Abstract

This talk discusses joining tables with PROC SQL. When joining small tables one can just "go with the defaults" and not worry about performance; these defaults do not always scale well to large tables. The issue is clouded even further when one or more of the tables reside in an external DBMS.

Understanding the choices made by the PROC SQL query optimiser helps people decide on a course of action they might embark on to improve the join performance of their queries. This talk presents useful information on the SQL Query Optimiser, and includes cases studies that demonstrate a "cookbook" strategy for improving SQL Join performance.

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. This allows users to specify what they want in the SQL query, without becoming overly involved in how to get it.

In effect the user allows PROC SQL to decide on the most appropriate processing strategy to satisfy the request — contrast this with less modern SAS code that is chock full of procedural details like PROC SORT steps.

Most SQL joins are ones that have an equality in the WHERE clause specifying which columns must match. PROC SQL can process these joins more efficiently than the general case. Joins that do have an equality like this in the WHERE clause are referred to as Equi-joins; those that do not are called Cartesian joins.

Equi-Joins

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. (Of course, we can avoid the sort for tables that have a known order that is useful)
- 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.

Not all Equi joins are one-to-many joins — SQL Join semantics require that all rows with a specific join key are compared with all rows from the other table having the same join key. The presence of an equijoin predicate in the WHERE clause just means that PROC SQL might have to consider many smaller Cartesian products instead of one large one.

Processing a few small Cartesian products is cheaper than processing the one large one. Assume:

- Two tables M and N with Mr and Nr rows respectively
- A join key K with Kv unique values

A complete Cartesian product product of M and N involves Mr X Nr compares.

Several (Kv) smaller Cartesian products (one for each unique value of K) involve Kv X ((Mr / Kv) X (Nr / Kv)) compares. This Simplifies to (Mr X Nr) / Kv compares.

Attaching some real numbers to this may help. Suppose two tables with 100 rows and 10 key values. A full Cartesian product involves 100 X 100 rows (10,000 compares). A Cartesian products on a set of records with the same key value involves 10 X 10 rows, but you have to do 10 of them (one for each group)
- This nets out to 1000 compares and is less work than the 10,000 compares required for the full Cartesian product.
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 within rows of 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.

Creating an index is not without cost, but at least it is a fixed cost that can be amortised over many queries.

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 one or both of the datasets are already sorted in a convenient sort order
Hash If one of the datasets will fit into memory. Actually, if 1% of the dataset will fit in a single memory extent whose size is the sql buffersize (64000 bytes)
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 Database 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=1 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:

<table>
<thead>
<tr>
<th>Module Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqxslct</td>
<td>Select</td>
</tr>
<tr>
<td>sqxjsl</td>
<td>Step Loop Join (Cartesian)</td>
</tr>
<tr>
<td>sqxjm</td>
<td>Merge Join</td>
</tr>
<tr>
<td>sqxjndx</td>
<td>Index Join</td>
</tr>
<tr>
<td>sqxjhsh</td>
<td>Hash Join</td>
</tr>
<tr>
<td>sqxsort</td>
<td>Sort</td>
</tr>
<tr>
<td>sqxsrc</td>
<td>Source Rows from table</td>
</tr>
<tr>
<td>sqxfil</td>
<td>Filter Rows</td>
</tr>
<tr>
<td>sqxsumg</td>
<td>Summary Statistics (with GROUP BY)</td>
</tr>
<tr>
<td>sqxsumn</td>
<td>Summary Statistics (not grouped)</td>
</tr>
<tr>
<td>sqxuniq</td>
<td>Distinct rows only</td>
</tr>
</tbody>
</table>

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

```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
   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
```

207
Indexes — not always a Silver Bullet

Many people are surprised by the poor performance of indexes when the number of records being extracted via the index is a significant fraction of the total. To help understand why this is so, I like to present the analogy of the postman delivering the mail. Instinct tells us that he sorts the mail in his mail bag into delivery address order and proceeds to deliver the mail in street order — he knows that the houses on his route have an order, and so by arranging his mail bag in the same order he can deliver the mail in sequence, and in a timely fashion.

If the mailman were to use an index (the street address) to avoid doing a sort of the mail bag, delivering the mail might go like this:

- Take the next envelope from the mail bag
- Locate the house by consulting a map (lookup up the record address in the index)
- Walk to that house (read the record)
- Deliver the envelope (output a matched row)
- Repeat until mail bag is empty (or until very tired!)

It is the cost of walking to the house for each envelope that makes this an unworkable method of delivering mail — it would only be practical for a very small mail bag (say on a very light mail day).

The postman probably uses the known sort order of the mail bag to his advantage to optimise his delivery. If he has just delivered mail to 101 main street, and the next mail is for 301 main street, he knows how many houses he can skip in the meantime. He may even be able to skip entire streets if he has just delivered mail to street A, and the next mail in the (sorted) mail bag is to street C — voila! no need to visit street B today.

Cookbook Questions to ask?

In order to improve the performance of your SQL queries, you should examine these situations and see if any of the "remedies" can be applied to your problem.

Questions to ask of the larger table

Can I make an index? This may help if you are joining a small table with the large indexed one. However, after a certain threshold (around 15%) of the large table accessed, it becomes more expensive to access via the index than to simply process the entire large dataset.

Can I indicate an order? You may be able to sort the large dataset — this will help if you make more than one query against it. If the dataset is not sorted, PROC SQL will have to sort it into an internal temporary file for each query that accesses it.

You may be extracting the data from an external file system whose order is well known. You should tell SAS of this order (with the SORTEDBY= dataset option) so that we do not perform redundant sorting inside PROC SQL.

Is this large table in a DBMS? There are many SUGI Papers that deal with this issue. Try [KENT], [LOREN], [SCOTT] for papers that focus on this subject. The member SQLJMAC from the SAS Sample Library implements one approach for extracting a list of keys to pass to the DBMS in a WHERE clause. After much experimentation I think the best approach is to have a SAS program generate another SAS program — this approach is shown in the examples towards the end of this paper.

Questions to ask of the smaller table

Can I build a list of unique keys? If you can, you may be able to use a technique that uses these keys to write an extended WHERE clause. This technique can be especially profitable if the larger table is in a DBMS.

Will the small table fit into memory? If it does, PROC SQL will select a hash join — you will know this by the presence of a sqxjsh in the output generated by the _method option. We will not select a hash join if our estimate shows that approximately 1% of the rows and columns from the dataset will not fit in one SQL buffer.

If your SAS job has lots of virtual memory available, you may be able to get PROC SQL to select hash join for larger tables by specifying the (heretofore undocumented)buffersize=SQL option on the PROC SQL statement - the default buffer size is 64000 bytes of memory.

Questions to ask of the WHERE Clause?

Can I fully specify the predicates? This may allow PROC SQL more freedom in determining the order in which to evaluate joins. However, in Release 6.11 of SAS Software PROC SQL will perform these transformations automatically. For example, the test for A.X=20 can be extended through the equijoin predicate to imply that B.Y=20 must also be true.

```
WHERE A.X=B.Y
AND A.X=20
/*internally added: AND B.Y=20*/
```

Can I avoid a Cartesian product of the entire table? It may be possible to re specify the query to avoid crossing all the rows. Sometimes this can be accomplished by preprocessing one of the tables to put it in a more suitable form. Sometimes the addition of another table to the query may result in less work needed to accomplish the solution.

Examples of Equi-joins

Small table joined to Large table

In this example, we explore improving the performance of the equijoin on a common variable KEY. The small table has about
800 records which is a very favorable ratio to the number of records (1 million) in the large table and should lead us to create an index on the key column of the large table.

First some test data and the obvious solution:

```plaintext
data small(drop=data) large;
  do key=l to 1000000;
    length data 12;
    data=put(key,words12.);
    if mod(key,1234)=0 then output small;
    output large;
  end;
proc sql _method;
  create table result as
    select large.key,large.data
    from small, large
    where small.key=large.key;
the _method output is:

This indicates that PROC SQL will load the small table into memory and process each row of the large table doing an in-memory table lookup to see if there is a match. On my HP735 with 64MB of memory, this query took 38.6 CPU seconds.

Creating an index is the solution here. The same query runs in 1.4 CPU seconds and the _method output is:

PROC SQL has chosen to sort both tables (on key) into temporary work space, and sort merge them. This was expensive! — it took 138.3 CPU seconds to process this query.

It may be that you know the order of the large file even though SAS does not. In our case we generated the large file with a do loop, but more likely you have just read it from a flat file that has an order. If this is the case, by all means let SAS know using the sortedby= dataset option.

PROC SQL has chosen to sort both tables (on key) into temporary work space, and sort merge them. This was expensive! — it took 138.3 CPU seconds to process this query.

If the small table were larger than 800 rows, you might see that PROC SQL had to resort to a sort-merge join, when it determines that the smaller table no longer fits into a hash table (that is, into memory). This would be quite expensive as it would have to sort the million row table as well as the small one. If we create a small table with 17500 observations (only a 1.75% subset), we see that the hash join is no longer selected.

Not so Small table joined to Large table

This query took 47.9 CPU seconds. We spent a large fraction of the time sorting the large table unnecessarily in the previous case!

There is still room for improvement. A hit-rate of 1.75% between the small table and the large one should prompt us to investigate creating an index.
create index key on large(key);
create table result as
select large.key,large.data
from work.small4, large
where small4.key=large.key;
drop index key from large;

/* methods chosen */
\sqxcrt\n\sqxjndx
\sqxsnc
\sqxsrc

This query took 18.7 CPU seconds (although it did take 185.8
CPU seconds to create the index). Obviously, if this query is run
only once against the large dataset then we have wasted resources
by creating the index. If we run many different "small" tables
against the large one, then creating an index is a win.

Just as an experiment, I upped the SQL buffer size to 256K (from
its default of 64K). This allows the small table to fit into a
hash-table in memory.

reset buffersize=256000;
create table result as
select large.key,large.data
from work.small4, large
where small4.key=large.key;

/* methods chosen */
\sqxcrt\n\sqxjhsh
\sqxsnc
\sqxsrc

This query took 37 CPU seconds. You may be able to up the
buffersize parameter in your SQL queries to allow tables to fit into
memory, but you should benchmark this on your computer —
sometimes you'll just swamp the paging ability of your system and
not achieve the gains demonstrated here.

The hash join is faster than the sort-merge even though the sort
could have completed in memory. This is due to the overhead in
interfacing to SAS sort method (which includes the hooks to allow
the user to substitute a host sort routine).

Small table joined to Large table in DBMS

Use the same tables as in the previous example, but assume that
the larger table lives in a DBMS supported by SAS/ACCESS
Software. This SAS code will "write another SAS program" that
contains the key values from the small table as hard coded values
in WHERE clauses for the large table.

Experience shows that doing this in "chunks" of 100 or so keys
at a time is the best approach - many DBMS optimizers will not
select an index if the IN clause contains more than that many
values. I have left the "chunk" size as a macro variable to make
it easy for you to experiment with this — and experiment you
should! The optimal number will vary from DBMS to DBMS,
from table to table, and will be influenced by the length of the key
being matched on.

This code just sets up a list of unique key values.

%let chunk=105;
proc sql;
create view uniq as
select unique key
from small
order by key;

data null;
file temp;
set uniq end=end;
if ~n_ = 1 then do;
put "create table result as"
/ "select key,data"
/ "from connection to dbrns~(select key,data"
/ "from large where key in(*
/ key;
end;

When processing the first chunk we want to use a create table
statement to create the result, so our program (which is writing a
SAS program) outputs:

else if mod(~n_,&chunk) = 0
and not end then do;
put "
/ "insert into result"
/ "select key,data"
/ "from connection to dbms"
/ "(select key,data"
/ "from large where key in(*
/ key;
end;

When we get to the end of a "chunk" of keys, we close out that
statement. If we need to start another "chunk" (i.e. we are not at
then end of the keys) we want to insert the next batch of returned
rows into the result that is already being accumulated.

else if end then do;
p
put "
/ "select key,data"
/ "from connection to dbms"
/ "(select key,data"
/ "from large where key in(*
/ key;
end;

Usually, however, we are in the middle of a chunk of keys, so all
we add to the program under construction is the value of this key.
in a form that the IN clause is expecting. If you adopt this example and have a character variable as your key, don’t forget to place quotes around the values.

```sql
else put key '"',":
run;
```

Finally, we run the program that we have built.

```sql
proc sql;
connect to <DBMS> as dbms;
%inc temp;
```

### Examples of Useful Cartesian Joins

Cartesian Joins often solve real-world problems in elegant ways. The challenge is to recast the SQL to solve them efficiently. I have collected these examples from my contacts with SAS users:

#### Nearest Neighbors problem

A researcher at UNC said: I have a query about DATA steps that is strange enough to have baffled not only the SAS tech support folks, but also Sally Muller. I am writing a macro to perform adaptive bandwidth kernel fitting, and need to perform a large number of calculations of the form $F(X_i - X_j)$ where $F$ is some function, $X$ is my variable and $i$ & $j$ subscript observations. I do not need to compute this function for all possible $i$ & $j$ combinations, only those where $|X_i - X_j| < 2.3$ ($X$ is sorted into increasing order). My general question is how best to set this up: this kernel fitting is part of a complex bootstrapping problem, so it will run 50,000-75,000 times with 15 to 5000 observations each time.

This sounds like a good application of SQL. Finding all combinations of rows such that the difference in variable $X$ is less than 2.3 apart in $X$ value. If we convert $X$ to an integer, we only have to consider 4 cases $X_1 = X_2 - 3$ thru $X_1 = X_2$. We can use a Cartesian join to create these four values and then compare for equality. This factors out to 5000 * 4 compares to compute the four potential offsets followed by a 5000 * 20000 equijoin.

First we output a table of the offsets:

```sql
data offset:
do offset = 0 to 3;
output;
end;
run;
```

Then we select using this table. PROC SQL will form an intermediate result for the x2_offset join, and join that intermediate result with x1.

The genesis of the final solution is due to Howard Schreier - HIS@cu.nih.gov. He notes the seeming paradox that making the query more complicated seems to be a good way to reduce its runtime.

Let us start with the obvious SQL formulation. We don’t have to consider the cases where $X_1$ is less than $X_2$ (by as much as 2.3) as the Cartesian product considers both $X_i$ with $X_j$ as well as $X_j$ with $X_i$.

```sql
proc sql _method stimer:
create table results as
select x1.i as i1,
x1.x as x1,
x2.i as i2,
x2.x as x2
from testdata x1,
testdata x2
where x1.x - x2.x
between 0 and 2.3;
```

Whew! that was a lot of work. It took 537 CPU seconds on my (nice, fast) HP735. How can we reduce the Cartesian product requirement from 5000 * 5000 (25 million compares)? The answer lies in augmenting the LESS THAN predicate with an equijoin that allows us to test fewer combinations - the equijoin predicate does not need to be exact, we can always "fine tune" the result set with the existing WHERE clause.

The clue to the solution is that records must be no more than 2.3 apart in $X$ value. If we convert $X$ to an integer, we only have to consider 4 cases $X_1 = X_2 - 3$ thru $X_1 = X_2$. We can use a Cartesian join to create these four values and then compare for equality. This factors out to 5000 * 4 compares to compute the four potential offsets followed by a 5000 * 20000 equijoin.
proc sql _method stimer;
create table work.results as
select xl.i as i1,
xl.x as xl,
x2.i as i2,
x2.x as x2
from testdata xl,
testdata x2,
offset
where xl.x = x2.x
between 0 and 2.3
and floor(xl.x) + offset = floor(x2.x);
This ran in 41.2 CPU seconds. The key to this faster solution is
to convert the Cartesian join over all rows to an equijoin over
slightly more rows, but with a \texttt{WHERE} clause that limits the
matching that is required. If the two X values had to be within 100
of each other, it may have become prohibitive to consider all 101
offsets, but you could modify the solution to compute the offset
to the nearest 10 using the \texttt{ROUND} function.
I don’t have a name for this kind of optimization, but as a general
principle, any time you can reduce the search space of potential
answers you can probably reduce the resources required to find
the solution.

\textbf{Date Ranges}

A PROC SQL Supporter on SAS-L says:
\textit{I have two files I want to match. One is of bills, the other is of
discounts:}

\begin{tabular}{|c|c|c|}
\hline
\textbf{BILL} & \textbf{DATE} & \textbf{STATE} & \textbf{AMOUNT} \\
\hline
1 & 01jan94 & CA & 100 \\
2 & 01dec93 & CA & 200 \\
3 & 15jan94 & KY & 80 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{START} & \textbf{END} & \textbf{STATE} & \textbf{DISCOUNT} \\
\hline
01jan94 & 01feb94 & CA & .5 \\
15feb94 & 15mar94 & NC & .4 \\
15mar94 & 15apr94 & NC & .3 \\
01jan93 & 31dec93 & & .22 \\
01jan94 & 01feb94 & & .25 \\
01feb94 & 31dec94 & & .7 \\
\hline
\end{tabular}

For each bill, I want to find a matching discount record. First I
look for a state match, and if there is no state match, I look for a
blank state in the discounts file. The bill date has to be within the
start and end dates of the discount record. This billing data has
millions of records.

This is currently done in a batch DBMS job, which uses SQL and
some host language. That program checks one record at a time
and handles mismatches as they occur, something like:

\begin{itemize}
\item get a discount record where the date matches
\item if OK then \texttt{newbill = amount * discount; go to next record}
\item oops, error!
\end{itemize}

We sometimes want to reproduce this program in SAS, but there
doesn’t seem to be a way to do the exception handling.
I created some test data (included so that you can play with these
eamples once you are at your computer).

\begin{verbatim}
data billing;
length state $2;
input date date7. +1 bill
   state $ amount;
   format date date7. ;
cards;
 01jan94 1 CA 100
 01dec93 2 CA 200
15jan94 3 KY 80
01jan94 4 NC 1000
01feb94 5 NC 1000
01mar94 6 NC 1000
01apr94 7 NC 1000
01may94 8 NC 1000
run;
\end{verbatim}

\begin{verbatim}
data discount;
input start date? +1 end date? +1 state $2.+1 discount;
format start end date? ;
cards;
01jan94 01feb94 CA .5
15feb94 15mar94 NC .4
15mar94 15apr94 NC .3
01jan93 31dec93 .22
01jan94 01feb94 .25
01feb94 31dec94 .7
run;
\end{verbatim}

The original solution goes like this.... First we compute
discounts for those bills that have a state specific discount

\begin{verbatim}
create table stmatch as
select distinct *,
   b.amount*d.discount as newbill
from billing b, discount d
where b.state = d.state
   and b.date >= d.start
   and b.date <= d.end;
\end{verbatim}

Then we compute discounts for those bills that had no state
specific discount and so the general discount should apply.
create table blmatch as
select distinct *
from billing b, discount d
where b.amount * d.discount as newbill
and b.date >= d.start
and b.date <= d.end
and b.bill not in
(select bill
from stmatch)
/* display results */
select * from stmatch
union all
select * from blmatch;

The solution is not satisfactory because the second create table has to check and see if the particular bill has matched a state specific discount before matching the general (state is blank) case.

My advice is to turn the discounts dataset upside down, and do the error checking up front. Instead of storing a range for which the discount is valid, store each day and its discount. (yup, you guessed — turn a Cartesian join into an equijoin). An added benefit is that you can have the state specific discount and the general discount on the same row — this will help avoid two queries later.

One drawback of my advice (besides it being worth what you paid for it) is that we do need to know the extent of the dates in the billing and discount datasets. Since this is just an example, I’ll assume we know them up front :-) We also need the list of unique states, which I get from a pass over the billings dataset, but in practice you’ll know this list, won’t you?

First we compute a table with all the dates in the range we are considering:

%let mindate = '01jan93';
%let maxdate = '01jan94';

data dates;
  do date = &mindate to &maxdate;
    output;
  end;
  format date date7.;
run;

Now we extract per-state information from the discount table:

%let mindate = '01jan93';
%let maxdate = '01jan94';

data dates;
  do date = &mindate to &maxdate;
    output;
  end;
  format date date7.;
run;

proc sql _method stiroer _tree;
/* get the uniq state names;*/
create view stname as
  select unique state
  from billing;

/* get the state specific discounts;*/
create view states as
  select state,
    start as s_start,
    end as s_end,
    discount as s_disc
  from discount
where state = '

When we calculate the general discounts, we need to "expand" them so that they apply to all states in the billing dataset. Here I join them with the stname view, but in practice you’ll probably have a validation table that lists valid states.

/* get the general discounts;*/
create view general as
  select s.state,
    start as g_start,
    end as g_end,
    discount as g_disc
  from discount d,
  stname s
where d.state = '

So now put that all together with a left join to make sure that you don’t lose any of the days in the window that we are considering. The original request needed error validation — you could check that a general discount was available for every date in the range at this point. In other words, do the error check "up front".

/* create the daily discount lookup table;*/
create table tempdisc as
  select date, g.state,
    s_disc, g_disc
  from dates d
  left join general g
  on date between g_start
  and g_end
  left join states s
  on date between s_start
  and s_end
  and g.state = s.state

Now we have daily discount rates and the problems reverts back to one of joining a large dataset (billings) with a smaller one (discounts) which has daily discount rates.
We can use the COALESCE function to choose an appropriate discount.

```sql
create table results as
select b.*,
    coalesce(s_disc, g_disc, 1) as disc,
    b.amount * calculated disc
    as newbill
from billings b,
    tempdisc t
where b.date = t.date
and b.state = t.state;
```

The tempdisc data is approximately 37000 observations of say 25 bytes. This will probably fit into an in memory table, so the join is resolved with a single pass over the huge million row billings table.

**Fuzzy Logic and SQL.**

An excellent paper (Best Contributing Paper, [CORELLE]) was presented at SUGI 17. The task at hand was to merge two tables collected by different institutions. There were many "almost" matches, but very few exact hits. The authors of this paper identified several matching criteria, and consider two records a match if they meet some but not all of the criteria.

They rank some criteria more important than others, and exploit the identity that logical expressions are always 1 or 0 by multiplying the logical expression by the "importance" of the criteria. If an observation meets enough criteria so as to raise its "match score" above a threshold, it is considered a keeper.

They solve the problem with a Cartesian Join whose search space is trimmed down by an equality check:

```sql
select *
from file1 a, file2 b
where a.sex = b.sex
and ( (a.day = b.day) * 2 )
+ ( (a.mon = b.mon) * 2 )
+ ( (a.yr = b.yr) * 2 )
+ ( (a.fnm = b.fnm) * 1 )
+ ( (a.lnm = b.lnm) * 1 )
) >= 6;
```

In this case, the equijoin predicate partitions the solution set into two groups — this is not much of a saving in terms of the numbers of records that need to be crossed to evaluate the Cartesian product. PROC SQL will:

- load a single buffer page of observations from filea
- load as many observations as can fit into available memory from fileb
- consider all interactions of the rows loaded

If all the rows from filea could not fit into a single buffer page during the first step above, we will reload that page with more observations and repeat the process.

If you encounter a SQL Join that warns "NOTE: The execution of this query involves performing one or more Cartesian product joins that cannot be optimized," you may be able to improve its performance by increasing the `buffersize =` PROC SQL option.

### A Note on the buffersize option

We have not documented this option for several reasons:

- It does no validity checks on the value you supply.
- We have not tested the impact on performance of raising the value across all operating systems and in conjunction with SAS Share servers.

In other words, please run some tests on your data to see if it is a useful option to specify in your situation.

PROC SQL will use approximately 1M of memory per hash join with the default `buffersize=64000`. If you increase it to `buffersize=128000` we'll use approximately 1.6M, and 3.2M for a `buffersize=256000`.

PROC SQL uses buffer pages for other processing too. Increasing the size of a buffer page using this option may mean that you need to increase the total memory available to your SAS Session (using the `memsize` SAS Invocation option on UNIX and PC Systems, for example).

### References

Trademarks

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

Your Turn

The author can be reached care of SAS Institute Inc., or at the Internet E-mail Address keni@unix.sas.com

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