

## Improving Maintenance and Performance of SQL queries

Bas van Bakel, OCS Consulting, Rosmalen, The Netherlands  
Rick Pagie, OCS Consulting, Rosmalen, The Netherlands

### ABSTRACT

Almost all programmers and a lot of end users are familiar with the term SQL or even with writing SQL queries. The challenge comes with the querying and joining of numerous large tables; together with the performance, which drops rapidly, the maintainability decreases and debugging can become very complicated.

This paper will introduce ground rules for a method of programming your SQL queries in a way that they are easy to maintain and debug. It will also show that a query developed in the proposed way will improve the performance enormously compared to other queries.

### INTRODUCTION

This paper relates to the PROC SQL procedure to select data from a number of tables, readers are assumed to be familiar with the SQL syntax.

In this paper the EXPLAIN command, the EXPLAIN PLAN statement and the IDXWHERE and IDXNAME options for indexes are mentioned. The EXPLAIN command can be used e.g. with IBM Query Management Facility (QMF) or with SAS/Access Interface to Teradata to find out how the SQL query uses indexes and tables. A similar statement, the EXPLAIN PLAN statement, can be used in Oracle. The IDXWHERE and IDXNAME options can be used within SAS and the idea behind these options, the EXPLAIN command and the EXPLAIN PLAN statement is to be sure the query uses the indexes where and whenever possible.

### TRADITIONAL SQL PROGRAMMING

Almost all programming languages support SQL and therefore nearly all programmers will eventually come in contact with SQL based queries. Although the number of statements one can use in SQL is very limited, it is very easy to produce 'spaghetti'-code and code that performs badly.

Sometimes reducing the maintenance costs of the developed SQL code can be achieved, simply by making your programming code clearer to other programmers. This can be achieved by introducing more comments in your code and by avoiding programming like

```
SELECT *  
FROM table a;
```

since in this way it is unclear which variables of the applicable table will be used in your query.

Often several tables are put in the FROM statement at the same time which can result in badly performing queries. Moreover, in the case of an error, it can be very difficult to find out what the reason for the error is:

```
SELECT a.var1, a.var2, b.var3, c.var4  
FROM table1 a, table2 b, table3 c  
WHERE a.var5 = b.var3 AND  
      a.var4 = c.var7 AND  
      (c.var2 = 'N' OR  
       b.var8 < a.var6);
```

### METHOD OF DEVELOPING SQL QUERIES; ENCAPSULATING JOINS

Reducing maintenance costs starts with the development of the code. If the code is developed in a concise way, you assure that maintenance will be made as easy as possible. This chapter will introduce a specific method of creating easy-to-maintain and easy-to-develop SQL code; the 'Encapsulating Join' method.

## PhUSE 2007

To demonstrate this method, data of a study in which the bodyweight of subjects is measured on several time points will be used. Suppose we would like to create a table in which the bodyweight of all subjects of a specific trial, the start date of treatment of those subjects and the description of the study are all available. The study description of all studies can be found in the dataset PROJECT in which one record per study exists. The dataset DEMO contains for each subject in each study the start date of treatment and the table BODYWEIG contains for each study, for each subject, the bodyweight for several time points.

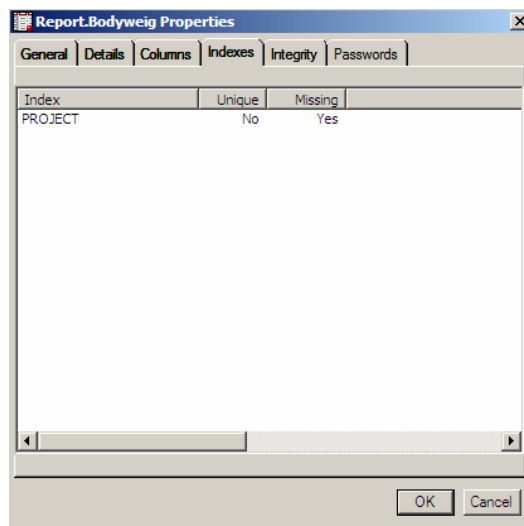
With the help of the following 'ground rules' the requested table will be generated according to the encapsulating join method. This is a simple example in which information of only 3 tables is joined, but this way of developing can easily be extended to more tables and more complex code.

### GROUND RULES

#### Rule 1: Make sure indexes are available and used if they will make the query run faster.

You would want to keep the number of indexes to a minimum to reduce disk space and update costs, but indexes can reduce query time a lot. In general indexes are useful for queries that retrieve a relatively small number of rows (less than 15%). Indexing small tables and indexing columns with a small number of distinct values generally does not result in performance gain.

Our PROJECT table is quite small, so we will not create an index on this table, but the DEMO table and especially the BODYWEIG table are large and contain over one hundred different studies and therefore an index is created on the column PROJECT as can be seen within SAS by using the PROC CONTENTS procedure or by looking at the indexes tab of the properties of the BODYWEIG dataset:



Using the index named PROJECT would be useful as is illustrated by the following two test runs. In these test runs the IDXWHERE and IDXNAME options are used to specify, respectively, whether or not SAS should use the indexes and which index should be used. If these options are not specified SAS will determine for you if the index is used or not. The MSGLEVEL=I option is used to show information in the log about the indexes used.

```
/*Test run 1: Do not use indexes*/  
OPTIONS MSGLEVEL = I;  
PROC SQL;  
  CREATE TABLE work.test AS  
  SELECT * FROM report.bodyweig (IDXWHERE=NO)  
  WHERE project = '123456';  
QUIT;
```

```
INFO: Data set option (IDXWHERE=NO) forced a sequential pass of the data rather than use  
of an index for where-clause processing.  
NOTE: Table WORK.TEST created, with 5672 rows and 22 columns.  
NOTE: PROCEDURE SQL used (Total process time):  
      real time          59.88 seconds  
      cpu time           0.91 seconds
```

```
/*Test run 2: Use the index PROJECT*/  
OPTIONS MSGLEVEL = I;
```

## PhUSE 2007

```
PROC SQL;
  CREATE TABLE work.test AS
  SELECT * FROM report.bodyweig (IDXNAME=project)
  WHERE project = '123456';
QUIT;

INFO: Index PROJECT selected for WHERE clause optimization.
NOTE: Table WORK.TEST created, with 5672 rows and 22 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.08 seconds
      cpu time           0.08 seconds
```

As you can see the processing time will be reduced drastically if the index is used.

In QMF you can also use the EXPLAIN command to find out if your query uses the indexes in the right way and you can get more information about how the tables are joined. A similar statement, the EXPLAIN PLAN statement, is available in Oracle and will help you investigate the query execution plan.

The EXPLAIN command cannot be used in SAS, but the undocumented options `_METHOD` and `_TREE`, which can be put directly after your PROC SQL statement, will give more information about the hierarchy of processing methods that will be chosen.

### Rule 2: Query only one table at the time.

Querying one table at the time will result in more readable code and will make step-by-step generation of your SQL code possible.

### Rule 3: Start with querying the table that restricts the number of resulting rows returned the most.

The BODYWEIG dataset will reduce the number of rows the most and will therefore be the first and only table to be queried in the next step. Once it is determined which columns are needed the first part of the total query can be created:

```
SELECT bw.project, /*Project number*/
       bw.subjno, /*Subject number*/
       bw.grouno, /*Group*/
       bw.sex, /*Gender*/
       bw.samdate, /*Date of sample*/
       bw.weight /*Body weight*/
FROM report.bodyweig (IDXNAME=project) bw
WHERE bw.project = '123456';
```

### Rule 4: Expand your query with information from the other tables by selecting the columns needed from other tables and joining them by using the JOIN statement.

The start date of treatment, which is available in the DEMO table, can be joined on the variables PROJECT and SUBJNO and therefore the query can be expanded:

```
SELECT de.start_da, /*Start date of treatment*/
       ONE.*
FROM report.demo (IDXNAME=project) de
JOIN
(
  SELECT bw.project, /*Project number*/
       bw.subjno, /*Subject number*/
       bw.grouno, /*Group*/
       bw.sex, /*Gender*/
       bw.samdate, /*Date of sample*/
       bw.weight /*Body weight*/
  FROM report.bodyweig (IDXNAME=project) bw
  WHERE bw.project = '123456'
) ONE
ON ONE.project = de.project
AND ONE.subjno = de.subjno;
```

Please note the first query is completely included in the 'JOIN (... ..)' part of the second query. The join encapsulates the entire first query. The FROM statement should always point to a table/view, the JOIN should point to the result of the previous query.

Never use a SELECT \* when selecting from a table or view. You can use the \* when referring to selecting all variables from a previous query that you have specified elsewhere in detail.

## PhUSE 2007

The last information that needs to be added is the study information from the PROJECT table. This will be done in exactly the same way (i.e. by putting a JOIN statement around this query):

```
SELECT pr.header, /*Study information*/
       TWO.*
FROM report.project pr
JOIN
  (
    SELECT de.start_da, /*Start date of treatment*/
          ONE.*
    FROM report.demo(IDXNAME=project) de
    JOIN
      (
        SELECT bw.project, /*Project number*/
              bw.subjno, /*Subject number*/
              bw.grouno, /*Group*/
              bw.sex, /*Gender*/
              bw.samdate, /*Date of sample*/
              bw.weight /*Body weight*/
        FROM report.bodyweig(IDXNAME=project) bw
        WHERE bw.project = '123456'
      ) ONE
    ON ONE.project = de.project
    AND ONE.subjno = de.subjno
  ) TWO
ON TWO.project = pr.project;
```

### Rule 5: If you have created a partial query that works as desired, do not change it.

This way of composing a query will not only make maintenance a little easier, but also makes development easier. After each step you can view the output of your query and determine whether the result is as expected. Therefore if you have added an extra JOIN and the query does not run anymore or gives incorrect results, only the outer query has to be checked for incorrect code.

### COMPARING THIS METHOD OF DEVELOPING SQL CODE WITH USING A SUB QUERY

Sub queries are usually very inefficient ways of programming. An advantage of using a sub query might be that it is more readable, but its performance will often be very bad compared to other methods of creating SQL code. If performance is a key issue, you should probably not use sub queries.

The following example illustrates the difference between the encapsulating join method and the use of sub queries. In this example it is chosen to keep only the last available bodyweight measurement for each subject.

```
/*Use of sub queries*/
PROC SQL;
CREATE TABLE work.test AS
SELECT bw.project, /*Project number*/
       bw.subjno, /*Subject number*/
       bw.grouno, /*Group*/
       bw.sex, /*Gender*/
       bw.samdate, /*Date of sample*/
       bw.weight /*Body weight*/
FROM report.bodyweig bw
WHERE bw.samdate = (SELECT MAX(bw2.samdate)
                   FROM report.bodyweig bw2
                   WHERE bw.project = bw2.project AND
                         bw.subjno = bw2.subjno AND
                         bw.grouno = bw2.grouno );
QUIT;
```

NOTE: Table WORK.TEST created, with 83544 rows and 6 columns.

NOTE: PROCEDURE SQL used (Total process time):

```
real time      7:25.53
cpu time       7:07.28
```

## PhUSE 2007

```
/*Use of encapsulating join method*/
PROC SQL;
  CREATE TABLE work.test AS
  SELECT bw.project, /*Project number*/
         bw.subjno, /*Subject number*/
         bw.grouno, /*Group*/
         bw.sex, /*Gender*/
         bw.samdate, /*Date of sample*/
         bw.weight /*Body weight*/
  FROM report.bodyweig bw
  JOIN
  (
    SELECT bw2.project,
           bw2.subjno,
           bw2.grouno,
           MAX(bw2.samdate) AS samdate
    FROM report.bodyweig bw2
    GROUP BY bw2.project,
             bw2.subjno,
             bw2.grouno
  ) ONE
  ON bw.project = ONE.project
  AND bw.subjno = ONE.subjno
  AND bw.grouno = ONE.grouno
  AND bw.samdate = ONE.samdate;
QUIT;
```

NOTE: Table WORK.TEST created, with 83544 rows and 6 columns.

NOTE: PROCEDURE SQL used (Total process time):

```
real time      1:00.04
cpu time       7.52 seconds
```

The CPU usage clearly demonstrates the benefit of joining compared to using a sub query.

### CONCLUSION

There are many ways to program a specific SQL query and each of them will have its own advantages and disadvantages. Performance of the queries is highly dependant on the way the tables are joined and whether or not indexes are defined. Furthermore the maintenance and development time of the query is highly dependant on the readability of the code and the possibility to debug your code easily.

The 'encapsulating join' method, described in this paper, describes a uniform way of programming SQL, which will lead to well performing queries which are easy to debug, because of the step-by-step development of your query.

The queries developed with this method might be longer than other queries, but, when accustomed to this method of developing SQL code, they are easy to read and therefore easy to maintain.

### ACKNOWLEDGMENTS

We would like to take the opportunity to thank Yves Poriau - OCS Consulting for contributing to our paper.

### CONTACT INFORMATION

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

Bas van Bakel  
OCS Consulting  
PO BOX 490  
5240 AL ROSMALEN  
THE NETHERLANDS  
Office: +31 (0)73 523 6000  
Fax: +31 (0)73 523 6600  
sasquestions@ocs-consulting.com  
www.ocs-consulting.com

Rick Pagie  
OCS Consulting  
PO BOX 490  
5240 AL ROSMALEN  
THE NETHERLANDS  
Office: +31 (0)73 523 6000  
Fax: +31 (0)73 523 6600  
sasquestions@ocs-consulting.com  
www.ocs-consulting.com

Brand and product names are trademarks of their respective companies.