A Note On Functions in SQL

Oracle allows the developer to define his or her own functions that can then be reference wherever an expression could be used in a SQL statement. This can be very useful, but if used incorrectly, can also be the cause of some performance nightmares that may not be easy to trace.

By: Andrew Deighton, Labyrinth IT Services Ltd

This article discusses the use of user-defined functions within SQL statements. The advantages and disadvantages of using them are examined in order to provide some guidance as to where they should be used and when they should be avoided.

Introduction

I recently had the opportunity to investigate some performance problems in a new application that was about to be put into production. The reporting facilities were not performing to an acceptable level, and as more data was added, the performance declined rapidly. After initial investigation of the problem, I was able to trace the bulk of the performance issues to the use of stored, user-defined functions within the application. Rewriting the SQL reports to avoid the function calls resulted in significant performance gains.

Why use functions?

There are a number of reasons why developers would choose to use user-defined functions within an application. These include:

- The ability to encapsulate business rules in a single place – the function library – and so aid maintainability of the application,
- Being able to increase SQL functionality by extending the library of built-in functions,
- To manipulate data (for example, simulate new data types such as co-ordinate systems),

- To improve performance (using functions in the WHERE clause allows more data to be filtered at the server side, resulting in less network traffic).

Another reason for using functions, and one that is less easy to justify, is that a lot of SQL code is written by the developers of the front-end applications – usually JAVA, C/C++ or Visual Basic programmers. While I have nothing against such programmers, the philosophies behind coding for a procedural language and coding SQL are significantly different, and if you are thinking procedurally, functions will often be the obvious, but not necessarily the most effective, way to work.

An Illustration

Consider the situation in which a market research company has carried out a survey, and stored the results in an Oracle table. The table could be defined as:

```
SURVEY = {RESPONDENT_ID, QUESTION_NUMBER, RANK}.
```

In order to work out the average rank for a question, a function could be used:

```
FUNCTION My_Average (question number) return number is
V_average number;
BEGIN
  SELECT (Sum(Rank)-Max(Rank)-Min(Rank))/(Count(Rank)-2)
  INTO V_average
  FROM survey
  WHERE Question_number = question;
  RETURN v_average;
  EXCEPTION
    WHEN OTHERS THEN
      RETURN -1;
END;
```

The formula can then be called within the SQL code of a report as in:

```
SELECT
  Question,
  Max(Rank) Best_score,
  Min(Rank) Worst_score,
  My_Average(Question) Average_score
FROM
  Survey
GROUP BY
  Question;
```

This returns, for each question in the survey, the maximum and minimum ranks, as well as the weighted average as calculated by the My_Average function.

If the requirements of the application change, and a different kind of average is required, the function can be changed, and any reports or applications that reference it automatically use the new definition, with no need to change or re-compile any application programs.

The Problem

Showing an "Explain Plan" for this query shows a full table scan through the SURVEY table, which one would expect when calculating values over the entire table. However, an Explain Plan does not show that for every row in the table that is returned, the function MY_AVERAGE is called, and it in turn carries out a full table scan. In effect we therefore have a nested loop full of full table scans.

The fact that the execution plan for the statement does not show the plan for the functions that are included can make performance problems harder to find than they would otherwise be. Looking at the execution plan for this statement shows an optimal execution plan that cannot be improved.

Running this query on a test table (Oracle8.1.6, Linux, 10000000 rows) took approximately 47 minutes. The execution time was reduced to around one minute by removing the My_Average function call and rewording the query with the equivalent statement:

```
SELECT
  Question,
  Max(Rank) Best_score,
  Min(Rank) Worst_score,
  (Sum(Rank)-Max(Rank)-Min(Rank))/(Count(Rank)-2)Average_score
FROM
  Survey
GROUP BY
  Question;
```

This produces exactly the same results, while significantly improving the performance of the query (giving over 97% reduction in execution time). Noteworthy, too, is the fact that Oracle reports exactly the same execution plan (and the same cost) for this statement as for the earlier statement.

The cost of implementing this would be that if the My_Average function were called from a large number of places within the application, then the change would need to be made in each place, possibly requiring application programs to be re-compiled.

A different example

Consider an application that uses a simple co-ordinate system. Points in this system could be stored in the following table:

```
POINT = {POINT_ID, X_COORD, Y_COORD}
```

It is then possible to define a function that calculates the distance between any two points by using a simple formula.

```
FUNCTION distance (X1 number, Y1 number, X2 number, Y2 number) RETURN NUMBER IS
BEGIN
   RETURN sqrt ( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );
   EXCEPTION
     WHEN OTHERS THEN
       Null;
END distance;
```

This simple function can then be called from SQL as in:

```
select point_id, x_coord, y_coord, distance (10,15,x_coord, y_coord)
from point
where distance (10,15,x_coord, y_coord) < 100
order by 4
```

This provides a simple mechanism for extracting the relevant points, while not imposing the large performance overheads that were associated with the previous example, and the function can be called from any part of the application without having to re-code the formula. The application could then easily be changed to work on a different co-ordinate system, for example, using the surface of a sphere rather than a flat surface, and no changes would be required within the SQL above, only the relevant functions would need updating.

This is only a very simple example of where a function can be used to increase the functionality of SQL, but serves as a useful demonstration.

Conclusion

It is very useful to have the ability to define your own functions and then to reference them within your SQL code. This allows you to do things that would have been very difficult or even impossible to do without them. However, they should not be used without some consideration for the impact that they can potentially have on performance. This is especially important given that the Oracle optimizer ignores the contents of the function when evaluating its execution path. Functions that select data from the database are particularly vulnerable to performance hits.

Andrew Deighton is a database consultant and director at Labyrinth IT Services. He has been working with Oracle for more than 10 years and has published several papers on various topics related to Oracle. He also works as an instructor for Learning Tree International. He can be contacted at andrew@labyrinth-it.co.uk.