the query, as entered by the user, is transformed into an internal form known as the PROC SQL query tree. This is essentially a directed acyclic graph that may have different shapes and constituents during its lifetime [Navathe 1994, Ramakrishnan 1998, Yu 1998].

The main advantage of representing the query this way is that it is a more easily handled and malleable form of the query than its textual counterpart. These benefits are extremely important w.r.t. query optimization and ferreting out parts for potential offloading to a SAS/ACCESS DBMS.

An example immediately follows concerning how the query tree looks just before query execution. Output is generated via the use of both the `_tree` and `_method` [Kent 1995] options.

The tree indicates that data is coming from one source, a data set called `x` whose libref is `sql`. This data set has three variables: z, x, and y. No WHERE or HAVING clause is involved. Nor does this query involve features like column and table aliases, aggregates, GROUP BY, and ORDER BY.

```
 1? proc sql _method _tree;
 2?
 3? select *
 4?   from sql.x;


NOTE: SQL execution methods chosen are:

     sqxslct
         sqxsrc( SQL.X )

Tree as planned.
                              /-SYM-V-(x.z:1 flag=01)
                    /-OBJ----|
                   |         |--SYM-V-(x.x:2 flag=01)
                   |          \-SYM-V-(x.y:3 flag=01)
          /-SRC----|
         |          \-TABL[SQL].x opt=''
--SSEL---|
```

The Inspector/Ferreter

The basic task of this routine is to do a preorder traversal of the query tree. If, during this traversal, it finds that the subtree rooted at the particular node that it is visiting passes all the tests for offloading to the DBMS then that subtree becomes a candidate for offloading. The ferreter/inspector will then attempt to textualize the subtree and will pass it off to the SAS/ACCESS software for approval/disapproval.

If the SAS/ACCESS software sends back an OK, then PROC SQL eliminates this subtree and replaces it with a node containing all the information needed by the DBMS to handle the part of the query associated with that subtree. After doing that, the algorithm continues with the preorder traversal, as there may very well be other subtrees that can be passed off.

However, if the SAS/ACCESS software sends back a failure indication, then PROC SQL will try to textualize a simpler version of the tree.

Any portions of the tree that can not be handled by the DBMS are handled by PROC SQL -- just as in Version 6.

Consider the example below. Output generated by the `_method` option indicates that PROC SQL was able to offload the query to the DBMS, i.e., Oracle. That DBMS does the hard part of the work; PROC SQL is simply going to sit back and display the results.

`_method` information further indicates that PROC SQL was not able to get the DBMS to do the work for the second query. The reason for that is because the version of Oracle referenced by SAS/ACCESS libname engine does not handle full outer joins. Hence, PROC SQL had to do it the old-fashioned way, that is, it had to materialize the entire query itself.

```
1          libname user oracle user=dbitest password=XXXXX
2              preserve_tab_names=no
3              preserve_col_names=no;
NOTE: Libref USER was successfully assigned as follows:
      Engine:        ORACLE
      Physical Name:
4
5
6           proc sql _method;
7
8              /*------------------------------------------------*/
9              /*-- The ORACLE engine likes this INNER join.   --*/
10             /*------------------------------------------------*/
11             select i.invprice, p.prodlist
12               from USER.product p inner join USER.invoice i
13                      on p.prodname = i.prodname;

NOTE: SQL execution methods chosen are:

      sqxslct
          sqxextr( connection to ORACLE
                      /* dbms=ORACLE, connect options=() */
                      ( select i.INVPRICE, p.PRODLIST
                          from PRODUCT p, INVOICE i
                         where p.PRODNAME = i.PRODNAME ) )

14
15             /*------------------------------------------------*/
16             /*-- It doesn't like FULL joins though.         --*/
17             /*------------------------------------------------*/
18             select i.invprice, p.prodlist
19               from USER.product p full join USER.invoice i
20                      on p.prodname = i.prodname;

NOTE: SQL execution methods chosen are:

      sqxslct
          sqxjm
              sqxsort
                  sqxsrc( USER.PRODUCT(alias = P) )
              sqxsort
                  sqxsrc( USER.INVOICE(alias = I) )
21
22          quit;
```

What Types of Queries are Candidates for Passing to the DBMS?

Any query (or part) that:

- references exactly one SAS/ACCESS libname,
- is legal according to the SQL2 Standard, and
- is recognizable by PROC SQL according to the documentation in "SAS OnlineDoc™ Version 7.0" will be textualized and passed to the DBMS's parser UNLESS it is disqualified by PROC SQL, due to one or more of the situations enumerated below.

Disqualification Situations

Any query that references two or more distinct SAS/ACCESS libnames will be disqualified immediately. This is a SAS/ACCESS restriction. Why does the restriction exist? Different libnames potentially can refer to different connections which could cause a potential conflict regarding the tables in the query, i.e., different connection schemas with different grant authority on the tables.

Any single table query that contains no references to a SAS/ACCESS libname DBMS will be disqualified immediately from consideration for offloading to the DBMS.

Any single table query that contains neither:

- a SELECT DISTINCT/UNIQUE, nor
- an SQL aggregate function, nor
- a GROUP BY clause, nor
- a HAVING clause, nor
- an ORDER BY clause

will be disqualified immediately.

Finally, any query tree, or subtree of that tree, will be disqualified if it involves one or more of the following:

- CONNECTION TO,
- remerging,
- one or more DATA step functions,
- data set options,
- one or more outer unions,
- one or more ANSIMISS/NOMISS inner or outer joins,
- any SQL function other than: count(*), count(x), freq(x), n(x), avg(x), mean(x), max(x), min(x), sum(x), coalesce(), or nullif()
- one or more truncated comparisons, or
- the INTO clause.

The DBMSes will reject many legal SQL2 queries

As was mentioned earlier, the SQL2 Standard has three conformance levels. Many vendors do not have support for much beyond the lowest level (Level One). Typically, many do not fully support any one level but, instead, have features from all three levels.

Some perfectly legal queries are rejected by various SAS/ACCESS DBMSes, due to the presence of:

- aliases in some contexts,
- the EXCEPT, EXCEPT ALL, INTERSECT, and INTERSECT ALL operations,
- nested and correlated subqueries in certain contexts,
- the COALESCE operation, and
- the NULLIF operation.

This document does not discuss the rules for these rejections. The reason for this is that these rejections are engine- and conformance-level specific; PROC SQL is not privy to that knowledge. Please see the SAS/ACCESS documentation for more information about this.

Enabling and Disabling Implicit PassThrough

The default status for IP is that it is automatically enabled at PROC SQL invocation-time.

It can be explicitly enabled or disabled for a query, or series of queries, by the PROC SQL options known as `ipassthru` and `noipassthru`, respectively.

There are two main reasons that you may want to disable IP:

- The DBMSes use SQL2 semantics for NULL values, these behave somewhat differently than SAS missing value semantics.
- PROC SQL may do a better job of query optimization for certain queries. This enables (pun intended) you to very easily compare query execution resource usage with and without the usage of IP.

```
proc sql noipassthru;
   /*-- Implicit Passthrough is disabled --*/
   ...
   /*-- enable IP --*/
   reset ipassthru;
   ...
   /*-- disable IP --*/
   reset noipassthru;
   ...
   /*-- enable it again --*/
   reset ipassthru;
   ...
quit;
```

PUSHING A WHERE CLAUSE DOWN INTO A VIEW

A WHERE clause that is part of a PROC SQL query has always been pushed down into, i.e., merged with, any relevant view that it is associated with. There are some very nice benefits to doing this [Navathe 1994, Ramakrishnan 1998, Yu 1998]. In *most* situations, it is the correct course of action [Pellenkoft 1997, Stonebraker 1998] .

The following example is from Version 6.12 of the SAS System.

The view is basically a join of two tables with the restriction that the x value in one table must equal the y value in the other table. Notice that the predicate `x > 3` is applied to the view result. Taking a look at the query tree, one can readily notice that this predicate has effectively been pushed down into the view definition. Furthermore, the optimizer has noticed that if `x > 3` and `x1.x = x2.y` both hold then it can deduce that `x1.x > 3` and that `x2.y > 3` must also hold. That potentially speeds up the join quite a bit, as more of the rows that do not contribute to the final result may be eliminated before the join even encounters them.

```
           proc sql _tree;

              create view WORK.vx as
                 select x1.x, x2.y
                   from sql.x x1, sql.x x2
                  where x1.x = x2.y;
NOTE: SQL view WORK.VX has been defined.


              select *
                from WORK.vx
               where x > 3;

Tree as planned.
                               /-SYM-V-(X1.X:2 flag=01)
                     /-OBJ----|
                    |          \-SYM-V-(X2.Y:3 flag=01)
           /-JOIN---|
          |         |                              /-SYM-V-(X2.Y:3 flag=01)
          |         |                    /-OBJ----|
          |         |          /-SRC----|
          |         |         |         |--TABL[SQL].X opt=''
          |         |         |         |          /-NAME--(Y:3)
          |         |         |          \-CGT----|
          |         |         |                    \-LITN(3)
          |         |--FROM---|
          |         |         |                    /-SYM-V-(X1.X:2 flag=01)
          |         |         |          /-OBJ----|
          |         |          \-SRC----|
          |         |                   |--TABL[SQL].X opt=''
          |         |                   |          /-NAME--(X:2)
          |         |                    \-CGT----|
          |         |                              \-LITN(3)
          |         |--empty-
          |         |          /-SYM-V-(X2.Y:3)
          |          \-CEQ----|
          |                    \-SYM-V-(X1.X:2)
 --SSEL---|
```

However, in Version 6, a WHERE clause that is part of a procedure statement whose DATA= option references a view is not pushed down into the view. This is mainly due to some thorny technical issues.

As an example, consider doing a task similar to that above. Instead of using PROC SQL, use PROC PRINT .

```
proc print data=WORK.vx;
   where x > 3;
run;
```

The way that this works is for the join, as defined in the view, to execute and just before each row is ready to be passed to PROC PRINT a check is made to determine if the value of its x component is greater than 3. If so, then it is passed on; otherwise, an attempt is made to fetch the next row from the join. This continues until all the rows are retrieved from the join.

Below is what the join would look like that was presented to PROC PRINT. Please compare the join trees. You will notice that the PROC SQL one has taken advantage of some optimization opportunities, whereas the PROC PRINT one was not able to.

```
Tree as planned.
                               /-SYM-V-(X1.X:2 flag=01)
                     /-OBJ----|
                    |          \-SYM-V-(X2.Y:3 flag=01)
           /-JOIN---|
          |         |                              /-SYM-V-(X1.X:2 flag=01)
          |         |                    /-OBJ----|
          |         |          /-SRC----|
          |         |         |          \-TABL[SQL].X opt=''
          |         |--FROM---|
          |         |         |                    /-SYM-V-(X2.Y:3 flag=01)
          |         |         |          /-OBJ----|
          |         |          \-SRC----|
          |         |                    \-TABL[SQL].X opt=''
          |         |--empty-
          |         |          /-SYM-V-(X1.X:2)
          |          \-CEQ----|
          |                    \-SYM-V-(X2.Y:3)
 --SSEL---|
```

In Version 7, some of the “thorny technical issues” alluded to earlier have been resolved. As a consequence, the PROC PRINT example just given will now perform exactly like the PROC SQL one.

COLUMN ALIASES ARE NOW MUCH LESS EXPENSIVE TO USE

An alias, instead of the LABEL= feature, is often used to provide a column with a new heading, e.g.,

```
select y as year
  from sql.x;
```

Another common use of aliases is to provide a handle for a nontrivial column expression so that it can be referenced, if need be, within other expressions of the query, e.g.,

```
/*-- won't quite work as written --*/

select x+1 as xplus1
  from sql.x
 where xplus1 > 365;
```

In Version 6, due to a variety of technical reasons, whenever a SELECT statement had at least one aliased column, the length of the query pipeline [Navathe 1994, Ramakrishnan 1998] for *that* SELECT *could* increase by at most one step in Version 6 of the SAS System, e.g.,

```
reset _method;

/*-- each of these queries will generate --*/
/*-- the SAME form of method tree in      --*/
/*-- Version 6.                           --*/

 select y as year
   from sql.x;

 ...

 select x+1 as xplus1
  from sql.x;

 ...

 select y as year, x+1 as xplus1
  from sql.x;

 ...

 select y, x+1 as xplus1
  from sql.x;

 ...

 NOTE: SQL execution methods chosen are:

       sqxslct
           sqxfil
               sqxsrc
```

Also, in Version 6, there was no legal way to reference an alias that was just simply a renaming of a variable. It was possible, however, to reference any other category of column alias provided that it was accomplished via the CALCULATED keyword, e.g.,

```
/*-- fails in Version 6 --*/

select y as year
  from sql.x
  where year > 1937;


/*-- this won't work either --*/
/*-- as it's also illegal    --*/

select y as year
  from sql.x
 where CALCULATED year > 1937;


/*-- this WILL work --*/

select y+0 as year
  from sql.x
 where CALCULATED year > 1937;
```

In Version 7, two improvements were made to the handling of simple renames. These renames, unlike those in Version 6, (a) do not cause the pipeline to possibly lengthen by one step and (b) the alias can be referenced in all contexts that the variable can be referenced in. Furthermore, the CALCULATED keyword does not have to immediately precede the alias for a variable rename; in fact, this keyword must not precede it for the same reason that it would not precede the renamed variable.

```
reset _method;

/*-- This works in Version 7. Notice      --*/
/*-- the shorter pipeline.                --*/

 select y as year
   from sql.x
  where year > 1937;

 ...


 NOTE: SQL execution methods chosen are:

       sqxslct
           sqxsrc( SQL.X )
```

FORMAT=, INFORMAT=, and LABEL= can also be applied

```
select y as year
           FORMAT=... INFORMAT=... LABEL=...
  from sql.x
 where year > 1937;
```

At present, due to some technical issues, LENGTH= can be applied to this simple rename *but* the rename will behave exactly as it did in Version 6. There are plans to address this in Version 9.

CONCLUSION

The three improvements discussed in this paper are already providing benefits to the Base SAS user community. Like many other enhancements, they can still be made better and thus provide even more benefits; these refinements will occur as PROC SQL evolves.

To summarize, the performance improvements are:

- many queries, or parts of queries, involving SAS/ACCESS tables are now passed directly to the underlying DBMS for result materialization,
- WHERE clauses associated with SAS procedure statements that reference a view via the DATA= option are now pushed down into the view, and
- it is now much less expensive to use an alias that is a simple variable rename.

We look forward to hearing from you at future conferenc- es.

Please visit the Base SAS Software Research and Development home page at `http://www.sas.com/rnd/base/`.

REFERENCES

American National Standards Institute (1989), *Database Language SQL With Integrity Enhancement, ANSI X3,135-1989*, American National Standards Institute, New York. Also available as ISO/IEC Document 9075:1989.

American National Standards Institute (1992), *Database Language SQL, ANSI X3,135-1992*, American National Standards Institute, New York. Also available as ISO/IEC Document 9075:1992.

Kent, P. (1995), " SQL Joins - The Long and The Short of It," *Proceedings of the Twentieth Annual SAS Users Group International Conference*, 20, 206-215.

Kent, P. (1996), " An SQL Tutorial - Some Random Tips," *Proceedings of the Twenty-First Annual SAS Users Group International Conference*, 21, 237-241.

Navathe, S. And Elmasri, R. (1994), Fundamentals of Database Systems, 2nd edition, New York: The Benjamin/Cummings Publishing Company, Inc.

Pellenkoft, A. (1997), Probabalistic and Transformation based Query Optimization, Ph.D. Thesis, University of Amsterdam, The Netherlands. Technical Report UBM: N 97-267.

Ramakrishnan, R. (1998), Database Management Systems, New York: WCB/McGraw-Hill.

Stonebraker, M. (1998), Object-relational DBMSs: Tracking the Next Great Wave, 2nd edition, San Francisco: Morgan Kaufmann Publishers, Inc.

Yu, C.T. and Meng, W. (1998), Principles of Database Query Processing for Advanced Applications, San Francisco: Morgan Kaufmann Publishers, Inc.

ACKNOWLEDGEMENTS

Fred Levine, now a member of the Query Research and Development group at SAS Institute, was my partner in implementing Implicit PassThrough. Without his programming skills and knowledge, Implict PassThrough would not be what it is today.