

Performance Problems Related to the Reparsing of SQL Statements

Jeff Maresh, Engineering Solutions, Inc.

Introduction

To maintain acceptable performance on Oracle databases, periodic tuning is necessary. Most database tuning problems are straightforward to diagnose with a variety of tools such as Oracle bstat/estat, Performance Pack, and a multitude of third party products. With the rapid pace of software development and deployment to meet the needs of highly competitive companies involved in e-commerce, the focus has increasingly moved towards application functionality rather than performance. As a result, design or implementation flaws that negatively affect performance may not become apparent until after a significant increase in the number of users. If the problem is not quickly identified and solved, further scalability may not be possible and day to day operations of the business could be negatively affected.

Performance flaws in applications developed in-house are usually easier to correct than third party apps because face to face contact with developers is possible. Working with application vendors to correct performance problems is sometimes difficult and one's ability to motivate them to modify code is usually in proportion to your company's spending habits with the vendor.

With the increasing use of Java based applications and many different middleware products, performance problems related to the nonreuse of SQL statements are becoming more common. The focus of this article is on the diagnosis and correction of these types of problems. In general, the symptoms appear as intermittent sluggish to poor performance of applications. Conventional performance metrics may indicate no apparent problems with the database. In extreme cases, response time and throughput may degrade significantly on all database applications.

How SQL statements are processed

To understand the problem and develop a solution, one must understand how SQL statements are processed by the database. Each time an application connects to a database instance, it is ultimately assigned to a server process that resides on the database server. The server process provides the interface between the end-user application and the database. It is responsible for checking SQL statement syntax, determining how to execute the SQL statements, executing the statements, and returning the results to the end-user application. When a SQL statement is received by the server process, a *soft parse* occurs. During this phase, the statement syntax is checked for correctness, it is verified that the user has the privileges to access the objects referenced in the statement, and the SQL hash value (SHV) is computed on the statement text. The SHV is number used to easily identify the statement within the database. Next, the library cache in the SGA is searched to determine if the SHV already exists. This will occur if the same or another server process has already run the identical SQL statement. If the SHV is found, the server process retrieves the information stored in the library cache about the

statement. This includes the algorithms used to access the various objects in the query known as the *execution plan*. After the statement has been retrieved, if bind variables were present in the statement, the literal values are substituted and the operations specified by the execution plan are performed by the server process. This is the desired sequence of events since it produces the lowest overall resource cost for executing SQL statements and produces the fastest response time.

If the SHV corresponding to the SQL statement is not found in the library cache during the soft parse, the server process must perform a *hard parse* on the statement. During this operation, the execution plan for the statement must be determined and the result must be stored in the library cache. This is typically a computationally expensive step and may be time-consuming for several reasons. Depending upon the complexity of the statement, hundreds or even thousands of different permutations may be tried before the optimal execution plan is discovered. By default, the optimizer will test as many as 80,000 different execution plans to find the one with the lowest resource cost. To store the execution plan, memory must be allocated in the library cache. During this process, a number of latches must be acquired and held by the server process. Latches are simply *flags* or *semaphores* used within the database to assure that only one process at a time is writing to a specific part of a common memory object such as the library, dictionary, and buffer caches. If many sessions are concurrently performing hard parses, contention for these latches may result. This will decrease the response time for the query. If many simple unique SQL statements are executed by many sessions, the shared pool may become highly fragmented which will further exacerbate the latch contention problem. In extreme cases, latch contention may become so great that sessions are consuming more than 50% of their time waiting to acquire the latches necessary to write to the library cache. Response time and throughput for the queries will likely become unacceptable.

Detecting the problem

Several methods may be used to detect excessive parsing problems. The following query is useful for detecting programs that are performing excessive hard parses.

```
SELECT /*+ RULE */ s.program, COUNT(*) users,
      SUM(t.value) parses, SUM(t.value)/COUNT(*) parses_per_session,
      SUM(t.value)/(SUM(sysdate-s.logon_time)*24) parses_per_hour
FROM v$session s, v$sesstat t
WHERE t.statistic# = 153
AND s.sid = t.sid
GROUP BY s.program HAVING SUM(t.value)/COUNT(*) > 2.0
ORDER BY parses_per_hour DESC;
```

The query produces several parse metrics aggregated by program name. The *parses* column indicates the total hard parse count. *parses_per_session* is the average number of parses for all sessions running the program, and *parses_per_hour* is the average number of parses per hour for all sessions running the program. Search for high numbers in the *parses_per_hour* column. The term *high* is relative. For OLTP programs, numbers below 10 are reasonable. For batch programs, higher values are acceptable. Any programs with values higher than 10 should be investigated further. If a batch program is not using bind variables properly, values may number in the thousands.

For programs that are suspect, query the library cache to identify the SQL statements being executed using the following query:

```
SELECT /*+ RULE */ t.sql_text
  FROM v$sql t, v$session s
 WHERE s.sql_address = t.address
       AND s.sql_hash_value = t.hash_value
       AND s.sid = <SID of interest>;
```

Run this query as many times as are required to get a reasonable sample. Examine the statements to determine if they are using bind variables. Bind variables will be indicated as :b1, :b2, in the statements found in the library cache. A statement retrieved using the above query might look like:

```
SELECT cust_name, address1, address2
  FROM customers WHERE cust_no=:b1;
```

If literals appear in the statement as in the following example, bind variables are not being used.

```
SELECT cust_name, address1, address2
  FROM customers WHERE cust_no=15446;
```

Once offending programs and SQL statements have been identified, the overall effects on the database should be assessed. If the offending code is causing a significant negative database impact, several database configuration parameters can be adjusted to improve overall database performance until the application can be corrected. System resources will not be used efficiently under this scenario but at least a reasonable level of performance will be maintained.

As stated earlier, excessive parsing will result in higher than optimal CPU consumption. However, the greater impact is likely to be contention for resources in the shared pool. If many small statements are hard parsed, shared pool fragmentation is likely to result. As the shared pool becomes more fragmented, the amount of time required to complete a hard parse increases. As the process of executing many unique statements continues, resource contention worsens. The critical resources will likely be memory in the library cache and the various latches associated with the shared pool. There are several straightforward methods to detect contention. The following query shows a list events on which sessions are waiting to complete before continuing. Since v\$session_wait contains one row for each session, the query will return the total number of sessions waiting for each event. The view contains real-time data so it should be run repeatedly to detect possible problems.

```
SELECT /*+ RULE */ SUBSTR(event,1,30) event, COUNT(*)
  FROM v$session_wait
 WHERE wait_time = 0
 GROUP BY SUBSTR (event,1,30), state;
```

If the *latch free* event appears continuously, then there is latch resource contention. The following query can be used to determine which latches have contention. Since v\$latchholder contains one row for each session, the query will return the total number of sessions waiting for each latch. The view contains real-time data so it should be run repeatedly.

```
SELECT /*+ RULE */ name, COUNT(*)  
  FROM v$latchholder  
 GROUP BY name;
```

If *library cache* or *shared pool* latches appear continuously with any frequency, then there is contention.

The above two queries are simple methods that can be used to initially detect the problem. The next step is to determine the magnitude of the problem. Running Oracle's bstat/estat utility for a period of 15 minutes to one hour when the problem is suspected is a suitable method. bstat/estat produces various database performance reports over the sample period. In the *wait events* section of the report, check the *latch free* event. If the percent of time waiting for this event is higher than 1% of the total wait time, then there is latch contention. To determine which latches have contention, consult the latch statistics section of the report. If the hit ratio on the *library cache* or *shared pool* latches is less than 99%, then there is contention for these latches.

The next step is to interpret the cause of latch contention if it is found to exist. If few SQL statements were found that do not use bind variables, then the problem may simply be that the shared pool is too small. However, if many statements were found that do not use bind variables, then the problem is likely due to shared pool fragmentation and the unusually high frequency of hard parses.

The Quick Fix

Correcting the offending software may require days or weeks depending upon the nature of the problem. Application modifications and testing are likely. The time required to correct the program may be longer if the offending program is a commercial product. However, if performance is poor, there are some things that can be done to improve performance until the source of the problem can be corrected.

1. Increase the size of the shared pool. For minor contention problems, an increase of 20% should be suitable. For more severe problems, consider incremental increases of 50% until performance improves. If the host system has limited memory and the buffer cache hit rate is above 90%, consider reducing the size of the buffer cache to increase the size of the shared pool. A buffer cache hit ratio of 80-85% with reduced latch contention will likely produce better database performance than a higher buffer cache hit ratio with high latch contention.
2. Consider reducing the value of the *optimizer_max_permutations* parameter if the cost-based optimizer is being used and the database is using Oracle Enterprise Server Version 8.0 or higher. This parameter controls the maximum number of execution plans that the optimizer will develop to identify the one with the lowest cost. The default value is 80,000 but values of 100 to 1,000 usually produce identical execution

plans to those when a higher value is used. Since hard parses account for a significant amount of CPU consumed on short-running SQL statements, one of the artifacts of high hard parse counts is high CPU consumption. Reducing the value of *optimizer_max_permutations* will help mitigate the problem.

3. Flush the shared pool periodically. This will reduce memory fragmentation in the shared pool, which will reduce the elapsed time of the hard parse. The frequency depends upon the size of the shared pool and the severity of the problem. For mild problems, consider flushing twice each day. For severe problems, it may be necessary to flush the shared pool every few hours.
4. Pin frequently used PL/SQL functions and packages in the shared pool. When a program calls a method within a package, the entire package must be loaded into the shared pool. If the shared pool is highly fragmented and there is considerable latch contention, a significant amount of clock time may be required to load large packages into memory. Pinning packages and functions will improve the response time when they are accessed.

Identification and Correction

Correction of the offending application is the key to solving the problem and optimizing the use of system resources. This is typically a coordinated effort between the DBAs and developers. There are many things that can cause applications not to use bind variables.

Application developers may have written their code so that bind variables are not used. Most of the popular development languages have several methods for writing SQL statements. Often times, developers choose the easiest method which may not use bind variables. Another possibility is that SQL statements are constructed dynamically within the application because of complex business rules. If dynamic SQL is employed and the statements run for more than a few minutes, they usually do not contribute to the latch contention problem. The greatest contributors to the problem are likely to be SQL statements that execute frequently and are of short duration. Once these statements have been identified, make sure that developers use a coding method that employs bind variables.

If the application uses a multi-tier architecture, the cause of problem may lie in the tier that accesses the database. If the database access layer was developed in-house, it should be analyzed using the methods described above. If the product was developed by an application vendor, the source code is typically proprietary so code analysis is likely impossible. However, there are often some variables that control how the product interfaces to the database. If the product has configuration parameters, make sure that they are set properly to use bind variables. Make sure that the driver that is used to connect to the database supports bind variables. The native Oracle driver does support bind variables but some ODBC and JDBC drivers may not.

It is straightforward to verify that an application is using bind variables using the Oracle *trace* facility and *tkprof*, the application profiler. To use this facility, the database must

be configured properly and tracing must be enabled for the session of interest. Consult a database tuning book for more information about using this valuable tool.

tkprof produces a list of all SQL statements executed along with their execution plans and some performance statistics. These metrics are aggregated for each unique SQL statement. Inspect the output file and verify that bind variables are being used as described earlier. In addition, verify that excess parsing is not occurring. Below is an example of a query that was parsed once for each execution. Notice that in the *count* column, the number of parses is equal to the number of executions. The *Parse* row indicates the number of *hard parses* that occurred for the statement. In the ideal case, the statement would be parsed once and executed many times.

call	count	cpu	elapsed	disk	query	current	rows
Parse	27	0.02	0.00	0	0	0	0
Execute	27	0.00	0.00	0	0	0	0
Fetch	108	0.03	0.00	2	189	0	81
total	162	0.05	0.00	2	189	0	81

Once the application has been corrected, the size of the shared pool should be reevaluated to determine if it could be reduced to its original size. If shared pool flushes were employed as a temporary remedy, try to reduce the number of flushes to perhaps once per day. Excessive shared pool flushes will also result in performance degradation.

The long-term solution

To maintain steady and optimal database performance, the best approach is to correct application problems before they migrate to the production environment. Many problems can be detected and corrected early in the development cycle by working with software development teams. The cost of monitoring and remediation early in the development cycle is much less expensive and with little or no business impact compared with the alternative. If your company uses database applications developed by third party vendors, it is best to become involved when the functionality of the product is being evaluated. Surprisingly, many application design flaws can be detected during the application development or product evaluation phase, which would not otherwise become obvious until after a significant increase in the number of users.

Summary

While most database performance problems are rather easy to recognize, diagnose, and correct, performance problems caused by dynamic SQL or from not using bind variables are more elusive. Applications that perform a high number of hard parses are likely to cause excessive shared pool fragmentation and latch contention. Once the symptoms has been detected and the cause confirmed, several actions could be taken to improve database performance until the application can be corrected. The next step is to work with software developers to determine why an excessive number of hard parses are occurring and to correct the software accordingly. Finally, the long-term solution is to prevent an offending application from ever reaching the database by evaluating new applications during the development phase, or during product evaluation.