Paper 73-27

Creating Efficient SQL – Four Steps to a Quick Query

Paul D Sherman, IBM Corporation, San Jose, CA 95193

ABSTRACT

Schema Query Language is a very flexible, general grammar for fetching and processing large amounts of data directly at the database level. Given this flexibility, there are many ways of achieving the same result. Some approaches yield terribly poor database performance especially when simple processing steps are overlooked. Most notable are incorrect uses of the ORDER BY and DISTINCT statements, whose often desired results almost always lead to sub-optimal database behavior. It is instructive to think about any “table”, which are named in the FROM clause, as just another type of “object” in an object-oriented sense. Tables consist of one or more fields or columns, each having well defined types, and when related or joined to each other hold references between themselves. An SQL statement is thus nothing more than an object model diagram of sorts. In that regard, I recommend the following sequence of four simple steps when building any SQL query statement.

INTRODUCTION

Proc SQL offers the SAS programmer a powerful tool for handling datasets. Even more useful is the pass-through extension allowing direct communication to an external database system directly from within a SAS program. With a few simple steps in mind, one can easily design SQL statements which maximize use of a remote or central database, while minimizing load and processing time on the local workstation.

```
proc sql noprint;
   connect to odbc (dsn=&d. uid=&u. pwd=&p.);
      create table &t. as
      select * from connection to odbc (
         /* SQL statement goes here */
      );
   disconnect from odbc;
   quit;
```

Why will we study only the SELECT statement? Although SQL is rich with both content inquiring and content affecting transactions, we focus herein on the former since fullselect grammar is more complex, more routinely utilized, and can often be applied as an argument value to the latter. Much insight will be gained through mastery of the content inquiring transaction or SELECT statement.

THE FOUR STEPS

The four steps to build an SQL query are shown below, general for any application.

1. Choose your data source ‘objects’ and write the FROM clause
2. Specify the relationships with the ON clause in each but first table
3. Add desired filtering to the first table, using the WHERE clause
4. Customize the output by sculpting the SELECT clause

(5). Optionally call for sorting and group-rollup aggregation using the ORDER BY or GROUP BY clauses.

COMPONENTS OF A SELECT STATEMENT

In order to emphasize the importance of each statement clause – and their development sequence – we’ll examine each clause separately in detail. An SQL query statement consists of four processing steps: Data fetch, data filter, sort and report.

DATA FETCH – THE <u>FROM</u> CLAUSE

Tables, value lists or other data source items are specified here. There are two popular structures, the implicit and explicit relationship forms. Although an implicit relationship form is widely used today, where tables are listed simply separated by commas and without any relation-expressions, such form leads to very hard to read SQL and possibly unoptimized code. The explicit relationship form as demonstrated in the following example, however, clearly defines how each data object is related.

```
Implicit   Explicit
table_a,   table_a
table_b,   inner join table_b on a.k1 = b.k1
table_c    inner join table_c on a.k1 = c.k1
```

When laying out any multi-table query, one must decide beforehand which table will drive or ‘pivot’ the others, producing results independent of any other relations. This ‘pivoting’ table is always listed first, and as such needs no relationship or “join” method, nor any relationship or “on” clause. All remaining tables follow, preceded by relation method and followed by relation expression.

```
<pivot-table>
<r-method> JOIN <table> ON <r-expression>
```

For example,

```
table_a as a
inner join table_b as b on a.k1 = b.k1 …
left outer join table_c on a.k1 = c.k1 …
```

Trouble follows when table_a does not have any rows returned for either table_b or table_c. Such yields a dangling object and often Cartesian product or database timeout, and almost always is due to a bad choice of pivot table, which instead should be outer-joined to one of the other table objects.

DATA FILTER – THE <u>WHERE</u> CLAUSE

Conditions much like a relationship expression are specified here, though most often are equalities or inequalities referred to constants or final values. Using the explicit relationship form, however, a well-designed SQL statement need only place here expressions of the pivot table alone, since only the pivot table lacks relation expression and relation method. In fact, incorporating the additional filtering into the pivot table as a subquery in the FROM clause, high-performance SQL statements need no WHERE clause, except in rare cases when post-join filtering is desired.

REPORTING – THE <u>SELECT</u> CLAUSE

As Frank Lloyd Wright once said, “Form follows function.” So true to dealing with data, one makes good sense to specify which and where data elements will be bubbled up to the outer levels of the query model. A properly designed model will nearly write its own choices for the SELECT clause. Further, one may even decide later, after table objects are built, joined and grouped, to include additional data values previously unconsidered – here is the creation of insight into your data. There is a time and a place for everything, and the time to write the SELECT clause comes *after* FROM and WHERE.

SORTING – THE <u>GROUP BY</u> AND <u>ORDER BY</u> CLAUSES

This optional step is probably the most misused clause, and most responsible for poor SQL statement response, database performance and needless database administrator table re-org effort.

Quite often one qualifies the SELECT clause with the DISTINCT statement, guaranteeing that no two rows will have the same column values over the desired unique keys. Internally, however, many database systems implement the DISTINCT function as a GROUP BY aggregation over all columns. Furthermore, this aggregation is performed last, outside of any other group operation. When one's SQL statement is a single fullselect all is good and results are as intended. Using DISTINCT on a subquery is where the trouble starts – the subquery becomes a simple fullselect statement, returning all rows whether distinct or not, then only after being joined into the outer fullselect are non-unique rows filtered. Finding this kind of error is very tedious, since the SQL statement is syntactically correct; quite often abnormally long database processing time, a difficult metric to measure, provides the only clue that something is wrong.

```
-WRONG-    => becomes =>   -this-
select a,b,c               select a,b,c
from foo                   from foo
where foo.k1 in (          where foo.k1 in (
        Select distinct x          select x
        From bar                   from bar
        Where …                    where …
      )                          )
                           group by a,b,c

-CORRECT-
select a,b,c
from foo
where foo.k1 in (
         select x
         from bar
         where …
         group by x
      )
```

EXAMPLE – COUNTING BEANS

Every good object model begins from a problem statement. Here we look to farmer Jack, who is growing various species of beans in his garden. His plant stalks are becoming quite tall and he's beginning to notice that some of his crop yield is declining. Jack suspects a hungry giant is eating his beans, since only the large, fresh beans are disappearing from the tops of the stalks. Jack wants to quantify how many beans of each type are vulnerable at the tops of his plants.

STEP 1: DESCRIBE THE MODEL

Four data source objects describe the model: Bean.Genus lists all the bean type names and their properties, Bean.Bag is an inventory of which bean is on what stalk, Bean.Stalk details the individual branches of each stalk, and Stalk.Branch further describes each branch of a stalk. Pictorially, they are shown below:

```
+--------+  +--------+  +--------+
| BEAN   |  | BEAN   |  | BEAN   |  +--------+
| GENUS  |  | STALK  |  | BAG    |  | STALK  |
+--------+  +--------+  +--------+  | BRANCH |
| STALK  |  | STALK  |  | BEANID |  +--------+
| SIZE   |  | BRANCH |  | BRANCH |  | BRANCH |
| SHAPE  |  | STEM   |  | TIMEST |  | DESC   |
| WEIGHT |  +--------+  | NAME   |  | NPODS  |
| COLOR  |              +--------+  +--------+
| STATE  |
| NAME   |
+--------+
```

The FROM clause is easily created, simply listing each table member. Notice our choice for the leading or 'pivot' table as the Bean.Genus. Since we desire a count of beans for each Bean.Genus bean type name, it makes good sense for the Bean.Genus to control the other table members. We have renamed the tables with concise names, using AS, for convenience.

```
FROM  bean.genus AS genus
      bean.stalk AS stalk
      bean.bag AS bag
      stalk.branch AS branch
```

STEP 2: CONNECT THE OBJECTS

Relate the table members to each other as makes sense to the problem statement and object model. There are two types of relations commonly used: Dynamic and Static. A dynamic relation holds references between fields of two table objects, while a static relation simply fixes a field of one table object to a constant value. Both types of relations are equally valid for the ON clause expression.

```
+--------+  +--------+  +--------+
| BEAN   |  | BEAN   |  | BEAN   |  +--------+
| GENUS  |  | STALK  |  | BAG    |  | STALK  |
+--------+  +--------+  +--------+  | BRANCH |
| STALK  |==| STALK  |  | BEANID |  +--------+
| SIZE   |  | BRANCH |==| BRANCH |==| BRANCH |
| SHAPE  ||=| STEM   |  | TIMEST |  | DESC   |
| WEIGHT || +--------+|=| NAME   |  | NPODS  |
| COLOR  ||==[null]   | +--------+  +--------+
| STATE  |            |
| NAME   |============|
+--------+
```

There should be as many expressions in the ON clauses as there are relational lines drawn in the object model. Often the placement of a relation-expression is arbitrary; here we could just as equally placed `stalk.branch = bag.branch` on the second (stalk) table, with no loss of generality or query performance. Read the object model in one direction throughout the entire relation extraction process.

```
FROM  bean.genus AS genus
      INNER JOIN bean.stalk AS stalk
         ON genus.stalk = stalk.stalk
         AND stalk.stem = null
      INNER JOIN bean.bag AS bag
         ON genus.name = bag.name
         AND stalk.branch = bag.branch
      INNER JOIN stalk.branch AS branch
         ON bag.branch = branch.branch
```

STEP 3: FILTER THE PIVOT

Perform the same table relation as above, only now on the first or 'pivot' table which has no ON clause. Here we limit the extent of all Bean.Genus types to those which have large size, as farmer Jack observes those are the only types being eaten.

```
  +--------+  +--------+  +--------+
  | BEAN   |  | BEAN   |  | BEAN   |  +--------+
  | GENUS  |  | STALK  |  | BAG    |  | STALK  |
  +--------+  +--------+  +--------+  | BRANCH |
  | STALK  |==| STALK  |  | BEANID |  +--------+
|=| SIZE   |  | BRANCH |==| BRANCH |==| BRANCH |
| | SHAPE  ||=| STEM   |  | TIMEST |  | DESC   |
| | WEIGHT || +--------+|=| NAME   |  | NPODS  |
| | COLOR  ||==[null]   | +--------+  +--------+
| | STATE  |            |
| | NAME   |============|
| +--------+
|
|==['LARGE']
```

Only the first, or pivot table's relations should be listed in the WHERE clause data filter. In fact, these should only be static relations, since dynamic relations linking the pivot table to another should have already been defined above, in one of the other table's ON clauses.

```
FROM  bean.genus AS genus
      INNER JOIN bean.stalk AS stalk
        ON genus.stalk = stalk.stalk
        AND stalk.stem = null
      INNER JOIN bean.bag AS bag
        ON genus.name = bag.name
        AND stalk.branch = bag.branch
```

```
         INNER JOIN stalk.branch AS branch
           ON bag.branch = branch.branch
   WHERE genus.size = 'LARGE'
```

STEP 4: CHOOSE THE INFORMATION
Now that all the table objects have been defined and connected, choose the fields which are of interest, labeled with <> brackets. In this example, Jack desires counting beans only by their type, though he could just as easily look at their properties, stalk type or branch position as well.

```
  +--------+  +--------+  +--------+
  | BEAN   |  | BEAN   |  | BEAN   |  +--------+
  | GENUS  |  | STALK  |  | BAG    |  | STALK  |
  +--------+  +--------+  +--------+  | BRANCH |
  | STALK  |==| STALK  | *| BEANID |  +--------+
|=| SIZE   |  | BRANCH |==| BRANCH |==| BRANCH |
| | SHAPE  ||=| STEM   |  | TIMEST |=|| DESC   |
| | WEIGHT || +--------+|=| NAME   | || NPODS  |
| | COLOR  ||==[null]   |  +--------+ |+--------+
| | STATE  |            |             |
| | NAME   |============+             |=[t-1days]
| +--------+            |==<NAME>
|
|==['LARGE']
```

Each item or row of the SELECT clause will be a column or field in the output table resultant data set.

```
   SELECT genus.name
   FROM   bean.genus AS genus
         INNER JOIN bean.stalk AS stalk
           ON genus.stalk = stalk.stalk
           AND stalk.stem = null
         INNER JOIN bean.bag AS bag
           ON genus.bane = bag.name
           AND stalk.branch = bag.branch
         INNER JOIN stalk.branch AS branch
           ON bag.branch = branch.branch
   WHERE genus.size = 'LARGE'
```

STEP 5: SORT AND SUMMARIZE
Specify only those fields over which aggregation, and for which a unique dataset, is desired. Any column field named in the SELECT clause must appear either in the GROUP BY clause, or as the argument of a grouping function such as min(), max(), avg() or count(). The HAVING clause, which follows the aggregation, is simply just another kind of data filter. Therefore one should think about WHERE and HAVING as pre- and post-aggregation filters; only the latter requires all of its expressions to be function arguments. After the HAVING clause filter, we re-sort the data set into descending sequence of bean count, therefore listing first for Jack those bean types having the greatest number of large beans at the top of the stalk within the most recent day.

```
SELECT genus.name,
       count(bag.beanid) as numbeans
FROM  bean.genus AS genus
      INNER JOIN bean.stalk AS stalk
        ON genus.stalk = stalk.stalk
        AND stalk.stem = null
      INNER JOIN bean.bag AS bag
        ON genus.name = bag.name
        AND stalk.branch = bag.branch
      INNER JOIN stalk.branch AS branch
        ON bag.branch = branch.branch
WHERE genus.size = 'LARGE'
GROUP BY genus.name
HAVING min(bag.timest)>=current timestamp-1 day
ORDER BY numbeans desc
```

Using data from the appendix below, within the most recent day there are eight vulnerable beans at the tops of farmer Jack's plants, and the 'WAX' bean type is most available to the giant:

```
NAME   NUMBEANS
WAX    5
LIMA   3
```

The above SQL statement, its data summarization, filtering and sorting have all been performed at the remote or central database, and have not occupied any local processing time. Further, since each table object was carefully introduced and related to each other, we are guaranteed of error-free output, without the possibility of Cartesian data sets or unexpected loss of database performance.

CONCLUSION

Applying the same rules of object-oriented design to SQL query development greatly enhances both code readability and query performance. Deciding which table objects are needed, and how they should relate, leads one to an optimum design. After the query model is established, quite often a column field previously unconsidered will appear useful. Paying careful attention to when the data set should be sorted or forced unique eliminates lengthy processing time and unexpected database behavior. Properly using ORDER BY and GROUP BY can even eliminate that Proc Sort and/or nodupkey step which often consumes enormous local computing resources. When building SQL, remember four simple steps and the best sequence for them: FROM, ON, WHERE and SELECT.

ACKNOWLEDGMENTS

I am grateful to Rob Strong for many helpful discussions on database design and relational methodology. I thank Sysouk Sananikone for providing query performance measurements and wonderful dba assistance. I express deep gratitude to colleague, teacher and friend Thomas Poliquin, who has motivated and succeeded in getting me to think in terms of objects; and to Annie Perlin for the gene of creativity. Finally, I thank IBM Corporation for providing the tools and opportunity which made this work possible.

TRADEMARK CITATION

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Paul D Sherman
IBM Corporation
5600 Cottle Road
San Jose, CA 95193
Phone: 408-256-6091
Fax: 408-256-2700
Email: shermanp@us.ibm.com

APPENDIX: FARMER JACK'S DATA

BEAN.GENUS

NAME	SIZE	STALK	COLOR
LIMA	LARGE	Fiberous	White
PINTO	Medium	Hollow	White
KIDNEY	Medium	Solid	Red
GARBANZO	Medium	Vine	OffWhite
GREEN	LARGE	Fiberous	Green
RED	Medium	Hollow	Red
BLACK	Medium	Solid	Black
SOY	small	Vine	OffWhite
MUNG	small	Fiberous	Yellow
COFFEE	Medium	Hollow	Brown
VANILLA	Medium	Solid	OffWhite
JELLY	Medium	Vine	Multi
STRING	LARGE	Fiberous	Green
WAX	LARGE	Hollow	OffWhite
JAVA	Medium	Solid	Brown

BEAN.STALK

STALK	BRANCH	STEM
Fiberous	A	B
Fiberous	B	C
Fiberous	C	D
Fiberous	D	E
Fiberous	E	.
Hollow	A	C
Hollow	C	D
Hollow	D	E
Hollow	E	F
Hollow	F	.
Solid	A	B
Solid	B	C
Solid	C	D
Solid	D	E
Solid	E	.
Vine	A	B
Vine	B	C
Vine	C	D
Vine	D	E
Vine	E	F
Vine	F	.

STALK.BRANCH

BRANCH	DESC	NPODS
A	First Branch	3
B	Second Branch	4
C	Third Branch	4
D	Fourth Branc	5
E	Fifth Branch	3
F	Sixth Branch	3

BEAN.BAG

BEANID	BRANCH	TIMEST	NAME
328002	C	2002-01-05-18.11.01	LIMA
984002	E	2002-01-05-18.11.05	LIMA
937002	A	2002-01-05-18.11.09	LIMA
500002	B	2002-01-05-18.11.13	LIMA
968002	A	2002-01-05-18.11.16	GARBANZO
171002	D	2002-01-05-18.11.20	GARBANZO
687002	C	2002-01-05-18.11.23	GARBANZO
937001	B	2002-01-05-18.11.26	GARBANZO
718001	C	2002-01-05-18.20.47	LIMA
390000	C	2002-01-05-18.20.49	LIMA
140000	E	2002-01-05-18.20.58	LIMA
500000	E	2002-01-05-18.21.00	LIMA
265001	A	2002-01-05-18.21.05	LIMA
343000	A	2002-01-05-18.21.07	LIMA
593001	B	2002-01-05-18.21.10	LIMA
468000	B	2002-01-05-18.21.12	LIMA
109000	A	2002-01-05-18.21.16	GARBANZO
796000	A	2002-01-05-18.21.17	GARBANZO
968000	D	2002-01-05-18.21.20	GARBANZO
312000	D	2002-01-05-18.21.24	GARBANZO
296001	C	2002-01-05-18.21.27	GARBANZO
296002	C	2002-01-05-18.21.29	GARBANZO
46001	B	2002-01-05-18.21.32	GARBANZO
453000	B	2002-01-05-18.21.33	GARBANZO
140001	C	2002-01-05-18.22.52	LIMA
671001	C	2002-01-05-18.22.58	GARBANZO
171001	B	2002-01-05-18.22.04	LIMA
562001	B	2002-01-05-18.22.11	GARBANZO
203001	D	2002-01-05-18.27.28	GARBANZO
31000	D	2002-01-05-18.27.31	GARBANZO
796002	F	2002-01-05-18.27.48	WAX
812002	D	2002-01-05-18.27.52	WAX
218002	C	2002-01-05-18.28.56	WAX
390002	B	2002-01-05-18.28.59	WAX
562002	B	2002-01-05-18.28.02	WAX
578000	D	2002-01-05-18.28.08	WAX
750001	D	2002-01-05-18.28.11	WAX
859002	F	2002-01-05-18.28.16	WAX
500001	F	2002-01-05-18.28.21	WAX
875001	D	2002-01-05-18.28.24	WAX
687001	C	2002-01-05-18.28.28	WAX
531000	B	2002-01-05-18.28.34	WAX
734001	B	2002-01-05-18.28.37	WAX
859001	D	2002-01-05-18.28.40	WAX
843001	D	2002-01-05-18.28.43	WAX
953001	F	2002-01-05-18.28.46	WAX
250001	D	2002-01-05-18.28.51	WAX
109001	C	2002-01-05-18.28.54	WAX
312001	B	2002-01-05-18.28.57	WAX
843000	B	2002-01-05-18.29.01	WAX
781001	D	2002-01-05-18.29.04	WAX
406001	D	2002-01-05-18.29.10	WAX
515000	F	2002-01-05-18.29.15	WAX
906001	D	2002-01-05-18.29.18	WAX

DATABASE DEFINITION STATEMENTS

```
create schema bean;

create table bean.genus (
   stalk char(16) not null,
   size char(8),
   shape char(8),
   weight char(8),
   color char(8),
   state char(8),
   name char(8) not null,
   primary key (name, stalk, state)
);

create table bean.stalk (
   stalk char(16) not null,
   branch char(8) not null,
   stem char(8),
   primary key (stalk, branch)
);

create table bean.bag (
   beanid integer not null,
   branch char(8),
   timest timestamp,
   name char(8),
   primary key (beanid)
);

create schema stalk;

create table stalk.branch (
   branch char(8) not null,
   desc char(32),
   npods integer,
   primary key (branch)
);

insert into bean.genus (name, color, size, shape, stalk, state) values
   ('LIMA', 'WHITE', 'LARGE', 'FLAT', 'Fiberous', 'Growing'),
    …
;

insert into bean.stalk (stalk, branch, stem) values
   ('Fiberous', 'A', 'B'),
    …
   ('Hollow', 'A', 'C'),
    …
   ('Solid', 'A', 'B'),
    …
   ('Vine', 'A', 'B'),
    …
;

insert into stalk.branch (branch, desc, npods) values
   ('A', 'First Branch', 3),
   ('B', 'Second Branch', 4),
   ('C', 'Third Branch', 4),
   ('D', 'Fourth Branch', 5),
   ('E', 'Fifth Branch', 3),
   ('F', 'Sixth Branch, 3)
;

insert into bean.bag (beanid, branch, timest, name) values
   (microsecond(current timestamp), 'C', current timestamp, 'LIMA'),
    …
;
```