Exploring Best Practices and Guidelines for Tuning SQL Statements

Ami Aharonovich
ilOUG, DBAces

Keywords:

SQL Tuning
Best Practices
SQL Performance
Execution Plans
Optimizer
Indexes
New Features

Introduction

This session will focus on providing important guidelines, recommendations and best practices for writing efficient and optimized SQL statements while ensuring high level of performance and response time.

The session will include real life examples and live demonstrations and will be followed by various SQL scripts to demonstrate the implementation of guidelines and techniques for implementing SQL statements best practices, including restructuring SQL statements and rewriting complex sub-queries, using EXISTS and WITH clause, using scalar sub-queries, implementing SQL query result cache, dealing with data type conversions and more.

Parsing Time is Important

Here I would like to present a simple demo, where we insert 100,000 new rows into a table, using two different anonymous blocks. The first one will use dynamic SQL (with EXECUTE IMMEDIATE) without using a bind variable while the second one will basically do exactly the same just with a small and significant difference – it will use a bind variable. Here is the code to be executed:

```
BEGIN
FOR i IN 1..100000 LOOP
EXECUTE IMMEDIATE 'INSERT INTO t (x,y) VALUES ('||i||','A')';
END LOOP;
END;
```

The above version is without the usage of bind variable and it will take about 40 seconds to complete.

```
BEGIN
FOR i IN 1..100000 LOOP
EXECUTE IMMEDIATE 'INSERT INTO t (x,y) VALUES (:i,''A'')' USING i;
END LOOP;
END;
```

The above version is with the usage of bind variable and it will take about 8 seconds to complete.

Now, we will query the data dictionary views to get the amount of memory which is allocated for each one of the insert statements. We will see the number of cursors that are still kept in cache and the total amount of memory allocated for those statements. Here is the query that will present the above:

```
SELECT SUBSTR(sql_text,11,8) "Bind", COUNT(*),
       ROUND(SUM(sharable_mem)/1024) "Memory KB"
FROM   v$sqlarea
WHERE  sql_text LIKE 'INSERT%INTO t (x,y)%'
GROUP BY SUBSTR(sql_text,11,8);
```

The above query will show a significant difference between the two anonymous blocks. The first one, without the bind variable, will have thousands of versions kept in the memory consuming a few hundreds of MB from the total amount of memory allocated for the shared pool. The second one, using the bind variable, will have a single version kept in memory and it will allocate only 14 KB of memory. That is a huge difference and of course a significant benefit for the usage of binds variables.

We will also discuss the bind variable peeking feature, the fact that the Oracle optimizer peeks at the binds value in order to determine how to optimize the query, when the query is first hard parsed. In addition, we will include also a short discussion on the 11g new feature called Adaptive Cursor Sharing, which provide an intelligent cursor sharing mechanism for statements that use bind variables and allows the optimizer to generate a set of plans that are optimal for different sets of bind values.

Inefficient SQL Statements

In this part we will discuss a few examples and present some common mistakes for writing inefficient SQL statements. The examples will include:

1) Writing conditions with mixed mode expressions, which will cause an implicit data type conversion and force a full table scan even if there is a possible index scan available

2) Using functions on the indexed columns, which will cause a full table scan instead of using the corresponding column index

3) Using join condition with function, which will also cause a full table scan join instead of using the relevant index access paths

4) Using UNION instead of UNION ALL

Restructuring SQL Statements

Here we will discuss a few alternatives for restructuring SQL statements while achieving better access paths and more optimal execution plans to provide overall better query performance. We will discuss the following options:

1) Using the EXISTS operator – instead of using a join or the IN operator, the EXISTS operator is a Boolean operator and can be used to provide a "Semi Join" instead of doing a regular/full join

2) Using in-line views – we will compare two queries, one using a correlated Subquery and one using an in-line view. We will discuss the differences between the queries and will use the SQL Trace feature to trace database activities for each query and then compare the results by using the TKPROF interpreter.

3) Using the WITH clause – we will see how the WITH clause can be used to benefit from making the query more readable and more significantly improving query performance as the WITH clause is evaluated only once, while the clause can appear multiple times in the query. Once again, we will compare between a correlated Subquery to the in-line view and this time will also add the WITH clause, using SQL Trace and TKPROF and we will discuss the differences between all three options.

4) Scalar Subqueries – we will discuss the usage of Scalar Subqueries, where and when we should consider changing our query to use this feature and how we can greatly benefit from the fact that the Oracle database caches the results of a Scalar Subquery and reuses the value even though the query should be called more than once. We will once again use the SQL Trace feature and TKPROF and investigate two queries, one using a regular function call and one using a Scalar Subquery and we will discuss the significant performance differences between the two queries.

Oracle 11g SQL Query Result Cache

Starting with Oracle database 11g, the database can now also cache SQL result sets!
If you have a query that is executed over and over again against slowly or never-changing data, you will find this new SQL Query Result Cache feature to be of great interest. This feature includes a new dedicated memory buffer stored in the shared pool which is used for storing and retrieving the cached results. Query results will become invalid automatically, when data in the objects being access by the query is modified. Multiple users can see these results without having to repeat the same query.

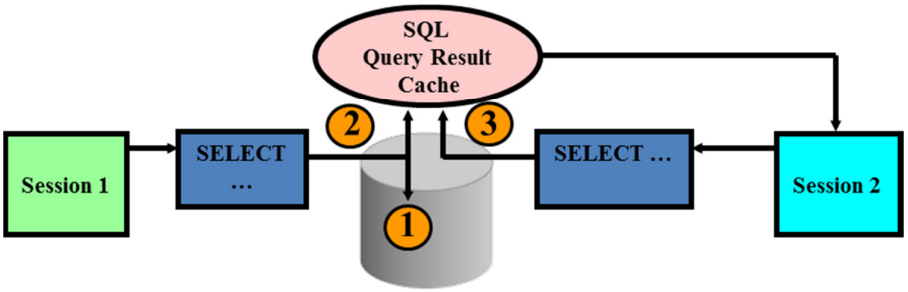

Illustration 1: 11g SQL Query Result Cache

Oracle 11g PL/SQL Function Result Cache

In the past, if you called a PL/SQL function 1,000 times and each function call consumed 1 second, the 1,000 calls would take 1,000 seconds. With this 11g new Function Results Cache feature, depending on the inputs to the function and whether the data underlying the function changes, 1,000 function calls could take about 1 second, total. The PL/SQL Function Result Cache allows storing the results of PL/SQL functions in the SGA (in the shared pool). This new feature provides the ability to mark a PL/SQL function to indicate that its result should be cached to allow lookup, rather than recalculation, when the same parameter values are called. This is done transparently using the input parameters as the lookup key and it is instance wide – meaning that all distinct sessions invoking the function can benefit from this new feature.

Indexes Guidelines

In this part we will present a few important guidelines regarding the usage of indexes:

1) Indexes and nulls – we will present an example case when a query is checking for existence of nulls in an indexed column and will evaluate the query execution plan and why the index cannot be used to retrieve all the nulls. In addition, we will see how we can overcome this behavior by changing the index structure and recreating the index in such a way that it will inclue also the nulls and will provide an index scan for the above query.

2) Nulls, Cardinality and Indexes – here we will discuss an interesting scenario where we use "invalid" values instead of the nulls, in order to have the values stored in the index. The example will present a date column which will store future dates instead of the nulls, and the influence of having those future values on column statistics, and as a result, on optimizer decisions and the execution plan that will be generated by the optimizer.

3) Oracle 11g Invisible Index – this new feature causes the optimizer to ignore the index and it is an alternative to making it unusable or dropping it. By using invisible index you can test the removal of an index before dropping it and also use temporary index structures for certain operations without affecting the overall application. Invisible indexes are stil being maintained during the execution of DML statements.

Oracle Database 12c New Features Overview

Towards the ending of this presentation, we will briefly describe a few key new features presented in Oracle Database 12c, including:

1) Using invisible columns – describing the usage of this feature and describing an example showing how this feature can be used for changing the order of columns in an existing table structure

2) Invoking PL/SQL from SQL – describing the usage of WITH clause with a PL/SQL function and how we can invoke this PL/SQL procedural code from within the SQL statement

3) Using new improved default values – describing how we can use new defaults for sequences, defaults when nulls are inserted and the new identity type

4) Enhanced statistics capabilities – describing new enhanced statistics capabilities for gathering statistics during loads, and gathering session private statistics for global temporary tables

General Tips, Guidelines and Closure

For the closure of this presentation, we will briefly discuss a few general tips and guidelines for monitoring and tuning your SQL statements. This discussion will include basic guidelines for monitoring and tuning SQL statements which are identified as heavy consumers of resources, setting goals for tuning your queries and keep tracing your running queries and track degradation in response time. In addition, we will briefly discuss a few important guidelines for writing and restructuring SQL statements, including minimizing parsing processes, using bind variables, avoiding mixed mode expressions and implicit data type conversions, using EXISTS instead of joins and the IN operator, using the WITH clause, checking for correct selectivity and cardinality estimations, finding problematic access paths such as "Merge Join Cartesian", the significant of having constraints and breaking up complicated SQL statements to avoid problematic and heavy joins with numerous tables.

This presentation will include real life examples and live demonstrations using Oracle Database 11g and 12c with SQL*Plus and SQL Developer. Slides will also include some references for additional readings and recommended articles for essential SQL tuning tips and techniques, improving performance through changes, tuning by tracing, dynamic sampling, adaptive cursors, SQL access advisor, SQL performance analyser, caching and pooling, constraints, metadata and more.

Contact address:

Ami Aharonovich
ilOUG, DBAces
9 Shaul Hameleh St.
55654, Kiryat Ono

Phone: +972-524-377737
Fax: +972-77-3373770
Email Ami@DBAces.co.il
Internet: www.DBAces.co.il