Inside the SQL Procedure's Query Optimiser

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

Abstract
This paper describes the methods and strategies that the SQL procedure considers before evaluating queries. Also discussed are enhancements made to the query optimiser since Release 6.06 of the SAS System.

Background
SQL (Structured Query Language) is a language for accessing data stored in tables. It is a nonprocedural language that is implemented by many database systems on mainframes, minicomputers, and personal workstations. The SQL language enjoys far greater applications portability than proprietary query languages due to its high degree of standardisation.

SQL is a high-level set oriented declarative language — queries express what data one is interested in and not how to access the database. The task of a query optimiser is to consider the alternatives and choose a strategy that will answer the question correctly and efficiently.

This paper discusses the PROC SQL optimiser under five general headings.
1. Algebraic Manipulation of Queries
2. Data Flow Analysis
3. Join-Query Strategies
4. Subquery Strategies
5. Set-Operator Strategies

The paper describes the algorithms implemented by PROC SQL in Release 6.06 of the SAS System, unless noted otherwise as an enhancement — these enhancements will be delivered with a subsequent version of the SAS System.

Algebraic Manipulation of Queries

Conjunction
A complex sentence in logic true if and only if each of its components is true

Conjuror
One that practices magic arts

— Webster, 9th Ed

The indexing features provided by the SAS System favor conjunctive WHERE clauses — expressions connected with the AND logical operator. By contrast, some systems (like the System 2000 DBMS) favor disjunctive expressions as they have the ability to evaluate the different operands of an OR expression and form the union of the matching records from those operands.

PROC SQL transforms expressions into conjunctive normal form to simplify the expressions and take advantage of indexes that may be defined. PROC SQL will transform expressions containing a NOT operator using De Morgan's rules of logic. The expressions

\[
\begin{align*}
\text{NOT}(A=B) \\
\text{NOT}(A>B)
\end{align*}
\]

are transformed into

\[
\begin{align*}
A \text{ NOT=} B \\
A \text{ <= } B
\end{align*}
\]

respectively. While these changes seem to offer little benefit, they do simplify the number of tests that must be made to evaluate the expression at runtime and they allow further transformations to occur later.

\[
\begin{align*}
\text{NOT}(A=1 \text{ OR } B=2)
\end{align*}
\]

is transformed into

\[
\begin{align*}
\text{NOT}(A=1) \text{ AND NOT}(B=2)
\end{align*}
\]

which is further refined into

\[
\begin{align*}
A \text{ NOT=} 1 \text{ AND } B \text{ NOT=} 2
\end{align*}
\]

The latter is in conjunctive normal form, and PROC SQL can capitalise on this later in the query optimisation process.

Once De Morgan's transformations have been applied, AND and OR operators are considered with the intention of combining operands. The expressions

\[
\begin{align*}
A \geq 7 \text{ AND } A < 10 \\
A = 1 \text{ OR } A = 2
\end{align*}
\]

are transformed into

\[
\begin{align*}
A \text{ BETWEEN } 7 \text{ AND } 10 \\
A \text{ IN } (1,2)
\end{align*}
\]

respectively. All of these transformations reduce the complexity of the expression that must be evaluated for each row processed by PROC SQL.

In some cases, the expression can be eliminated completely. Both expressions

\[
\begin{align*}
A = 1 \text{ AND } A = 2 \\
A \text{ NOT=} 1 \text{ OR } A \text{ NOT=} 2
\end{align*}
\]

are universally false and true respectively, and this identity is recognised. It is unlikely that a user would code an expression like this directly, but if he used \(A=1\) in his WHERE clause and \(A=2\) were part of the WHERE clause stored in a view, then PROC SQL can recognise that the composite WHERE clause is false.

These optimisations are made for WHERE statements used with conventional SAS procedures as well as for WHERE clauses in SQL queries.
Data Flow Analysis

The efficiency of many applications written using SAS software can be improved with judicious usage of the DROP and KEEP data set options. PROC SQL inspects the query and prunes all variables that are not needed by a higher level. Consider:

```
SELECT A
FROM TABLE
WHERE B > 12
ORDER BY C;
```

All three variables are needed from TABLE, but B can be discarded as soon as the WHERE clause has been evaluated, and C is not required after the sorting process.

Trimming unnecessary variables during each stage of processing a query allows PROC SQL to reduce the size of intermediate results and process queries more efficiently.

Join-Query Strategies

Conceptually, a join in SQL is performed by considering each row in one table with every row in the other table, and evaluating the WHERE clause for each crossing. This is a relatively expensive process, so PROC SQL inspects the WHERE clause to see if it can use a different (cheaper) algorithm to get the same end result.

The algebraic transformations mentioned above put the WHERE clause in conjunctive normal form. This is useful as PROC SQL can consider each operand of the AND operator in isolation without altering the results of the join query.

The first optimisation is to pre-evaluate any components of the WHERE clause that relate to a single table only. Even if no other optimisations can be made, this can reduce the number of crossings that must be evaluated in evaluating the Cartesian product (or join).

Consider two tables A and B with 100 rows each, joined by the expression

```
A.X = 1 and A.Y > B.Y
```

A naive algorithm would make 10000 (100 * 100) evaluations of the expression (for each combination of rows in the two tables).

A better algorithm would evaluate

```
A.X = 1
```

over the rows of A (100 evaluations) and then use the subset of rows that matched half the records in A, this scheme would be twice as efficient as the completely naive approach.

Once expressions that relate to a single table have been "pushed down" to be pre-evaluated before processing the join, PROC SQL must decide between alternative join algorithms. In Release 6.06, PROC SQL chooses between these join methods.

- Index Lookup join
- Sort and Merge join
- Step Loop (Brute Force) join

Subsequently, we have also developed a hash-based join, that does not require indexes or sorting. Each algorithm is described here, followed by the method PROC SQL uses to decide which one is appropriate for a given query.

Each of these algorithms will be discussed in turn, using this hypothetical query as a framework.

```
SELECT *
FROM LEFT, RIGHT
WHERE LEFT.X = RIGHT.X;
```

Note that the query is an Equijoin. That is to say, at least one of the conjuncts in the WHERE clause compares a variable from the left table to a variable from the right table for equality. The only join method PROC SQL can employ on joins that are not equijoins is the step loop method.

Joins with a WHERE clause that relates expressions (other than simple variables) are transformed where possible. A WHERE clause like

```
WHERE LEFT.X + 20 = RIGHT.X
```

can be transformed into

```
WHERE <temp1> = RIGHT.X
```

if we replace the reference to the LEFT table with an inline view that selects the needed columns as well as

```
LEFT.X + 20 as <temp1>
```

PROC SQL is capable of these transformations in Release 6.06.

Some expressions are unbalanced in that references to variables in both tables appear on the same side of the comparison operator.

The two expressions

```
WHERE LEFT.X = RIGHT.X;
WHERE LEFT.X - RIGHT.X = 0;
```

are logically identical (subtracting RIGHT.X from both sides of the latter will not change the outcome). In Release 6.06, PROC SQL will recognise the first form as an equijoin but not the second. A subsequent enhancement allows us to recognise the second form as well.

Index Lookup Join

PROC SQL can perform an indexed join if either table has an index on X. We would process the query as follows if the LEFT has an index on X.

1. Fetch next record from RIGHT.
2. Position index of LEFT at value of X from RIGHT.
3. Fetch next record from LEFT via index.
   - next record exists? — output and goto 3.
   - no more records? — goto 1.

This approach works very well when "pulling" a small subset from a large file that is indexed. An index in a master data set will be useful for joining a transaction data set with a few product-ids with that master data set — we will not have to access each page of the master data set.

The index approach is not always optimal. If the transaction data set references product-ids in a cyclical fashion (say 1, 100, 300, 500 then back to 1 and so on) we may find that the same page in the master data set must be read more than once. It is possible to sort the references into page-order for the join, but PROC SQL does not do this currently.

568
The algorithm can be used for outer joins when the outer join involves the non indexed table — a right join in our example. If no matching records were found for a given value of X, then we fabricate the missing values needed for the outer join and output.

**Sort and Merge Join**

The Sort and Merge join is similar to traditional SAS DATA step merges, except when dealing with BY groups that have two or more records in both of the contributing data sets — SQL must form the cross product of the rows.

It is efficient to sort the data on the join variables — the cross products of the BY groups individually are much smaller than the cross product of both tables as a whole.

The sorts that are required for processing a merge join are inserted into the pipeline of operations. Recall that expressions that relate to a single table are always evaluated before attempting a join. The same holds for sorting — we will screen out unwanted records as early as possible. In Release 6.06 these sorts are always done by the SAS SORT method, but an enhancement allows selection of the host sorting method also. Many sorts can be done in-memory and incur no extra I/O overhead.

As an example:

```sql
SELECT A.X, B.Y
FROM A, B
WHERE A.X = B.X AND A.Y > 20;
```

will be processed as

```sql
MERGE JOIN ON X OF
  SORT ON X
  ROWS FROM A
  WHERE A.X = B.X AND A.Y > 20
  SORT ON X
  ALL ROWS FROM B
```

**Hash Join**

The hash join algorithm avoids the need to sort the two data sets being joined. It performs especially well when either of the tables can fit in main memory. This method has been implemented since Release 6.06 was shipped.

PROC SQL estimates the sizes of the two contributors to the join, and loads the records from the smaller one into an indexed cache. The values of the join variables are used to compute the position in the cache.

The other contributor is then processed a record at a time. The value of the join variables are used to compute the position in the cache, the matching records are located and combined records are output.

Hash joins show impressive performance gains over merge joins — we have measured up to 40% improvement in this area. In Release 6.06 hash joins (had we implemented them) would have been preferable to sort merge joins in most cases. However, now that SAS data sets retain information on their inherent sort order, merge-join is better if the tables are already in the correct order (no sorting required) and hash-join prevails in the other cases.

**Step Loop Join**

This method forms the Cartesian product by “brute force” and evaluates the WHERE clause for each combination of rows from the respective tables. It is far more expensive than any of the specialised join methods described above, but it is not limited to equijoin processing.

Fortunately, very few queries require this drastic approach. An enhancement to PROC SQL warns the user that the query requires a Cartesian product should they specify a join that is not an equijoin.

**Selecting an Appropriate Join Method**

PROC SQL must choose between the various methods for processing a join for each query. If the WHERE clause does not contain an equijoin predicate then we have no choice but to use the expensive Step Loop method. If there is an equijoin, then we choose among the alternatives according to the following rules:

- Is there a useful index? If so, then use an Index Lookup join.
- Otherwise resort to Merge join.

When we implemented the hash-join method, the choices became:

- Is there a useful index? If so, then use an Index Lookup join.
- Otherwise resort to Merge join.
- Otherwise resort to Hash join.

**N-way Joins**

Until now this paper has kept things simple by considering two-way joins only. When more than two tables are joined (PROC SQL can join up to sixteen in a single query), we partition the join into a series of pairwise joins. Consider the alternatives for a three-way join with tables A, B, and C on a common variable X (no indexes).

- Join A and B into a temporary result, joining that with C.
- Join A with the temporary result of joining B and C.

PROC SQL estimates the sizes of the intermediate results of all the alternatives and chooses the permutation with the smallest. An enhancement developed since Release 6.06 also tracks the inherent order of the temporary tables. In Release 6.06 we would sort A and B (on X) and merge-join into a temporary, then sort the temporary and C (on X) and merge-join for a final result.

Now the software tracks the fact that the output of joining A and B is already in X order, so we avoid the extra sorting process. We will also take advantage of the “sort-order” information that is now preserved by SAS data sets to avoid sorting when possible.

**Subquery Strategies**

**Uncorrelated Subqueries**

PROC SQL evaluates the result of the subquery once, and caches the result to be used each time the subquery is referenced during the outer query.
Correlated Subqueries

PROC SQL evaluates the result of the subquery for each new instance of the correlation values and stores the result (together with the correlation values that produced it) in an indexed cache. If a subsequent row of the outer query has the same correlation values, then it is not necessary to re-evaluate the subquery — we can just use the stored result.

There are logical transformations that convert correlated subqueries into equivalent joins. We are currently researching the implementation of these transformations for some later release of PROC SQL.

IN Subqueries

Sometimes IN subqueries can make use of an index to satisfy the request. If the column \( X \) in data set TABLE has an index, the WHERE clause evaluation process will take the current value of \( X \) and look it up in that index to determine if the WHERE clause is true or false without actually retrieving the record from the data set.

If an index cannot be used, the values returned by the IN subquery are cached in an indexed format, so that a lookup can be done quickly for various values of \( X \).

Set Operators

Set operations on more than two tables are broken down into pairwise operations in the same fashion that joins are.

The versions of the SQL set operators that do not involve the ALL keyword (UNION, INTERSECT, and EXCEPT) involve sorting the two tables as well as eliminating duplicate rows from the results. These operations are more costly to process than the traditional "SET A B" concatenation done by the data step.

The ALL versions of the set operators do not involve as much processing as the ones with the true SET semantics required by ANSI SQL. However, a view defined with UNION ALL of two tables is still more expensive to process than the corresponding data step code. The data step is especially well optimised for this style of processing. We are currently researching methods of recognising set operations that can be processed in a more streamlined fashion — however users now have the option of using data step views should they do lots of this style of processing.

Other Improvements

These items do not really pertain to the PROC SQL query optimiser, but do present information on the performance of PROC SQL.

EXPLAIN option

This option will print a summary of the methods used to evaluate the query to the SAS log. It will help SQL users understand the strategy used to evaluate their queries. The option is still under development at this time.

CALCULATED reference

This is an enhancement to the language that lets one avoid repeating expressions that are calculated in the SELECT clause. Traditional SQL would have it that the result of the \( A \times B \) in this example is only visible as \( X \) in an outer query. PROC SQL will introduce a new qualifier for variable names that indicates a reference to the expression calculated on the select clause of the immediate query.

In the first example the result of an expression calculated in the SELECT clause is used as a test in the WHERE clause. In the second example, we calculate LIST and COST as the first two columns, and then use those calculated values to compute PROFIT.

Both of these examples can be coded in standard SQL by repeating the expression — in many cases the SQL interpreter will have to repeat the calculation too. These examples can also be coded as in-line views, but the query is a lot more complicated.

NATIVE SQL with SAS/ACCESS® Software

This feature (called SQL passthru by Internal Research and Development groups) will allow one to embed arbitrary SQL into PROC SQL queries. This SQL will be passed directly to the database — it is expected to return a table that will be treated by PROC SQL much like an in-line view. An example of this feature used with SAS/ACCESS Software for DB2® is:

```sql
PROC SQL;
SELECT a, b FORMAT=emdkyy.
FROM connection to db2
    SELECT alongcolumn,
         anothercol
    FROM aditable
    WHERE ...
    AS xxx(a,b)
WHERE month(b) IN (1,2,3)
;
```

The SQL in parentheses after the `connection to` phrase is passed directly to SAS/ACCESS software. It is used to prepare a cursor using the dynamic SQL features of the DBMS.

As far as PROC SQL is concerned the `connection to` phrase defines a virtual table that we have named `xxx` for this query and it has the two columns `a` and `b` — this syntax can be used anywhere an in-line view might appear. If the DBMS does not support some feature that PROC SQL does, you can still get the desired result — the syntax above shows `FORMAT=` and the `MONTH` function being
This feature allows you to be more specific about the SQL that is passed into the DBMS. Assume that DBTAB1 and DBTAB2 are database tables and that VTAB1 and VTAB2 are SAS/ACCESS descriptors for these tables.

The first query in this example accesses data via the SAS/ACCESS descriptor — that data is materialised one row at a time, so PROC SQL has no alternative but to do the grouping by Y and summing of X internally. The second query hands the SQL expression directly to the DBMS, which performs the grouping and summing. PROC SQL just processes the resulting rows returned by the DBMS.

```
SELECT Y, SUM(X) 
FROM VTAB1 
GROUP BY Y;
```

```
SELECT * 
FROM CONNECTION TO <dbms> 
( 
  SELECT Y, SUM(X) 
  FROM DBTAB1 
  GROUP BY Y 
); 
```

Another case where this pass thru capability is useful is when one is joining tables that reside in the DBMS.

In the first query PROC SQL selects one of its internal join methods — Sort Merge for Release 6.06 as the SAS/ACCESS interface does not surface index information on the underlying tables. Processing this query requires sorting both tables returned from the DBMS into KEY order by PROC SQL.

In the second query, the join can be processed by the DBMS itself — and it may take advantage of any specialised method available in the DBMS. PROC SQL will just have to process the resulting rows.

```
SELECT * 
FROM VTAB1, VTAB2 
WHERE VTAB1.KEY = VTAB2.KEY;
```

```
SELECT * 
FROM CONNECTION TO <dbms> 
( 
  SELECT * 
  FROM VTAB1, VTAB2 
  WHERE VTAB1.KEY = VTAB2.KEY 
); 
```