## TUESDAY, 17 FEBRUARY 2009

### Oracle SQL Parsing Essentials

Any developer working with Oracle must understand how SQL statements are parsed if they are to build high performance, scalable applications. It is not an exaggeration to say that a few badly written queries can have a devastating effect on the overall performance of the database. This is not just limited to queries that retrieve data inefficiently but also applies to SQL statements that are repeatedly parsed unnecessarily.

When an SQL statement is submitted to Oracle, it will do one of three things; a hard parse, a soft parse or no parse at all.

## HARD PARSE

The first time an SQL statement is executed, Oracle must do a hard parse. It must perform all of the following steps.

   1. Parse the statement.
   2. Determine the optimal execution plan.
   3. Execute the statement.

The main distinction between a hard and a soft parse is that a hard parse must do step two – query optimization.

During optimization, the cost based optimizer (CBO) attempts to determine the best way to access the data. The CBO uses complex algorithms based on system and table statistics to generate many (possibly thousands) of execution plans, each of which is assigned a numeric cost. The plan with the lowest cost is selected for use.

It is important to understand that optimization is a very expensive operation. It is a resource hungry, CPU intensive process that can take longer to run than it does to retrieve the data.

We can see an example of this with a TKPROF report on a simple SQL command.

select \*
from
 all\_objects where object\_id = 115

call count cpu elapsed disk query current rows
------- ------ -------- ---------- ------- --------- ---------- -------
Parse 1 0.14 0.13 0 0 0 0
Execute 1 0.00 0.00 0 0 0 0
Fetch 1 0.00 0.00 0 12 0 1
------- ------ -------- ---------- ------- --------- ---------- -------
total 3 0.14 0.13 0 12 0 1

Misses in library cache during parse: 1
Optimizer mode: ALL\_ROWS

The parsing operation took over a tenth of a second, whereas the time to fetch the data didn't even register because it was so quick.

But there is another issue. During optimization, Oracle will lock areas of memory using a device called a latch. Latches protect shared memory from being updated by multiple sessions at the same time. The longer the latches are held, the longer other sessions are queued, waiting for their turn to access the memory segment. This will ultimately inhibit scalability.

## SOFT PARSE

Hard parses should be avoided whenever possible if we are to provide a scalable solution. Fortunately, once the difficult task of producing the execution plan has been done, Oracle will automatically cache it in a memory area called the shared pool. Next time an identical query is executed, Oracle will retrieve the execution plan produced by the original query and re-use it. This is known as a soft parse.

Now Oracle only needs to do the following.

   1. Parse the statement.
   2. Execute the statement.

If we execute the same SQL statement that we ran previously, the TKPROF output looks like this.

select \*
from
 all\_objects where object\_id = 115

call count cpu elapsed disk query current rows
------- ------ -------- ---------- ------- -------- ---------- --------
Parse 1 0.00 0.00 0 0 0 0
Execute 1 0.00 0.00 0 0 0 0
Fetch 1 0.00 0.00 0 12 0 1
------- ------ -------- ---------- ------- -------- ---------- --------
total 3 0.00 0.00 0 12 0 1

Misses in library cache during parse: 0
Optimizer mode: ALL\_ROWS

This time we did a soft parse. We re-used the execution plan which was produced the first time we ran the query and no measurable time was spent parsing.

So Oracle takes care of caching the execution plan automatically for us. However, care must be taken to ensure that the appropriate SQL statement can be found in the cache. Consider the following Java method which returns an account name from an account id.

public String getAccountName(Connection con,
 int accId) throws SQLException {
 String accName = null;

 Statement stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery(
 "SELECT acc\_name FROM acc WHERE acc\_id = " + accId);

 if (rs.next()) {
 accName = rs.getString("acc\_name");
 }
 stmt.close();

 return accName;
}

Earlier I stated that Oracle will do a soft parse if an identical query has been executed before. The problem here is that the SQL produced by this method is unlikely to be identical from one invocation to the next. For example, if we enquire on accounts 100, 200 and 500 we get the following SQL.

SELECT acc\_name FROM acc WHERE acc\_id = 100;
SELECT acc\_name FROM acc WHERE acc\_id = 200;
SELECT acc\_name FROM acc WHERE acc\_id = 500;

Only the literal values are different in each case but, as far as Oracle is concerned, these are three separate statements which means three hard parses. We need Oracle to perform a single hard parse for the first statement and soft parses for subsequent executions.

## BIND VARIABLES

The solution is to use bind variables with a PreparedStatement.

public String getAccountName(Connection con,
 int accId) throws SQLException {
 String accName = null;

 PreparedStatement pstmt = con.prepareStatement(
 "SELECT acc\_name FROM acc WHERE acc\_id = :acc\_id");
 pstmt.setInt(1, accId);
 ResultSet rs = pstmt.executeQuery();

 if (rs.next()) {
 accName = rs.getString("acc\_name");
 }
 pstmt.close();

 return accName;
}

Now every invocation of this method will produce the same SQL statement.

SELECT acc\_name FROM acc WHERE acc\_id = :acc\_id

Oracle will do a hard parse the first time the query is submitted. This is unavoidable. However, subsequent executions will read the cached execution plan from the shared pool, substitute the bind variable values and execute the query. The SQL is soft parsed and the query optimization stage is avoided.

If you wish to produce a well-behaved, high performance, scalable application then the importance of bind variables cannot be understated. It is simply one of the most important concepts a Java programmer must understand when working with Oracle.

Later we will see the dramatic difference in performance between hard and soft parsing.

## A WORD ABOUT CURSOR\_SHARING

Oracle provides a feature which enables auto-binding on SQL statements. Setting theCURSOR\_SHARING initialization parameter to FORCE, instructs Oracle to dynamically replace literals in an SQL statement with bind variables. The original literal values are then substituted, just as if bind variables had been used in the client program. All this happens transparently in the database itself.

For example

SELECT acc\_name FROM acc WHERE acc\_id = 100;

becomes

SELECT acc\_name FROM acc WHERE acc\_id = :SYS\_B\_0;

This may seem like a silver bullet solution but unfortunately results can be unpredictable and side effects may even degrade the performance of certain queries.

This feature should only ever be considered in an emergency, when changing the application to use bind variables is not an option. It is no substitute for carefully written and well tested SQL that uses PreparedStatements and bind variables from the outset.

## STATEMENT PARSING

So soft parsing improves performance and scalability. But what does a soft parse actually involve? Broadly speaking, the parsing stage consists of syntax checking, semantic analysis and a lookup in the shared pool. Let's look at each of these in turn.

## SYNTAX CHECKING

Is it valid SQL as defined by the Oracle SQL reference documentation?

SELECT acc\_name RFOM acc;

SQL Error: ORA-00923: FROM keyword not found where expected

The FROM keyword has been misspelt.

## SEMANTIC ANALYSIS

Is the syntactically correct SQL statement valid in terms of the objects being accessed, permissions and column ambiguities etc?

SELECT acc\_name FROM cust.acc;

SQL Error: ORA-00942: table or view does not exist

Either ACC does not exist in the CUST schema or the user does not have the necessary privileges to access it.

## SHARED POOL LOOKUP

At this point we have a valid SQL statement and Oracle will attempt to find an existing execution plan in the shared pool.

However, it is not sufficient to just locate a statement in the pool with identical text. An existing statement may refer to different objects, in different schemas, or the environment in which it was executed may be incompatible, making the execution plan inappropriate for our query. Oracle must resolve these issues which takes time.

The point I am trying to make here is that, while a soft parse is far more efficient than a hard parse, it is not without cost. So what is even more efficient than doing a soft parse? Doing no parse at all.

## NO PARSE

The problem with the getAccountName method shown earlier is that, each time this method is called, the SQL statement is compiled (when prepareStatement is executed) and each time the statement is compiled, a soft parse is performed. If, however, the PreparedStatement was created outside of the method and then passed in as a parameter, the statement could be parsed once and executed many times.

public void testGetAccountName(Connection con) throws SQLException {

 PreparedStatement pstmt = con.prepareStatement(
 "SELECT acc\_name FROM acc WHERE acc\_id = :acc\_id");

 for (int i=1; i <= 10; i++) {
 String accName = getAccountName(pstmt, i);
 System.out.println("Account Name: " + accName);
 }

 pstmt.close();
}

public String getAccountName(PreparedStatement pstmt,
 int accId) throws SQLException {
 String accName = null;

 pstmt.setInt(1, accId);
 ResultSet rs = pstmt.executeQuery();

 if (rs.next()) {
 accName = rs.getString("acc\_name");
 }

 return accName;
}

The PreparedStatement effectively maintains a direct reference to the memory structure which represents our statement. The statement parsing and the shared pool lookup process performed by the soft parse is no longer necessary. All Oracle needs to do is bind the new values to the existing structure and execute the query.

## STATEMENT CACHING

Parsing an SQL statement once and executing it many times should be our goal, but passing around a PreparedStatement is not always convenient or even possible. Code can become cluttered if every method in the call stack is forced to include the PreparedStatement object.

One solution is to use implicit statement caching. Implicit statement caching improves performance by caching frequently used executable statements in the JDBC driver. This allows us to keep our original getAccountName implementation which creates and closes thePreparedStatement within the method.

By setting the following properties on the data source, we allow up to 200 statements to be cached for each connection to the database.

public Connection getConnection() throws SQLException {

 OracleDataSource ods = new OracleDataSource();
 ods.setURL("jdbc:oracle:thin:@localhost:1521:test01");

 ods.setUser("SCOTT");
 ods.setPassword("TIGER");

 Properties prop= new Properties();
 prop.setProperty("MaxStatementsLimit","200");

 ods.setConnectionCachingEnabled(true);
 ods.setConnectionCacheProperties (prop);

 Connection con = (OracleConnection)ods.getConnection();

 con.setAutoCommit(false);

 return con;
}

The JDBC driver will automatically store prepared and callable statements in the cache each time the statement is closed. If the same SQL statement is executed again, it will be retrieved from the cache and the statement can be executed without any parsing at all.

## A WORD ABOUT PL/SQL

PL/SQL makes use of bind variables and statement caching automatically.

Let's rewrite the getAccountName method we saw earlier as a PL/SQL function.

CREATE OR REPLACE FUNCTION get\_account\_name(p\_acc\_id acc.acc\_id%TYPE)
RETURN VARCHAR2 AS
 l\_acc\_name acc.acc\_name%TYPE;
BEGIN

 SELECT acc\_name
 INTO l\_acc\_name
 FROM acc
 WHERE acc\_id = p\_acc\_id;

 RETURN l\_acc\_name;

EXCEPTION
 WHEN NO\_DATA\_FOUND THEN
 RETURN null;

END;

In this example, p\_acc\_id is substituted with a bind variable and the select statement is transparently cached by the PL/SQL engine. We can see this by running the function one thousand times then looking at the statement in the V$SQL view.

DECLARE
 l\_name acc.acc\_name%TYPE;
BEGIN

 FOR i IN 1 .. 1000
 LOOP
 l\_name := get\_account\_name(i);
 END LOOP;

END;
/

anonymous block completed

SELECT sql\_text, parse\_calls, executions
FROM v$sql
WHERE lower(sql\_text) LIKE 'select acc\_name from acc%';

SQL\_TEXT PARSE\_CALLS EXECUTIONS
--------------------------------------------- ------------- ------------
SELECT ACC\_NAME FROM ACC WHERE ACC\_ID = :B1 1 1000

1 rows selected

Here we see that bind variable B1 has been substituted into the query which means that, even though the statement was executed one thousand times, it was parsed only once.

In fact it is more difficult to make PL/SQL *not* use bind variables. To do so we would need to re-code the function to use the EXECUTE IMMEDIATE statement as follows.

CREATE OR REPLACE FUNCTION get\_account\_name(p\_acc\_id acc.acc\_id%TYPE) RETURN VARCHAR2 AS
 l\_acc\_name acc.acc\_name%TYPE;
BEGIN

 EXECUTE IMMEDIATE
 'SELECT acc\_name FROM acc WHERE acc\_id = ' || p\_acc\_id
 INTO l\_acc\_name;

 RETURN l\_acc\_name;

EXCEPTION
 WHEN NO\_DATA\_FOUND THEN
 RETURN null;
END;
/

DECLARE
 l\_name acc.acc\_name%TYPE;
BEGIN

 FOR i IN 1 .. 1000
 LOOP
 l\_name := get\_account\_name(i);
 END LOOP;

END;
/

anonymous block completed

We can see the effect of this change in the V$SQL view.

SELECT sql\_text, parse\_calls, executions
FROM v$sql
WHERE lower(sql\_text) LIKE 'select acc\_name from acc%';

SQL\_TEXT PARSE\_CALLS EXECUTIONS
-------------------------------------------- ------------- ------------
SELECT acc\_name FROM acc WHERE acc\_id = 241 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 8 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 491 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 994 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 55 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 224 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 94 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 376 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 977 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 38 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 814 1 1
SELECT acc\_name FROM acc WHERE acc\_id = 17 1 1
...

1000 rows selected.

## EXAMPLE

To illustrate the performance implications of SQL parsing, we will execute a simple query many times from Java, using three different techniques. The slight differences in the Java code will demonstrate the effects of hard parsing, soft parsing and no parsing in the database. We will monitor the results using Oracle Enterprise Manager.

## HARD PARSE

First, we execute the method we saw earlier, which uses a JDBC Statement rather than aPreparedStatement. This will trigger a hard parse each time the SQL is submitted.

public String getAccountName(Connection con,
 int accId) throws SQLException {
 String accName = null;

 Statement stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery(
 "SELECT acc\_name FROM acc WHERE acc\_id = " + accId);

 if (rs.next()) {
 accName = rs.getString("acc\_name");
 }
 stmt.close();

 return accName;
}

The method is executed a total of one million times in ten concurrent threads and takes about twelve minutes to complete.

## ADDM PERFORMANCE ANALYSIS

The first indication that all is not well can be seen on the Database Instance home page under the ADDM Performance Analysis region.


Clicking on 'Hard Parse Due to Literal Usage' takes us to the following page.

Click anywhere on the image to see full size.

Oracle has recognized that we have many SQL statements which vary only by the literal value we have specified in our WHERE clause. Oracle recommends that we set the CURSOR\_SHARINGparameter to FORCE or that we change our application logic to use bind variables instead of literals. Later, we will follow the second recommendation and change our application logic to use a PreparedStatement rather than a Statement.

Also, if we navigate to the Duplicate SQL page, a feature designed to help us recognise hard parse issues such as this, then we see the following.


Further evidence that we have a significant performance problem can be seen on the graphs in the Performance section of Enterprise Manager.


Clearly, we are having some concurrency issues. If we click on the Concurrency link we see that library latching is to blame. This is a classic symptom of excessive hard parsing.



## SOFT PARSE

So let's rewrite the method to use bind variables and a PreparedStatement to run the same query.

public String getAccountName(Connection con,
 int accId) throws SQLException {
 String accName = null;

 PreparedStatement pstmt = con.prepareStatement(
 "SELECT acc\_name FROM acc WHERE acc\_id = :acc\_id");
 pstmt.setInt(1, accId);
 ResultSet rs = pstmt.executeQuery();

 if (rs.next()) {
 accName = rs.getString("acc\_name");
 }
 pstmt.close();

 return accName;
}

Once again, the method is executed one million times in ten concurrent threads but this time it takes only four minutes.

And how did the system behave during the query? Lets take a look at the Average Active Sessions on the Performance page.


Note the difference in scale of the Active Sessions on the left compared to the hard parse version. We are using less CPU, waiting less time for resources and we have reduced the library latch contention considerably. The benefits of soft parsing are clear.

However, if we take a look at the SQL Details page for our query we see evidence of another potential performance bottleneck.


The value that catches the eye is Total Parses in the Shared Cursor Statistics region. It is about the same as the number of executions. The fact that we did only two hard parses is good but the fact we did a million soft parses suggests we could do better.

## NO PARSE

Let's enable implicit statement caching when we retrieve our database connection and re-run the previous test.

public Connection getConnection() throws SQLException {

 OracleDataSource ods = new OracleDataSource();
 ods.setURL("jdbc:oracle:thin:@localhost:1521:test01");

 ods.setUser("SCOTT");
 ods.setPassword("TIGER");

 Properties prop= new Properties();
 prop.setProperty("MaxStatementsLimit","200");

 ods.setConnectionCachingEnabled(true);
 ods.setConnectionCacheProperties (prop);

 Connection con = (OracleConnection)ods.getConnection();

 con.setAutoCommit(false);

 return con;
}

This time it took just two minutes to run the test and the overhead on the database server was reduced considerably.


If we take a look at the SQL Details page we see that, although our query was executed over a million times, it did just one hard parse and ten soft parses (one for each thread in our test).


If we compare the activity statistics from the previous two tests, the benefits of reduced parsing becomes apparent.

## SOFT PARSE



## NO PARSE


The execution time is halved and the wait time reduced considerably when we keep the number of soft parses to a minimum.

## SUMMARY

**Parse once, execute many times**.

This should be the goal of any developer who intends to build high performance, scalable applications using the Oracle database. We must endeavour to keep, not only hard parsing to a minimum but soft parsing also.

In Java, the use of PreparedStatements and bind variables is essential to avoid unnecessary hard parses. This is one of the most valuable lessons, regarding Oracle database performance, that a Java developer can learn.

However, it is also important to avoid unnecessary soft parses. Effective re-use of aPreparedStatement and implicit statement caching are both useful techniques to avoid excessive soft parses.

Fortunately, PL/SQL avoids these problems by using bind variables and statement caching automatically.