

Tips for Writing Efficient SQL Queries

**From:
Vigyan Kaushik**

In order to improve overall application performance, it's very important to construct SQL queries in the most efficient way. There can be many different ways to write a SQL query. Here are few tips that can help you in writing efficient and reusable SQL queries. All examples given below are based on Oracle default demo tables EMP and DEPT. You can create these tables in your local schema from the following directory in windows environment.

```
%ORACLE_HOME%\sqlplus\demo\demobld.sql
```

Order of the tables in Joins: If you specify 2 or more tables in the FROM clause of a SELECT statement, then Oracle parser will process the tables from right to left, so the table name you specify last will be processed first. In this case you have to choose one table as driving table. Always choose the table with less number of records as the driving table.

Name the Columns in a Query: There are three good reasons why it is better to name the columns in a query rather than to use *"select * from ..."*.

1. Network traffic is reduced. This can have a significant impact on performance if the table has a large number of columns, or the table has a long or long raw column (both of which can be up to 2 GB in length). These types of columns will take a long time to transfer over the network and so they should not be fetched from the database unless they are specifically required.

2. The code is easier to understand.

3. It could save the need for changes in the future. If any columns is added to or removed from the base table/view, then "select *" statement can produce wrong results set and statement may fail.

Use table alias: Always use table alias and prefix all column names with the aliases when you are using more than one table.

Never compare NULL to anything else: All expressions return NULL if one of the operands is NULL. This is applicable for all operators except Concatenation operator (||).

Use Bind Variables: It is also better to use bind variables in queries. That way the query becomes generic and therefore re-usable. For example, instead of writing a query like -

```
SELECT ename, sal
FROM emp
WHERE deptno = 20;
```

Change it to -

```
SELECT ename, sal
FROM emp
WHERE deptno = :deptno;
```

The first query can be re-used for deptno number 20 only, whereas the second query can be re-used for any other deptno also.

SQL Writing Convention: It is a good practice to use a standard syntax for wiring SQL queries. I will recommend following standards to use while writing SQL queries.

Write all standard SQL TEXT in upper case:

For example:

```
SELECT ename, sal  
FROM emp  
WHERE deptno = 20;
```

Write all non standard SQL TEXT (Table name, Column name etc) in lower case:

For example:

```
SELECT ename, sal  
FROM emp  
WHERE deptno = 20;
```

Formatter Plus in Toad can be use to format SQL statements in this format. Select the complete SQL statement and right click on the “Format Code” menu.

Note: It is important to write similar SQL statement in same case.

For example: Oracle will reparse following queries as they are not written in the same case

```
Select * from EMP;
```

```
Select * from emp;
```

Use EXISTS instead of DISTINCT: Use EXISTS in place of DISTINCT if you want the result set to contain distinct values while joining tables.

For example:

```
SELECT DISTINCT d.deptno, d.dname  
FROM dept d, emp e  
WHERE d.deptno = e.deptno;
```

The following SQL statement is a better alternative.

```
SELECT d.deptno, d.dname  
FROM dept d  
WHERE EXISTS (SELECT e.deptno  
FROM emp e  
WHERE d.deptno = e.deptno);
```

Use of expressions and indexes: The optimizer fully evaluates expressions whenever possible and translates certain syntactic constructs into equivalent constructs. This is done either because Oracle can more quickly evaluate the resulting expression than the original

expression or because the original expression is merely a syntactic equivalent of the resulting expression.

Any computation of constants is performed only once when the statement is optimized rather than each time the statement is executed. Consider these conditions that test for salaries greater than \$2000.

```
sal > 24000/12  
sal > 2000  
sal*12 > 24000
```

If a SQL statement contains the first condition, the optimizer simplifies it into the second condition.

Please note that optimizer does not simplify expressions across comparison operators. The optimizer does not simplify the third expression into the second. For this reason, we should write conditions that compare columns with constants whenever possible, rather than conditions with expressions involving columns.

The Optimizer does not use index for the following statement:

```
SELECT *  
FROM emp  
WHERE sal*12 > 24000 ;
```

Instead of this use the following statement:

```
SELECT *  
FROM emp  
WHERE sal > 24000/12 ;
```

Use of NOT operator on indexed columns: Never use NOT operator on an indexed column. Whenever Oracle encounters a NOT on an index column, it will perform full-table scan.

For Example:

```
SELECT *  
FROM emp  
WHERE NOT deptno = 0;
```

Instead use the following:

```
SELECT *  
FROM emp  
WHERE deptno > 0;
```

Function or Calculation on indexed columns: Never use a function or calculation on an indexed column. If there is any function is used on an index column, optimizer will not use index.

For Example:

Do not use until need exactly match string:

```
SELECT *
FROM emp
WHERE SUBSTR (ename, 1, 3) = 'MIL';
```

Use following instead:

```
SELECT *
FROM emp
WHERE ename LIKE 'MIL%';
```

Do not use the following as || is the concatenate function. Like other functions and it disables index.

```
SELECT *
FROM emp
WHERE ename || job = 'MILLERCLERK';
```

Use the following instead

```
SELECT *
FROM emp
WHERE ename = 'MILLER' AND job = 'CLERK';
```

Avoid Transformed Columns in the WHERE Clause : Use untransformed column values.

For example, use:

```
WHERE a.order_no = b.order_no
```

Rather than

```
WHERE TO_NUMBER (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
= TO_NUMBER (SUBSTR(b.order_no, INSTR(a.order_no, '.') - 1))
```

Combine Multiples Scans with CASE Statements: Often, it is necessary to calculate different aggregates on various sets of tables. Usually, this is done with multiple scans on the table, but it is easy to calculate all the aggregates with one single scan. Eliminating n-1 scans can greatly improve performance.

Combining multiple scans into one scan can be done by moving the WHERE condition of each scan into a CASE statement, which filters the data for the aggregation. For each aggregation, there could be another column that retrieves the data.

The following example has count of all employees who earn less then 2000, between 2000 and 4000, and more than 4000 each month. This can be done with three separate queries.

```
SELECT COUNT (*)
```

```
FROM emp  
WHERE sal < 2000;
```

```
SELECT COUNT (*)  
FROM emp  
WHERE sal BETWEEN 2000 AND 4000;
```

```
SELECT COUNT (*)  
FROM emp  
WHERE sal > 4000;
```

However, it is more efficient to run the entire query in a single statement. Each number is calculated as one column. The count uses a filter with the CASE statement to count only the rows where the condition is valid. For example:

```
SELECT COUNT (CASE WHEN sal < 2000  
                THEN 1 ELSE null END) count1,  
       COUNT (CASE WHEN sal BETWEEN 2001 AND 4000  
                THEN 1 ELSE null END) count2,  
       COUNT (CASE WHEN sal > 4000  
                THEN 1 ELSE null END) count3  
FROM emp;
```

Please feel free to write your questions/comments at vkaushik@dbapool.com