

Using Hotsos Tuning Techniques to Find and Fix Performance Issues: Six Real World Examples

Joe Johnson, OCP
Hotsos Symposium
March 2009

Introduction

Oracle database performance tuning is often viewed as a mysterious mixture of art and science. However, successful Oracle performance tuning really comes down to just three tasks: Identifying the specific problem, finding the cause of the problem, and then working to resolve that specific problem. Hotsos methodologies and tools provide reliable and reproducible techniques for performing each of these tasks. This paper provides examples of how these techniques were used to identify and resolve six real-world production performance problems.

Problem Identification

In order to fix a performance problem, you first need to know what the application is doing in the database during the times that the application is performing poorly. Because poor performance is often a symptom of slow SQL response time, a way is needed to accurately measure the application's response time. All of the examples in this paper used *Method R* to find the application SQL that were contributing to the slowest response times. *Method R* is described in *Optimizing Oracle Performance* by Cary Millsap and Jeffrey Holt (O'Reilly 2003). I've been applying this technique to tuning problems for over 5 years and have found, with few exceptions, that *Method R* is the single best method to accurately identify the true source of application performance issues.

At the heart of *Method R* is the collection of detailed database response time information for user sessions that are experiencing slow application performance. This response time information is collected using the 10046 trace event. The resulting raw trace files are then formatted using a profiler tool so that the captured SQL statements with the slowest response times can be identified, along with their execution plans, and associated wait events. Both commercial and open source profilers are available. The Hotsos Profiler is the premier commercial profiler tool for formatting 10046 trace files. All 10046 trace files in this paper were formatted using an open source profiler called OraSRP.

Problem Cause

Formatted 10046 trace files can also provide significant insight into why a particular SQL statement has a slow response time. This information can be determined by examining the amount of physical and logical I/O performed by the statement, the types of wait events the statement experiences, and the duration of those wait events. In addition to the 10046 trace information, many of the examples in this paper used the Hotsos SQL Test Harness tools to determine why a particular application SQL statement was experiencing slow response time and to provide clues for areas of possible performance improvement. The Hotsos SQL Test Harness is available to students who have attended the *Optimizing Oracle SQL, Intensive* Hotsos Education course. I highly recommend this course to anyone who wants to gain a clearer understanding of how the Oracle cost-based optimizer works, and how to improve the performance of problem SQL.

Problem Resolution

For most of the examples in this paper, the Hotsos SQL Test Harness tools were also used to compare alternatives when arriving at the problem solution. However, some of the identified performance problems were resolved using other techniques. Those techniques will be described in the sections for those examples.

Examples

The six real-world examples that will be discussed in this paper include:

- Example One: Full Table Scan on Large Table
- Example Two: The Lock Wait Problem That Wasn't
- Example Three: Corrupt Table Causes Slow Response Time
- Example Four: IS NULL Predicate Hurts Performance
- Example Five: SQL Slow in Application, Fast Outside Application
- Example Six: CBO Chooses Poor Join Method For Key Query

Example One: Full Table Scan on Large Table

Poorly performing application SQL is a common source of slow response time. Often this poor performance is caused by queries that perform full table scans of large tables when only a small subset of the total table data is ultimately returned by the query. This problem is usually avoided through the proper use of indexing and selective SQL predicates. However, in this example, full table scans were occurring on a large application table even though selective predicates should have encouraged the CBO to make use of an available index.

Problem Identification

A formatted 10046 trace file of a user session that was accessing the poorly performing application identified the SQL statement shown in Figure 1 as having the slowest overall response time.

Figure 1: Example One Formatted 10046 Trace File

Statement Text

```
select *
from (
select this_.LBPA_DOC_ID as LBPA1_19_1_, this_.DOC_TYPE_ID as DOC2_19_1_,
this_.FOLDER_ITEM_ID as FOLDER3_19_1_, this_.FILENET_BATCH_SEQ_NUM as
FILENET4_19_1_, this_.POLICY_NUM as POLICY5_19_1_, this_.STICKY_BATCH_NUM as
STICKY6_19_1_, this_.SCANNED_TS as SCANNED7_19_1_,
this_.SCANNED_INSURED_FIRST_NM as SCANNED8_19_1_,
this_.SCANNED_INSURED_LAST_NM as SCANNED9_19_1_, this_.CREATE_ROW_TS as
CREATE10_19_1_, this_.CREATE_ROW_USER_ID as CREATE11_19_1_,
this_.CREATE_ROW_PGM_NM as CREATE12_19_1_, this_.UPDATE_ROW_TS as
UPDATE13_19_1_, this_.UPDATE_ROW_USER_ID as UPDATE14_19_1_, this_.ACTIVE_IND
as ACTIVE15_19_1_, this_.SEND_TO_WORKFLOW_IND as SEND16_19_1_,
this_.AUTOMATED_FUNDS_TRANSFER_NUM as AUTOMATED17_19_1_,
this_.FILENET_IMAGE_CNT as FILENET18_19_1_
from LBPA.LBPA_DOC this_
where this_.ACTIVE_IND='Y' and this_.DOC_TYPE_ID in ('ADMIN')
order by this_.SCANNED_TS desc )
where rownum <= 25
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|----------|----------------|-----------------|----------------|-------------------|----------|-----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 1 | 1 | 0.0100s | 0.0080s | 0 | 0 | 0 | 0 |
| Exec | 0 | 1 | 0.0000s | 0.0001s | 0 | 0 | 0 | 0 |
| Fetch | | 1 | 9.5300s | 51.7363s | 70,609 | 70,609 | 0 | 25 |
| Total | 1 | 3 | 9.5400s | 51.7444s | 70,609 | 70,609 | 0 | 25 |
| Per Fch | 1.0 | 3.0 | 9.5400s | 51.7444s | 70,609.0 | 70,609.0 | 0.0 | 25.0 |
| Per Row | 0.0 | 0.1 | 0.3816s | 2.0698s | 2,824.4 | 2,824.4 | 0.0 | 1.0 |

The output in Figure 1 shows that the SQL statement was parsed and executed once, and did one fetch that performed a total of 141,218 I/Os (70,609 physical and 70,609 logical), taking 51.7444 seconds to return 25 rows of data. Since this query is issued multiple times as the user navigates the web-based application, it was this 52 second response time that was the cause of the slow response time experienced by the application users.

Problem Cause

Figure 1 shows us that most of the response time (51.7363 seconds) can be attributed to the statement's fetch operation. To understand why the fetch response time is so poor, we need to examine how the SQL statement is actually being executed. A similarly sized test system was used for this purpose. Figure 2 shows the execution plan for the SQL statement as generated by the Hotsos SQL Test Harness `do.sql` script.

Figure 2: Example One Original Execution Plan

```
select *
from (
select this_.LBPA_DOC_ID as LBPA1_19_1_, this_.DOC_TYPE_ID as DOC2_19_1_,
  this_.FOLDER_ITEM_ID as FOLDER3_19_1_, this_.FILENET_BATCH_SEQ_NUM as
  FILENET4_19_1_, this_.POLICY_NUM as POLICY5_19_1_, this_.STICKY_BATCH_NUM as
  STICKY6_19_1_, this_.SCANNED_TS as SCANNED7_19_1_,
  this_.SCANNED_INSURED_FIRST_NM as SCANNED8_19_1_,
  this_.SCANNED_INSURED_LAST_NM as SCANNED9_19_1_, this_.CREATE_ROW_TS as
  CREATE10_19_1_, this_.CREATE_ROW_USER_ID as CREATE11_19_1_,
  this_.CREATE_ROW_PGM_NM as CREATE12_19_1_, this_.UPDATE_ROW_TS as
  UPDATE13_19_1_, this_.UPDATE_ROW_USER_ID as UPDATE14_19_1_, this_.ACTIVE_IND
  as ACTIVE15_19_1_, this_.SEND_TO_WORKFLOW_IND as SEND16_19_1_,
  this_.AUTOMATED_FUNDS_TRANSFER_NUM as AUTOMATED17_19_1_,
  this_.FILENET_IMAGE_CNT as FILENET18_19_1_
from LBPA.LBPA_DOC this_
where this_.ACTIVE_IND='Y' and this_.DOC_TYPE_ID in ('ADMIN')
order by this_.SCANNED_TS desc )
where rownum <= 25
```

| Id | Operation | Name | Rows | Bytes | TempSpc | Cost(%CPU) | Time |
|-----|-----------------------|----------|------|-------|---------|------------|----------|
| 0 | SELECT STATEMENT | | 25 | 6150 | | 45982 (6) | 00:50:12 |
| * 1 | COUNT STOPKEY | | | | | | |
| 2 | VIEW | | 275K | 64M | | 45982 (6) | 00:50:12 |
| * 3 | SORT ORDER BY STOPKEY | | 275K | 26M | 74M | 45982 (6) | 00:50:12 |
| * 4 | TABLE ACCESS FULL | LBPA_DOC | 275K | 26M | | 39588 (7) | 00:45:56 |

Figure 2 shows that the LBPA_DOC table referenced in the query is being accessed using a full table scan (FTS). If the LBPA_DOC table is a large table, then that FTS operation is probably the cause of the slow response time experienced by the application users. Using the Hotsos SQL Test Harness `hgetstats.sql` script shown in Figure 3, we see that the table is indeed fairly large, using 138,628 blocks to store the table's 9,337,942 rows.

Figure 3: Example One LBPA_DOC Table Information

```
SQL> @hgetstats.sql
Enter the owner name: LBPA
Enter the table name: LBPA_DOC
Enter the display level (T)able, (C)olumn, (I)ndex, (A)ll: T
-----
Table: LBPA_DOC
-----
Rows: 9337942    Blocks: 138628    Avg Row Len: 105
```

Since we want to avoid a FTS on this large table, we should investigate whether appropriate indexes are in place for the columns used as predicates in the SQL statement: ACTIVE_IND and DOC_TYPE_ID. The Hotsos SQL Test Harness script `hix.sql` was used to examine the indexes on the LBPA_DOC table as shown in Figure 4.

Figure 4: Example One LBPA_DOC Available Indexes

```
SQL> @hix.sql
Enter the owner name: LBPA
Enter the table name: LBPA_DOC
```

| Index Name | Unique? | Height | Column Name |
|---------------------|----------|----------|---|
| LBPA_DOC_PK | Y | 2 | LBPA_DOC_ID |
| LBPA_DOC_X1 | N | 3 | DOC_TYPE_ID |
| LBPA_DOC_X10 | N | 3 | DOC_TYPE_ID ACTIVE_IND |
| LBPA_DOC_X11 | N | 3 | DOC_TYPE_ID ACTIVE_IND |
| | | | POLICY_NUM |
| LBPA_DOC_X2 | N | 1 | FOLDER_ITEM_ID |
| LBPA_DOC_X3 | N | 3 | POLICY_NUM |
| LBPA_DOC_X7 | N | 3 | POLICY_NUM ACTIVE_IND |
| LBPA_DOC_X8 | N | 2 | STICKY_BATCH_NUM ACTIVE_IND |

The query in Figure 4 shows that there are at least two indexes that use the DOC_TYPE_ID as the leading index column, and one index, LBPA_DOC_X10, includes both of the columns referenced in the predicate. With these indexes available, we have to wonder why the Oracle cost-based optimizer (CBO) is not making use of one of the indexes to avoid the FTS. One logical assumption might be that the predicates in the query are not very selective, and therefore a FTS is the more efficient than an index lookup when retrieving those rows. The queries in Figure 5 were used to determine the selectivity of these two columns.

Figure 5: Example One LBPA_DOC Row Selectivity

```
SQL> select count(*) from lbpa.lbpa_doc where active_ind = 'Y';

COUNT(*)
-----
9,337,391

SQL> select count(*) from lbpa.lbpa_doc where doc_type_id = 'ADMIN';

COUNT(*)
-----
277,752
```

The output in Figure 5 shows that any indexes on ACTIVE_IND are probably not helpful for this query because 99.99% of the rows have a value of Y for ACTIVE_IND. However, the second query shows that any index on the DOC_TYPE_ID column should be helpful for this query because only 2.98% of the rows have a value of Y for DOC_TYPE_ID. So why isn't the CBO using one of the available indexes on the DOC_TYPE_ID column? The answer is provided by the `hds.sql` script from the

Hotsos SQL Test Harness, This script compares block and row selectivity for a specified column and WHERE clause as shown in Figure 6.

Figure 6: Example One LBPA_DOC Block Selectivity

```
SQL> @hds
Table Owner   : LBPA
Table Name    : LBPA_DOC
Column List   : DOC_TYPE_ID
Where Clause  : where DOC_TYPE_ID in ( 'ADMIN' )
```

| Table blocks below hwm (B) | Table rows (R) |
|-------------------------------|-------------------|
| 138,628 | 9,337,942 |

| DOC_TYPE_ID | Block selectivity (pb = b/B) | Block count (b) | Row selectivity (pr = r/R) | Row count (r) |
|-------------|---------------------------------|--------------------|-------------------------------|------------------|
| ADMIN | 50.02% | 69,660 | 2.98% | 277,752 |

Figure 6 shows that while only 277,752, or 2.98%, of the rows in the LBPA_DOC table have a value of ADMIN in the DOC_TYPE_ID column, those 277,752 rows are stored in 50.02% of the table's blocks. Because 50% of the table blocks will need to be read in order to satisfy the query, the CBO is choosing to perform a FTS instead of using the available index. This makes sense because a FTS will result in fewer I/Os than accessing the index blocks, then the table's blocks, when 50% of the table blocks will ultimately need to be visited to satisfy the query. Therefore, this high block selectivity/low row selectivity disparity is the reason we are experiencing a FTS for this particular SQL statement.

Problem Resolution

The first question to answer is how did the rows develop this particular high block selectivity/low row selectivity disparity? In this case, the LBPA_DOC table was loaded from multiple legacy data sources as part of the application's implementation. Loading the data from these disparate sources caused the rows with like DOC_TYPE_IDs to be scattered throughout the table.

Our resolution to this issue was to unload the table's records, then reload the records in DOC_TYPE_ID order using the technique shown in Figure 7.

Figure 7: Unloading and Reloading LBPA_DOC in DOC_TYPE_ID Order

```
SQL> create table LBPA.LBPA_DOC_TMP nologging as
  2  select *
  3  from LBPA_DOC
  4  where 1=2;

SQL> insert /*+ APPEND */ into LBPA.LBPA_DOC_TMP
  2  select *
  3  from LBPA.LBPA_DOC;

SQL> truncate table LBPA.LBPA_DOC;
```

Figure 7 Continued: Unloading and Reloading LBPA_DOC in DOC_TYPE_ID Order

```
SQL> alter table LBPA.LBPA_DOC nologging;

SQL> insert /*+ APPEND */ into LBPA.LBPA_DOC
  2  select *
  3  from LBPA.LBPA_DOC_TMP
  4  order by DOC_TYPE_ID;

SQL> alter table LBPA.LBPA_DOC logging;
```

Once the operations in Figure 7 were complete, CBO statistics were generated and the hds.sql script was again used to analyze the block selectivity query, with the results shown in Figure 8.

Figure 8: Example One LBPA_DOC Block Selectivity After Table Reload

```
SQL> @hds
Table Owner   : LBPA
Table Name    : LBPA_DOC
Column List   : DOC_TYPE_ID
Where Clause  : where DOC_TYPE_ID in ( 'ADMIN' )

Table blocks below hwm      Table rows
      (B)                    (R)
-----
                138,628      9,337,942

DOC_TYPE_ID      Block selectivity  Block count  Row selectivity  Row count
      (pb = b/B)      (b)          (pr = r/R)      (r)
-----
ADMIN            2.63%              3,642        2.97%           277,782
```

Notice that Figure 8 clearly shows that the block and row selectivity for records with a value of ADMIN in the DOC_TYPE_ID column are now consistent: 2.97% of the rows meet that condition, and those rows are now stored in 2.63% of the table blocks. This should cause the CBO to favor the available LBPA_DOC_X1 index. To test this theory, the query was executed again using the do.sql script to determine the new execution plan shown in Figure 9.

Figure 9: Example One Execution Plan After Table Reload

```
select *
from (
select this_.LBPA_DOC_ID as LBPA1_19_1_, this_.DOC_TYPE_ID as DOC2_19_1_,
  this_.FOLDER_ITEM_ID as FOLDER3_19_1_, this_.FILENET_BATCH_SEQ_NUM as
  FILENET4_19_1_, this_.POLICY_NUM as POLICY5_19_1_, this_.STICKY_BATCH_NUM as
  STICKY6_19_1_, this_.SCANNED_TS as SCANNED7_19_1_,
  this_.SCANNED_INSURED_FIRST_NM as SCANNED8_19_1_,
  this_.SCANNED_INSURED_LAST_NM as SCANNED9_19_1_, this_.CREATE_ROW_TS as
  CREATE10_19_1_, this_.CREATE_ROW_USER_ID as CREATE11_19_1_,
  this_.CREATE_ROW_PGM_NM as CREATE12_19_1_, this_.UPDATE_ROW_TS as
  UPDATE13_19_1_, this_.UPDATE_ROW_USER_ID as UPDATE14_19_1_, this_.ACTIVE_IND
  as ACTIVE15_19_1_, this_.SEND_TO_WORKFLOW_IND as SEND16_19_1_,
  this_.AUTOMATED_FUNDS_TRANSFER_NUM as AUTOMATED17_19_1_,
  this_.FILENET_IMAGE_CNT as FILENET18_19_1_
from LBPA.LBPA_DOC this_
where this_.ACTIVE_IND='Y' and this_.DOC_TYPE_ID in ('ADMIN')
order by this_.SCANNED_TS desc )
where rownum <= 25
```

Figure 9 Continued: Example One Execution Plan After Table Reload

| Id | Operation | Name | Rows | Bytes | TempSpc | Cost (%CPU) | Time |
|-----|-----------------------------|-------------|------|-------|---------|-------------|----------|
| 0 | SELECT STATEMENT | | 25 | 6150 | | 13917 (3) | 00:02:48 |
| * 1 | COUNT STOPKEY | | | | | | |
| 2 | VIEW | | 293K | 68M | | 13917 (3) | 00:02:48 |
| * 3 | SORT ORDER BY STOPKEY | | 293K | 28M | 81M | 13917 (3) | 00:02:48 |
| * 4 | TABLE ACCESS BY INDEX ROWID | LBPA_DOC | 293K | 28M | | 6942 (2) | 00:01:24 |
| * 5 | INDEX RANGE SCAN | LBPA_DOC_X1 | 293K | | | 1250 (3) | 00:00:16 |

The reloading of the table in DOC_TYPE_ID order did cause the CBO to use the available index and drastically reduced the SQL statement’s response time. Figure 10 uses the Hotsos SQL Test Harness diff.sql script to compare the I/O activity and response times for the SQL statement before and after the table reload.

Figure 10: Example One Before-After Comparison

| TYPE | NAME | BEFORE | AFTER | DIFFERENCE |
|-------|------------------------------|--------|--------|------------|
| Latch | cache buffers chains | 809401 | 61911 | 747490 |
| | library cache | 22380 | 138 | 22242 |
| | row cache objects | 1842 | 54198 | -52356 |
| | shared pool | 850 | 83 | 767 |
| Stats | buffer is pinned count | 0 | 551925 | -551925 |
| | consistent gets | 138476 | 4322 | 134154 |
| | db block changes | 0 | 0 | 0 |
| | db block gets | 0 | 0 | 0 |
| | execute count | 9 | 9 | 0 |
| | index fast full scans (full) | 0 | 0 | 0 |
| | parse count (hard) | 3 | 2 | 1 |
| | parse count (total) | 10 | 10 | 0 |
| | physical reads | 138353 | 0 | 138353 |
| | physical writes | 0 | 0 | 0 |
| | redo size | 0 | 0 | 0 |
| | session logical reads | 138476 | 4322 | 134154 |
| | session pga memory | 65536 | 65536 | 0 |
| | session pga memory max | 0 | 0 | 0 |
| | session uga memory | -6976 | 0 | -6976 |
| | session uga memory max | 0 | 0 | 0 |
| | sorts (disk) | 0 | 0 | 0 |
| | sorts (memory) | 1 | 1 | 0 |
| | sorts (rows) | 277782 | 277782 | 0 |
| | table fetch by rowid | 4 | 277787 | -277783 |
| | table scan blocks gotten | 138414 | 0 | 138414 |
| | table scans (long tables) | 1 | 0 | 1 |
| | table scans (short tables) | 0 | 0 | 0 |
| Time | elapsed time (centiseconds) | 2265 | 86 | 2079 |

The output shown in Figure 10 demonstrates that the improvement in response time was achieved by using the available index on DOC_TYPE_ID to reduce the total number of logical and physical I/Os that were required to satisfy the query, thereby reducing the query’s elapsed time from 22.65 seconds to .86 seconds – a 96% reduction in response time for this key application query.

Example Two: The Lock Wait Problem That Wasn't

Lock waits can be another source of slow application response time. If multiple users are performing DML on the same records simultaneously, the users waiting for locks on those rows to be released perceive that wait as slow application response time whenever those lock waits are excessive. In this example, the application uses a partitioned table to store application data that is loaded by a nightly batch process. The batch job starts ten concurrent processes to load application data, and the batch process had been running with acceptable response times for nearly a year. But, suddenly, on January 5, 2009, the job's execution time went from minutes, to hours.

Problem Identification

One drawback of *Method R* is that it relies on trace files that must be collected while the application is experiencing slow performance. In this example, the DBA team was not notified of the performance problem until the following morning, after the nightly load jobs had completed their tasks. Therefore, Quest Software's Performance Analyzer for Oracle monitoring tool was initially used to try and identify the source of the response time problem. The Quest tool was used because it stores historical performance data and would allow us to see what things looked like in the database the night before. The Quest tool showed several lengthy lock waits on the application table that was being loaded by the ten concurrent nightly batch processes. Furthermore, the Quest tool also showed that indeed, the same lock waits were not present in the weeks and months leading up to that point. Additionally, it was determined that all but one of the ten batch processes timed out on their associated lock waits, and only one batch process (presumably the one holding the lock) was able to do any processing. By all indications, the problem was caused by a locking issue.

The information provided by the Quest tools was helpful, but did not contain enough session-level detail about response time for us to know with certainty that locking was the problem. Therefore, 10046 traces were generated for all ten batch processes during the next night's load operation in order to get a more detailed look into what was happening during the load process, A formatted example of one of those ten process trace files appears in Figure 11.

Figure 11: Example Two Formatted 10046 Trace File Showing Lock Waits

Statement Text

```
INSERT INTO auto_rating (PRODUCT_STATUS_CD, PRODUCT_ID, CREATE_ROW_TS,
    UNIVERSAL_UNIQUE_ID, SYSTEM_NM, RISK_ID, XML_DOC_TXT) VALUES (:1, :2, :3, :4,
    :5, :6, XMLType(:7))
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|--------------|----------------|-----------------|----------------|-------------------|---------------|------------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 0 | 644 | 0.0400s | 0.0498s | 0 | 0 | 0 | 0 |
| Exec | 0 | 644 | 7.9600s | 72.3262s | 15,780 | 26,766 | 16,641 | 642 |
| Fetch | 0 | 0 | 0.0000s | 0.0000s | 0 | 0 | 0 | 0 |
| Total | 0 | 1,288 | 8.0000s | 72.3761s | 15,780 | 26,766 | 16,641 | 642 |
| Per Exe | 0.0 | 2.0 | 0.0124s | 0.1124s | 24.5 | 41.6 | 25.8 | 1.0 |
| Per Row | 0.0 | 2.0 | 0.0124s | 0.1127s | 24.6 | 41.7 | 25.9 | 1.0 |

Figure 11 Continued: Example Two Formatted 10046 Trace File Showing Lock Waits

| Event Name | % Time | Seconds | Calls | - Time per Call - | | |
|------------------------------------|---------------|-----------------|--------|-------------------|---------|---------|
| | | | | Avg | Min | Max |
| eng: TX - contention | 38.2% | 25.0645s | 9 | 2.7849s | 1.7842s | 2.9490s |
| eng: HW - contention | 22.4% | 14.7109s | 8 | 1.8389s | 0.0000s | 2.9479s |
| db file sequential read | 14.0% | 9.1653s | 773 | 0.0118s | 0.0000s | 0.2576s |
| log file sync | 13.2% | 8.6493s | 196 | 0.0441s | 0.0067s | 0.6127s |
| EXEC calls [CPU] | 11.7% | 7.6900s | 644 | 0.0119s | 0.0000s | 0.0300s |
| SQL*Net message from client [idle] | 0.3% | 0.1802s | 307 | 0.0005s | 0.0003s | 0.0049s |
| log buffer space | 0.1% | 0.0630s | 1 | 0.0630s | 0.0630s | 0.0630s |
| PARSE calls [CPU] | 0.0% | 0.0400s | 644 | 0.0000s | 0.0000s | 0.0100s |
| direct path read temp | 0.0% | 0.0375s | 13,834 | 0.0000s | 0.0000s | 0.0044s |
| library cache load lock | 0.0% | 0.0374s | 4 | 0.0093s | 0.0063s | 0.0159s |
| direct path write | 0.0% | 0.0102s | 1,037 | 0.0000s | 0.0000s | 0.0001s |
| buffer busy waits | 0.0% | 0.0024s | 10 | 0.0002s | 0.0000s | 0.0011s |
| SQL*Net message to client | 0.0% | 0.0008s | 197 | 0.0000s | 0.0000s | 0.0000s |
| latch: cache buffers chains | 0.0% | 0.0005s | 8 | 0.0000s | 0.0000s | 0.0002s |
| SQL*Net break/reset to client | 0.0% | 0.0005s | 1 | 0.0005s | 0.0005s | 0.0005s |
| Total | 100.0% | 65.6529s | | | | |

The output in Figure 11 shows that the SQL statement was parsed and executed 644 times, and performed a total of 59,187 I/Os (15,780 physical and 43,407 logical), taking approximately 0.1124 seconds per execution. This is relatively fast on a per-execution basis. However, Figure 11 also shows that waits for the lock-related events *eng: TX - contention* and *eng: HW - contention* comprised the majority of the response time. In fact, nine of the ten trace files that were collected during the batch load process exhibited identical behavior to that shown in Figure 11. Again, all indications seem to be that locking and waits on lock-related events are the cause of the issue.

Closer investigation of the locks being held during the load process indicated that the locks were not just on the AUTO_RATING table, but on a sub-partition of that table as shown in Figure 12.

Figure 12: Example Two Sub-partition Locking

```
SQL> select decode(vl.request,0,'holder: ','waiter: ')||vl.sid "STATUS: SID",
2         dbo.owner, dbo.object_name, dbo.object_type,
3         vl.lmode, vl.request, vl.type
4 from v$sqllock vl, v$sqllocked_object vlo, dba_objects dbo
5 where vl.sid=vlo.session_id
6 and vlo.object_id=dbo.object_id
7 and (id1, id2, type) in (select id1, id2, type from v$sqllock where request>0)
8 order by id1, request;
```

| STATUS: SID | OWNER | OBJECT_NAME | OBJECT_TYPE | LMODE | REQUEST | TYPE |
|-------------|-------|-----------------------------|------------------|-------|---------|------|
| Holder: 252 | RPS | SYS_LOB0000057290C00009\$\$ | LOB SUBPARTITION | 6 | 0 | TX |
| Waiter: 269 | RPS | SYS_LOB0000057290C00009\$\$ | LOB SUBPARTITION | 0 | 4 | TX |
| Waiter: 245 | RPS | SYS_LOB0000057290C00009\$\$ | LOB SUBPARTITION | 0 | 4 | TX |
| Waiter: 244 | RPS | SYS_LOB0000057290C00009\$\$ | LOB SUBPARTITION | 0 | 4 | TX |
| Waiter: 259 | RPS | SYS_LOB0000057290C00009\$\$ | LOB SUBPARTITION | 0 | 4 | TX |

Further investigation of this issue requires us to know more about how the underlying table partitions are configured. Figure 13 shows the DDL used to create the AUTO_RATING table. This DDL shows that the table is range partitioned by CREATE_ROW_TS, and list sub-partitioned by SYSTEM_NM.

Figure 13: Example Two DDL for AUTO_RATING Table

```

CREATE TABLE RPS.AUTO_RATING (
  AUTO_RATING_ID NUMBER(38, 0) NOT NULL,
  CREATE_ROW_TS TIMESTAMP (6) NOT NULL,
  SYSTEM_NM VARCHAR2(60) NOT NULL,
  XML_DOC TXT SYS.XMLTYPE NOT NULL ENABLE,
  CONSTRAINT AUTO_RATING_PK PRIMARY KEY(AUTO_RATING_ID)
  USING INDEX TABLESPACE RPS_INDEX)
XMLTYPE XML_DOC TXT STORE AS CLOB (CHUNK 32000 NOCACHE LOGGING)
PARTITION BY RANGE (CREATE_ROW_TS)
SUBPARTITION BY LIST (SYSTEM_NM)
(PARTITION RPS_2008 VALUES LESS THAN (TO_DATE('1-JAN-2009','DD-MON-YYYY'))
(
  SUBPARTITION RPS_GRP01_2008 VALUES ('AUTOPLUS','MRATE') TABLESPACE RPS_GRP01_2008,
  SUBPARTITION RPS_GRP02_2008 VALUES ('SISAUTO','VISW','OASIS') TABLESPACE RPS_GRP02_2008
),
PARTITION RPS_2009 VALUES LESS THAN (TO_DATE('1-JAN-2010','DD-MON-YYYY'))
(
  SUBPARTITION RPS_GRP01_2009 VALUES ('AUTOPLUS','MRATE') TABLESPACE RPS_GRP01_2009,
  SUBPARTITION RPS_GRP02_2009 VALUES ('SISAUTO','VISW','OASIS') TABLESPACE RPS_GRP02_2009))

```

All of this locking information was useful, but since neither the structure of the table, nor the application SQL accessing the table had changed, there was still no clear reason as to *why* sub-partition locking was suddenly an issue when the system had worked fine just days before.

Problem Cause

A second, more thorough examination of the formatted 10046 trace files was performed. This analysis again indicated that ten processes had significantly long response times for the INSERT operation shown in Figure 11. But closer examination showed that one of the ten trace files was not waiting on an enqueue lock, but another event called *Data file init write*. The portion of the formatted 10046 trace file showing this wait appears in Figure 14.

Figure 14: Example Two Wait on Data File Init Write

| Event Name | % Time | Seconds | Calls | - Time per Call - | | |
|--|---------------|-----------------|-------|-------------------|---------|---------|
| | | | | Avg | Min | Max |
| <u>Data file init write</u> | 34.0% | 23.3569s | 102 | 0.2290s | 0.0000s | 2.2449s |
| <u>eng: ix - contention</u> | 29.4% | 20.2239s | 7 | 2.8891s | 2.3641s | 2.9490s |
| <u>log file sync</u> | 13.3% | 9.1391s | 262 | 0.0348s | 0.0025s | 0.4103s |
| <u>db file sequential read</u> | 11.3% | 7.7694s | 756 | 0.0102s | 0.0000s | 0.0947s |
| <u>EXEC calls [CPU]</u> | 11.3% | 7.7600s | 629 | 0.0123s | 0.0000s | 0.9100s |
| <u>SQL*Net message from client [idle]</u> | 0.5% | 0.3316s | 442 | 0.0007s | 0.0003s | 0.0328s |
| <u>PARSE calls [CPU]</u> | 0.1% | 0.0800s | 629 | 0.0001s | 0.0000s | 0.0100s |
| <u>control file parallel write</u> | 0.0% | 0.0497s | 3 | 0.0165s | 0.0064s | 0.0312s |
| <u>rdbms ipc reply</u> | 0.0% | 0.0386s | 4 | 0.0096s | 0.0001s | 0.0256s |
| <u>direct path read temp</u> | 0.0% | 0.0237s | 9,787 | 0.0000s | 0.0000s | 0.0023s |
| <u>direct path write</u> | 0.0% | 0.0095s | 991 | 0.0000s | 0.0000s | 0.0000s |
| <u>eng: HW - contention</u> | 0.0% | 0.0059s | 4 | 0.0014s | 0.0000s | 0.0035s |
| <u>db file single write</u> | 0.0% | 0.0019s | 1 | 0.0019s | 0.0019s | 0.0019s |
| <u>buffer busy waits</u> | 0.0% | 0.0018s | 10 | 0.0001s | 0.0000s | 0.0004s |
| <u>control file sequential read</u> | 0.0% | 0.0010s | 34 | 0.0000s | 0.0000s | 0.0001s |
| <u>SQL*Net message to client</u> | 0.0% | 0.0010s | 262 | 0.0000s | 0.0000s | 0.0000s |
| <u>latch: cache buffers chains</u> | 0.0% | 0.0009s | 4 | 0.0002s | 0.0000s | 0.0005s |
| <u>change tracking file synchronous read</u> | 0.0% | 0.0009s | 4 | 0.0002s | 0.0000s | 0.0005s |
| <u>SQL*Net break/reset to client</u> | 0.0% | 0.0006s | 2 | 0.0003s | 0.0001s | 0.0005s |
| Total | 100.0% | 68.7970s | | | | |

Metalink analysis of this wait event indicated time was being spent acquiring a new extent for a segment, or extending a datafile for the tablespace that stores a segment, or both. Therefore the storage parameters for the AUTO_RATING table and its tablespace were examined in more detail as shown in Figure 15.

Figure 15: Example Two Storage Parameters for AUTO_RATING Table

```
SQL> select table_owner, table_name, partition_name,
2  subpartition_name, tablespace_name, next_extent
3  from dba_tab_subpartitions
4  where table_name = 'AUTO_RATING';
```

| TABLE_OWNER | TABLE_NAME | PARTITION_NAME | SUBPARTITION_NAME | TABLESPACE_NAME | NEXT_EXTENT |
|-------------|-------------|----------------|-------------------|-----------------|-------------|
| RPS | AUTO_RATING | RPS_2008 | RPS_GRP02_2008 | RPS_GRP02_2008 | 104857600 |
| RPS | AUTO_RATING | RPS_2008 | RPS_GRP01_2008 | RPS_GRP01_2008 | 104857600 |
| RPS | AUTO_RATING | RPS_2009 | RPS_GRP02_2009 | RPS_GRP02_2009 | 104857600 |
| RPS | AUTO_RATING | RPS_2009 | RPS_GRP01_2009 | RPS_GRP01_2009 | 104857600 |

```
SQL> select tablespace_name, file_name, round(bytes/1048576) "SIZE_MB", increment_by
2  from dba_data_files
3  where tablespace_name like 'RPS_GRP0%'
4  order by 1,2
```

| TABLESPACE_NAME | FILE_NAME | SIZE_MB | INCREMENT_BY |
|-----------------------|---|--------------|--------------|
| RPS_GRP01_2008 | /u22100/oradata/rps0p/rps_2008_grp01_001.dbf | 10000 | 32000 |
| ... | | | |
| RPS_GRP01_2008 | /u22100/oradata/rps0p/rps_2008_grp01_035.dbf | 10000 | 32000 |
| RPS_GRP02_2008 | /u22100/oradata/rps0p/rps_2008_grp02_001.dbf | 10000 | 32000 |
| ... | | | |
| RPS_GRP02_2008 | /u22100/oradata/rps0p/rps_2008_grp02_035.dbf | 10000 | 32000 |
| RPS_GRP01_2009 | /u22100/oradata/rps0p/rps_2009_grp01_001.dbf | 2265 | 1 |
| RPS_GRP02_2009 | /u22100/oradata/rps0p/rps_2009_grp02_001.dbf | 3824 | 1 |

The rows highlighted in bold in the query results reveal the problem. Recall that this performance problem began on 05-JAN-09. Because the table is partially partitioned by date, records started to be written to the RPS_GRP01_2009 and RPS_GRP02_2009 tablespaces for the first time on 01-JAN-09. Further, because the AUTO_RATING table makes use of a CLOB, each table insert uses a fairly large amount of storage - in fact, the previous year's data used almost 850GB of storage. However, it appears that the RPS_GRP01_2009 and RPS_GRP02_2009 tablespaces were initially created very small, and were configured to grow by only one database block (8K in this case) with each autoextend request. With the LOB's CHUNK size specified at a size of 32000 bytes as shown in the DLL in Figure 13, each insert was causing the table space to extend by 4 database blocks, and this occurred over and over again with each insert being performed by the ten concurrent load processes. It was this wait for the datafile to autoextend which was in turn leading to the long lock waits seen in the other session's 10046 trace files.

Problem Resolution

The RPS_GRP01_2009 and RPS_GRP02_2009 tablespaces were manually grown to a more appropriate size of 50GB each - enough to hold several week's data. When the nightly batch jobs were re-run the following evening, all the lock waits were gone and the jobs completed well within their normal run times.

Example Three: Corrupt Table Causes Slow Response Time

Method R is particularly useful for troubleshooting commercial applications where the application SQL is not readily available. In this example, the Oracle database that was the repository for a commercial web-based application was timing out when a key application task was being performed, even though the database had been configured in accordance with the vendor's recommendations, and all other portions of the application were working as expected.

Problem Identification

A 10046 trace was created for a session that was exercising the portion of the application that was experiencing the timeout. A portion of that formatted trace file showing the SQL and execution plan of the statement with the slowest response time is shown in Figure 16.

Figure 16: Example Three Formatted 10046 Trace File

Statement Text

```
select
  e.OBJECT_ID,e.OBJECT_CLASS,e.IS_OBJECT,e.END1_ID,e.END2_ID,i.ATTR_ID,i.ATTR_VALUE,tmp.DATA_DATE
from HIST_EVENTS e,HIST_INF_LONG i, IDS_TO_DATES_TMP tmp
where e.ID=i.ID and e.CUSTOMER_ID=:v0 and e.EVENT_TIME<=tmp.DATA_DATE and
  i.END_TIME>tmp.DATA_DATE and tmp.CMDB_ID=e.OBJECT_ID
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|----------|------------------|------------------|----------------|-------------------|----------|----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 1 | 1 | 0.0000s | 0.0626s | 0 | 0 | 0 | 0 |
| Exec | 1 | 1 | 0.0100s | 0.0354s | 0 | 0 | 0 | 0 |
| Fetch | | 1 | 324.8400s | 363.0394s | 0 | 1,634,175 | 0 | 0 |
| Total | 2 | 3 | 324.8500s | 363.1376s | 0 | 1,634,175 | 0 | 0 |
| Per Fch | 2.0 | 3.0 | 324.8500s | 363.1376s | 0.0 | 1,634,175.0 | 0.0 | 0.0 |
| Per Row | 2.0 | 3.0 | 324.8500s | 363.1376s | 0.0 | 1,634,175.0 | 0.0 | 0.0 |

Statement Plan

met 1 time

| Rows | Row Source Operation | Object Id |
|------------|--|-----------|
| 0 | TABLE ACCESS BY INDEX ROWID HIST_EVENTS (cr=0 pr=0 pw=0 time=0.0000s) | 13407 |
| 87,270,921 | NESTED LOOPS (cr=279,070 pr=0 pw=0 time=87.2838s) | |
| 447 | NESTED LOOPS (cr=12 pr=0 pw=0 time=0.0153s) | |
| 1 | TABLE ACCESS FULL IDS_TO_DATES_TMP (cr=3 pr=0 pw=0 time=0.0001s) | 13930 |
| 447 | TABLE ACCESS FULL HIST_INF_LONG (cr=9 pr=0 pw=0 time=0.0134s) | 13443 |
| 87,270,474 | INDEX RANGE SCAN HIST_EVENTS_TIME_I (cr=279,058 pr=0 pw=0 time=87.3691s) | 13409 |

The output in Figure 16 shows that the SQL statement was parsed and executed once, and performed one fetch that performed a total of 1,634,175 logical I/Os, taking 363.1376 seconds to return no rows. Further, the execution plan shows that 87,270,474 row IDs were returned by the INDEX RANGE SCAN on the HIST_EVENTS_TIME_I index on the HIST_EVENTS table. In order to determine if this was a reasonable amount of I/O for this query, the size of each of the tables involved in the query were examined as shown in Figure 17.

Figure 17: Example Three Table Statistics

```
SQL> select table_name, num_rows, blocks, temporary, last_analyzed
  2  from dba_tables
  3  where table_name in ('HIST_EVENTS','HIST_INF_LONG','IDS_TO_DATES_TMP')
```

| TABLE_NAME | NUM_ROWS | BLOCKS | T | LAST_ANAL |
|------------------|----------|--------|---|-----------|
| HIST_EVENTS | 3287651 | 51917 | N | 04-AUG-08 |
| HIST_INF_LONG | 10203 | 88 | N | 04-AUG-08 |
| IDS_TO_DATES_TMP | | | Y | |

This query output shows two interesting findings. First, the query output shows that the `IDS_TO_DATES_TMP` table is of the type temporary and is lacking CBO statistics. However, since the `init.ora` parameter `optimizer_dynamic_sampling=2` for this database, global temporary tables were dynamically being analyzed by the CBO at parse time so missing CBO statistics was unlikely to be the cause of the slow response time. More importantly the query also shows that the `HIST_EVENTS` table contains 3,287,651 rows. However, the execution plan in Figure 16 showed that more than *eighty-seven million* row IDs were being returned by the INDEX RANGE SCAN on the `HIST_EVENTS_TIME_I` index on the `HIST_EVENTS` table. This large discrepancy between actual table rows and the number of row IDs returned by the index scan led us to use the `hgetstats.sql` Hotsos SQL Test Harness script to examine the `HIST_EVENT_TIME_I` index for possible anomalies. The output of the `hgetstats.sql` script is shown in Figure 18.

Figure 18: Example Three Index Statistics on Suspect Index

```
SQL> @hgetstats.sql
Enter the owner name: CMDB_HIST
Enter the table name: HIST_EVENTS
Enter the display level (T)able, (C)olumn, (I)ndex, (A)ll: I
-----
Table: HIST_EVENTS
-----
Rows: 3287651  Blocks: 51917  Avg Row Len: 107
----- Index Statistics -----
-----
Index name      : HIST_EVENTS_TIME_I
Index type      : NORMAL
Last analyzed   : 04-AUG-08
Degree         : 1
Partitioned     : NO
Rows           : 3238995
Levels         : 2
Leaf Blocks     : 15331
Distinct Keys   : 2917
Avg LB/Key     : 5
Avg DB/Key     : 23
Clust. Factor   : 68962
Table Rows     : 3287651
Table Blocks    : 51917
```

The output in Figure 18 shows that the index statistics accurately reflected the indexed rows in the underlying table, so this index was also removed from consideration as cause of the slow response time.

Recall that the problem query accessed three tables: HIST_EVENTS, HIST_INF_LONG, and a global temporary table called IDS_TO_DATES_TMP. So the next approach was to incrementally build the query to determine when the poor response time started to occur. First, a modified version of the query was run using only the HIST_EVENTS table. This query had excellent response time. Next, a modified version of the query was run using only the HIST_EVENTS and HIST_INF_LONG tables. This query also had excellent response time. Finally, the original query including the global temporary table IDS_TO_DATES_TMP was run, and the poor response time returned.

Problem Cause

Since the inclusion of the global temporary table appeared to be the source of the poor response time, a quick test was performed to determine if the temporary table itself was the source of the issue, or whether the additional join operation that results by adding a third table (of any type) was the cause of the problem. To perform this test, a non-temporary table called ERASEME was created as a copy of the IDS_TO_DATES_TMP table, then substituted into the original query as shown in Figure 19.

Figure 19: Example Three Query Testing With New Temporary Table

```
SQL> create table cmdb_hist.eraseme
  2 as select * from cmdb_hist.ids_to_dates_tmp;
```

Statement Text

```
SELECT e.object_id, e.object_class, e.is_object, e.end1_id, e.end2_id,
       i.attr_id, i.attr_value, tmp.data_date
FROM cmdb_hist.hist_events e, cmdb_hist.hist_inf_long i, cmdb_hist.eraseme tmp
WHERE e.id = i.id
      AND e.customer_id = 1
      AND e.event_time <= tmp.data_date
      AND i.end_time > tmp.data_date
      AND tmp.cmdb_id = e.object_id
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|----------|----------------|----------------|----------------|-------------------|----------|----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 1 | 1 | 0.0100s | 0.0092s | 0 | 0 | 0 | 0 |
| Exec | 0 | 1 | 0.0000s | 0.0001s | 0 | 0 | 0 | 0 |
| Fetch | | 1 | 0.0000s | 0.0000s | 0 | 0 | 0 | 0 |
| Total | 1 | 3 | 0.0100s | 0.0094s | 0 | 0 | 0 | 0 |
| Per Fch | 1.0 | 3.0 | 0.0100s | 0.0094s | 0.0 | 0.0 | 0.0 | 0.0 |
| Per Row | 1.0 | 3.0 | 0.0100s | 0.0094s | 0.0 | 0.0 | 0.0 | 0.0 |

Statement Plan

met 1 time

| Rows | Row Source Operation | Object Id |
|------|---|-----------|
| 0 | TABLE ACCESS BY INDEX ROWID HIST_EVENTS (cr=0 pr=0 pw=0 time=0.0000s) | 13407 |
| 1 | NESTED LOOPS (cr=0 pr=0 pw=0 time=0.0000s) | |
| 0 | NESTED LOOPS (cr=0 pr=0 pw=0 time=0.0000s) | |
| 0 | TABLE ACCESS FULL ERASEME (cr=0 pr=0 pw=0 time=0.0000s) | 13930 |
| 0 | TABLE ACCESS FULL HIST_INF_LONG (cr=0 pr=0 pw=0 time=0.0000s) | 13443 |
| 0 | INDEX RANGE SCAN HIST_EVENTS_TIME_I (cr=0 pr=0 pw=0 time=0.0000s) | 13409 |

The results were dramatic. When a different table, with the same structure and data as the original temporary table, was substituted into the query, the response time was sub-second. This indicated that something about the global temporary table IDS_TO_DATES_TMP was causing the poor response time for this query. Even though the same index on the HIST_INF_LOG table was being used, the index scan was no longer returning 87M row IDs, but zero.

Problem Resolution

Based on the results of the test show in Figure 19, it was decided to drop and recreate the global temporary table. The Oracle-supplied DBMS_METADATA package was used to generate the DDL for the temporary table, then the table was dropped and recreated as show in Figure 20.

Figure 20: Example Three Recreate Temporary Table

```
SQL> DROP TABLE CMDB_PERSIST.IDS_TO_DATES;

SQL> CREATE GLOBAL TEMPORARY TABLE CMDB CMDB_PERSIST.IDS_TO_DATES
  2  (CMDB_ID RAW(16), T_VALUES_INDEX NUMBER) ON COMMIT PRESERVE ROWS;
```

Following the re-creation of the table as shown in Figure 20, the previously slow application operations were found to have sub-second response times, essentially mirroring the results from the test using the ERASEME table. Because the original table was dropped, we were not able to determine the true root cause of the problem with the original IDS_TO_DATES_TMP temporary table. However, the problem has not re-occurred again.

Example Four: IS NULL Predicate Hurts Performance

Because of the way that Oracle handles NULLS in indexed columns, the use of an IS NULL operator in a predicate can cause the CBO to ignore available indexes on predicate columns. This example shows how that limitation caused an application query to perform poorly.

Problem Identification

An important operation in a web-based application was experiencing slow response time, causing the application to experience time-outs. A formatted 10046 trace file collected by a session experiencing the time-out is show in Figure 21.

Figure 21: Example Four Formatted 10046 Trace File

SQL Hash Value: 1855881994 uid: 39 depth: 0 optimizer mode: ALL_ROWS

Statement Text

```
select count(*) as col_0_0_
from JBPM.JBPM_TASKINSTANCE ewstaskins0_
where ewstaskins0_.CLASS_='E' and (ewstaskins0_.APPID_ in (:1)) and
(ewstaskins0_.ACTORID_ in (:2)) and ((ewstaskins0_.START_ is null) and
(ewstaskins0_.END_ is null) and ewstaskins0_.ISSUSPENDEED_=0 and
ewstaskins0_.ISOPEN_=1 and ewstaskins0_.ISCANCELLED_=0 or (ewstaskins0_.START_
is not null) and (ewstaskins0_.END_ is null) and ewstaskins0_.ISSUSPENDEED_=0
and ewstaskins0_.ISOPEN_=1 and ewstaskins0_.ISCANCELLED_=0) and
(ewstaskins0_.CREATE_ is not null)
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|----------|----------------|-----------------|----------------|-------------------|----------|----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 1 | 1 | 0.0000s | 0.0012s | 0 | 0 | 0 | 0 |
| Exec | 1 | 1 | 0.0100s | 0.0053s | 0 | 0 | 0 | 0 |
| Fetch | | 1 | 2.4800s | 29.1125s | 7,804 | 14,711 | 0 | 1 |
| Total | 2 | 3 | 2.4900s | 29.1191s | 7,804 | 14,711 | 0 | 1 |
| Per Fch | 2.0 | 3.0 | 2.4900s | 29.1191s | 7,804.0 | 14,711.0 | 0.0 | 1.0 |
| Per Row | 2.0 | 3.0 | 2.4900s | 29.1191s | 7,804.0 | 14,711.0 | 0.0 | 1.0 |

Figure 21 Continued: Example Four Formatted 10046 Trace File

Statement Plan

met 1 time

| Rows | Row Source Operation | Object Id |
|--------|---|-----------|
| 1 | SORT AGGREGATE (cr=14,711 pr=7,804 pw=0 time=29.1124s) | |
| 2 | TABLE ACCESS BY INDEX ROWID JBPM_TASKINSTANCE (cr=14,711 pr=7,804 pw=0 time=29.1124s) | 10033 |
| 42,850 | INDEX RANGE SCAN IDX_TASK_ACTORID (cr=160 pr=159 pw=0 time=1.1990s) | 10151 |

The output in Figure 21 shows that the SQL statement was parsed and executed once, and performed one fetch that performed a total of 22,515 I/Os (7,804 physical and 14,711 logical), taking 29.1191 seconds to return 1 row of data. The execution plan in Figure 21 shows that the query is using an available index, called `IDX_TASK_ACTORID` when processing the query, but that the table lookup that follows that index scan is where the bulk of the time is being spent - 29.1124 seconds.

Problem Cause

In order to troubleshoot this query, the first thing we should do is examine the predicates, see what indexes are available on those predicates, and what impact each predicate might be having on the execution plan that the CBO is choosing. Figure 22 shows the available indexes on the table.

Figure 22: Example Four Available Indexes

```
SQL> @hgetstats.sql
Enter the owner name: jbpm
Enter the table name: jbpm_taskinstance
Enter the display level (T)able, (C)olumn, (I)ndex, (A)ll: T
-----
Table: JBPM_TASKINSTANCE
-----
Rows: 5381740  Blocks: 87617  Avg Row Len: 121
```

```
SQL> @hix.sql
Enter the owner name: jbpm
Enter the table name: jbpm_taskinstance
```

| Index Name | Unique? | Height | Column Name |
|---------------------------|---------|--------|-------------------|
| IDX_TASKINSTANCE_NAME | N | 3 | NAME_ |
| IDX_TASKINSTANCE_PROCINST | N | 2 | PROCINST_ |
| IDX_TASKINST_TASK | N | 2 | TASK_ |
| IDX_TASKINST_TOKN | N | 2 | TOKEN_ |
| IDX_TASKINST_TSK | N | 2 | TASK_ |
| | | | PROCINST_ |
| IDX_TASK_ACTORID | N | 3 | ACTORID_ |
| IDX_TSKINST_SLINST | N | 1 | SWIMLANINSTANCE_ |
| IDX_TSKINST_TMINST | N | 2 | TASKMGMTINSTANCE_ |
| JBPM_TASKINSTANCE_PK | N | 3 | ID_ |

Since some of the predicates in the WHERE clause do not appear to be indexed, an examination of each of the predicate columns was performed. A summary of that analysis is shown in Table 1.

Table 1: Example Four Predicate Column Analysis

| WHERE Clause | Observations |
|---|---|
| where ewstaskins0_.CLASS_='E' and | No index, but all rows have a value of E |
| (ewstaskins0_.APPID_ in (:1)) | No index, but all rows have the same value |
| and (ewstaskins0_.ACTORID_ in (:2)) | Indexed, IDX_TASK_ACTORID currently being used |
| and ((ewstaskins0_.START_ is null) | No index, 3,273 of the 5.3M table row are NULL |
| and (ewstaskins0_.END_ is null) | No index, 6,854 of the 5.3M table row are NULL |
| and ewstaskins0_.ISSUSPENDED_=0 and | No index, but almost all rows have a value of 0 |
| ewstaskins0_.ISOPEN_=1 and | No index, 6,864 of the 5.3M table row are NULL |
| ewstaskins0_.ISCANCELLED_=0 or | No index, but almost all rows have a value of 0 |
| (ewstaskins0_.START_ is not null) | Same as above |
| and (ewstaskins0_.END_ is null) | Same as above |
| and ewstaskins0_.ISSUSPENDED_=0 | Same as above |
| and ewstaskins0_.ISOPEN_=1 | Same as above |
| and ewstaskins0_.ISCANCELLED_=0) | Same as above |
| and (ewstaskins0_.CREATE_ is not null) | No index, but almost all rows are NOT NULL |

Table 1 shows that the only useful index currently available to this query - the index on the ACTORID_ column - is already being used. However, we know from the execution plan in Figure 21 that this index causes 42,850 row IDs to be returned. But we also know that the most restrictive predicate used in both sections of the WHERE clause is END_ IS NULL because that predicate will return only 6,854 records. Therefore, by using the index on ACTORID_, we end up reading many rows that we ultimately discard because their END_ date IS NULL. If we could avoid doing this, the overall response time of the query should improve.

However, simply creating an index on the END_ column will not resolve this issue because single column, B*Tree indexes do not include entries for null column values, so checking for END_ IS NULL in the WHERE clause would negate the use of such an index, even though it would be very helpful because only 6,854 of the 5,381,740 rows in the table have a END_ value of null.

Problem Resolution

If at least one column on a multi-column index is non-NULL, then the index *will* store NULL values and will let us filter the null from the result set. Further examination of the records in JBPM_TASKINSTANCE table showed that while the ACTORID_ column was nullable, it contained very few null values, and there were no records where both ACTORID_ and END_ were null.

Therefore, a new composite index was created on the JBPM_TASKINSTANCE table that included both the END_ and ACTORID_ columns as shown in Figure 23.

Figure 23: Example Four New Composite Index

```
SQL> create index jbpm.IDX_TASKINST_END_ACTORID
  2  on jbpm.jbpm_taskinstance (end_, actorid_)
  3  tablespace jbpm_index;
```

```
SQL> exec dbms_stats.gather_index_stats('JBPM','IDX_TASKINST_END_ACTORID');
```

Once the new index was in place, a 10046 trace file was again generated for a session running the query. The formatted output of that trace file is shown in Figure 24.

Figure 24: Example Four Formatted 10046 Trace File After New Index

Statement Text

```
select count(*) as col_0_0_
from JBPM.JBPM_TASKINSTANCE ewstaskins0
where ewstaskins0.CLASS_='E' and (ewstaskins0.APPID_ in (:1)) and
(ewstaskins0.ACTORID_ in (:2)) and ((ewstaskins0.START_ is not null) and
(ewstaskins0.END_ is null) and ewstaskins0.ISSUSPENDED_=0 and
ewstaskins0.ISOPEN=1 and ewstaskins0.ISCANCELLED_=0 or (ewstaskins0.START_
is null) and (ewstaskins0.END_ is null) and ewstaskins0.ISSUSPENDED_=0 and
ewstaskins0.ISOPEN=1 and ewstaskins0.ISCANCELLED_=0) and
(ewstaskins0.CREATE_ is not null)
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|----------|----------------|----------------|----------------|-------------------|----------|----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 2 | 2 | 0.0000s | 0.0025s | 0 | 0 | 0 | 0 |
| Exec | 2 | 2 | 0.0300s | 0.0238s | 0 | 0 | 0 | 0 |
| Fetch | | 2 | 0.0000s | 0.0008s | 0 | 20 | 0 | 2 |
| Total | 4 | 6 | 0.0300s | 0.0271s | 0 | 20 | 0 | 2 |
| Per Fch | 2.0 | 3.0 | 0.0150s | 0.0135s | 0.0 | 10.0 | 0.0 | 1.0 |
| Per Row | 2.0 | 3.0 | 0.0150s | 0.0135s | 0.0 | 10.0 | 0.0 | 1.0 |

met 1 time

| Rows | Row Source Operation | Object Id |
|------|---|-----------|
| 1 | SORT AGGREGATE (cr=10 pr=0 pw=0 time=0.0002s) | |
| 2 | CONCATENATION (cr=10 pr=0 pw=0 time=0.0002s) | |
| 0 | TABLE ACCESS BY INDEX ROWID JBPM_TASKINSTANCE (cr=5 pr=0 pw=0 time=0.0001s) | 10033 |
| 4 | INDEX RANGE SCAN IDX_TASKINST_END_ACTORID (cr=3 pr=0 pw=0 time=0.0000s) | 36438 |
| 2 | TABLE ACCESS BY INDEX ROWID JBPM_TASKINSTANCE (cr=5 pr=0 pw=0 time=0.0000s) | 10033 |
| 4 | INDEX RANGE SCAN IDX_TASKINST_END_ACTORID (cr=3 pr=0 pw=0 time=0.0000s) | 36438 |

The output in Figure 24 shows that the SQL statement was parsed and executed twice, and performed two fetches that performed a total of 20 logical I/Os, taking just 0.0135 seconds to return 1 row of data with each execution. The execution plan in Figure 24 shows that the new index is being used, and that the CBO has generated an entirely new execution plan based on that index.

Example Five: SQL Slow in Application, Fast Outside Application

Most modern applications have many layers to their technology stack: Web browsers, physical network hardware, application server hardware and software, database connectors (e.g. JDBC, ODBC), and database server hardware and software. This example demonstrates how *Method R* can be very helpful when trying to determine which layer of the application technology stack is causing slow response time - it's not always the database!

Problem Identification

An application upgrade was applied to a vendor application. Following the upgrade, the response time for nearly all application queries was very slow. A formatted portion of a 10046 trace file that was collected for an application process is shown in Figure 25.

Figure 25: Example Five Formatted Trace File From Application

Statement Text

```
SELECT
  SE_ID,SE_OWNER,SE_GROUP,SE_WS_ADR,SE_STATE,SE_DATE,SE_SUSPEND_DATE,SE_CATALOG_ID,SE_ADMIN,SE_FILE_C,SE_USER_C,SE_GRP_C
FROM SESSIONTAB
WHERE (SE_ID = :V001)
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|-----------|----------------|----------------|----------------|-------------------|----------|----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 0 | 6 | 0.0000s | 0.0002s | 0 | 0 | 0 | 0 |
| Exec | 0 | 6 | 0.0000s | 0.0008s | 0 | 0 | 0 | 0 |
| Fetch | | 6 | 0.0000s | 0.0003s | 0 | 12 | 0 | 6 |
| Total | 0 | 18 | 0.0000s | 0.0014s | 0 | 12 | 0 | 6 |
| Per Fch | 0.0 | 3.0 | 0.0000s | 0.0002s | 0.0 | 2.0 | 0.0 | 1.0 |
| Per Row | 0.0 | 3.0 | 0.0000s | 0.0002s | 0.0 | 2.0 | 0.0 | 1.0 |

Statement Flat Profile

| Event Name | % Time | Seconds | Calls | - Time per Call - | | |
|------------------------------------|---------------|--------------------|-------|-------------------|---------|-------------|
| | | | | Avg | Min | Max |
| SQL*Net message from client [idle] | 100.0% | 1,381.0303s | 14 | 98.6450s | 0.0003s | 1,381.0235s |
| SQL*Net message from client | 0.0% | 0.0169s | 18 | 0.0009s | 0.0003s | 0.0025s |
| log file sync | 0.0% | 0.0029s | 2 | 0.0014s | 0.0004s | 0.0025s |
| SQL*Net message to client | 0.0% | 0.0000s | 31 | 0.0000s | 0.0000s | 0.0000s |
| FETCH calls [CPU] | 0.0% | 0.0000s | 6 | 0.0000s | 0.0000s | 0.0000s |
| PARSE calls [CPU] | 0.0% | 0.0000s | 6 | 0.0000s | 0.0000s | 0.0000s |
| EXEC calls [CPU] | 0.0% | 0.0000s | 6 | 0.0000s | 0.0000s | 0.0000s |
| Total | 100.0% | 1,381.0503s | | | | |

Statement Plan

met 6 times

| Rows | Row Source Operation | Object Id |
|------|---|-----------|
| 6 | TABLE ACCESS BY INDEX ROWID SESSIONTAB (cr=12 pr=0 pw=0 time=0.0001s) | 12047 |
| 6 | INDEX UNIQUE SCAN UI2816 (cr=6 pr=0 pw=0 time=0.0000s) | 12048 |

The output in Figure 25 shows that the SQL statement was parsed and executed six times, and performed six fetches that performed a total of 12 logical I/Os, taking 0.0002 seconds to return 1 row of data per execution. Despite these fast response times, the application users were reporting slow performance from the application while performing these same actions. The top wait events in Figure 5 may reveal the issue. While the trace file shows a sub-second response time for the query, there were still very high wait times for the *SQL*Net message from client [idle]* event.

Problem Cause

The extremely long response times for the *SQL*Net message from client [idle]* event shown in Figure 25 would seem to indicate that the problem may lie in the communication between the application and the database, not with the database SQL calls themselves.

To test this theory, the same SQL statement was issued from the application server using SQL*Plus. The results of that test are shown in the formatted 10046 trace shown in Figure 26.

Figure 26: Example Five Formatted Trace File From SQL*Plus

Statement Text

```
SELECT
  SE_ID,SE_OWNER,SE_GROUP,SE_WS_ADR,SE_STATE,SE_DATE,SE_SUSPEND_DATE,SE_CATALOG_ID,SE_ADMIN,SE_FILE_C,SE_USER_C,SE_GRP_C
FROM SESSIONTAB
WHERE (SE_ID = :V001)
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|-----------|----------------|----------------|----------------|-------------------|----------|-----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 1 | 18 | 0.0000s | 0.0018s | 0 | 0 | 0 | 0 |
| Exec | 1 | 18 | 0.0000s | 0.0034s | 0 | 0 | 0 | 0 |
| Fetch | | 18 | 0.0000s | 0.0012s | 0 | 36 | 0 | 18 |
| Total | 2 | 54 | 0.0000s | 0.0065s | 0 | 36 | 0 | 18 |
| Per Fch | 0.1 | 3.0 | 0.0000s | 0.0003s | 0.0 | 2.0 | 0.0 | 1.0 |
| Per Row | 0.1 | 3.0 | 0.0000s | 0.0003s | 0.0 | 2.0 | 0.0 | 1.0 |

Statement Flat Profile

| Event Name | % Time | Seconds | Calls | - Time per Call - | | |
|------------------------------------|---------------|----------------|-------|-------------------|---------|---------|
| | | | | Avg | Min | Max |
| SQL*Net message from client [idle] | 94.0% | 1.1666s | 35 | 0.0333s | 0.0003s | 0.9594s |
| SQL*Net message from client | 3.6% | 0.0440s | 54 | 0.0008s | 0.0003s | 0.0011s |
| log file sync | 2.4% | 0.0301s | 4 | 0.0075s | 0.0010s | 0.0145s |
| SQL*Net message to client | 0.0% | 0.0000s | 91 | 0.0000s | 0.0000s | 0.0000s |
| direct path read | 0.0% | 0.0000s | 6 | 0.0000s | 0.0000s | 0.0000s |
| EXEC calls [CPU] | 0.0% | 0.0000s | 18 | 0.0000s | 0.0000s | 0.0000s |
| FETCH calls [CPU] | 0.0% | 0.0000s | 18 | 0.0000s | 0.0000s | 0.0000s |
| PARSE calls [CPU] | 0.0% | 0.0000s | 18 | 0.0000s | 0.0000s | 0.0000s |
| Total | 100.0% | 1.2409s | | | | |

Statement Plan

met 18 times

| Rows | Row Source Operation | Object Id |
|------|---|-----------|
| 6 | TABLE ACCESS BY INDEX ROWID SESSIONTAB (cr=36 pr=0 pw=0 time=0.0005s) | 12047 |
| 6 | INDEX UNIQUE SCAN UI2816 (cr=18 pr=0 pw=0 time=0.0002s) | 12048 |

Figure 26 shows that the test produced the same execution plan and similar response times to those found in the original trace file in Figure 25. But, the test demonstrated that running the SQL statement from the application server, but from outside the application, produced much lower wait times for the *SQL*Net message from client [idle]* event.

Problem Resolution

Given the findings of the test shown in Figure 26, the application owners reviewed the portion of the application technology stack that managed the connection between the application server and the database. This investigation revealed that the application used a vendor-supplied database connector to communicate with the database. This custom connector had a number of configurable parameters, one of which was modified by the recent application upgrade. Changing these connector settings back to their previous values reduced the wait times for the *SQL*Net message from client [idle]* event to less than 1 second and resolved the application performance issue.

Example Six: CBO Chooses Poor Join Method for Key Query

Despite its sophistication, the CBO occasionally chooses inefficient execution plans that adversely affect application performance. When this occurs, it is important to try and determine why this is happening before simply assuming the CBO is “wrong.” Recall that Example One showed a case where, on the surface, the CBO appeared to be making a bad choice not to use an available index – until we discovered that the underlying table had a high block selectivity/low row selectivity disparity at the physical level. In this example, we see a case where the CBO is not choosing an effective execution plan and steps we can take to influence the CBO.

Problem Identification

The response time for a web-based application used to manage workflow tasks was experiencing slow response times for several key application queries. A formatted 10046 trace of a user session running the problem queries showed that the SQL in Figure 27 had the slowest response time.

Figure 27: Example Six Formatted Trace File

Statement Text

```
select *
from (
select row_.*, rownum rownum_
from (
select workitemdb0.WRK_ID as WRK_IDO_, workitemdb0.LST_UPDT_TS as
LST_UPDT2_O_, workitemdb0.USER_ID as USER_IDO_, workitemdb0.WRK_SRC_TXT as
WRK_SRC_4_O_, workitemdb0.SUSP_TS as SUSP_TSO_, workitemdb0.DUE_TS as
DUE_TSO_, workitemdb0.CRTE_TS as CRTE_TSO_, workitemdb0.LST_UPDT_PGM_NM as
LST_UPDT8_O_, workitemdb0.LST_UPDT_USERID as LST_UPDT9_O_,
workitemdb0.STS_CD as STS_CDO_, workitemdb0.APP_ID as APP_IDO_,
workitemdb0.WRK_TYP_CD as WRK_TYP_CDO_, workitemdb0.WRK_QUE_ID as
WRK_QUE_IDO_, workitemdb0.UNQ_ID as UNQ_IDO_, workitemdb0.ASSIGN_TS as
ASSIGN_TSO_, workitemdb0.ACTION_CD as ACTION_CDO_, workitemdb0.RECD_TS as
RECD_TSO_, workitemdb0.WEIGHT as WEIGHTO_, workitemdb0.AUTO_ASSIGN_IND as
AUTO_AS19_O_, workitemdb0.CRTE_ROW_PGM_NM as CRTE_RO20_O_,
workitemdb0.CRTE_ROW_USERID as CRTE_RO21_O_
from WFGEN3.WRK workitemdb0 left join WFGEN3.WRK_REFER workitemresort0 on
(workitemdb0.WRK_ID=workitemresort0.WRK_ID and workitemresort0.REFER_KEY_CD=
:1 )
where (workitemdb0.STS_CD in( :2, :3, :4, :5, :6 )) and (workitemdb0.USER_ID
in ( :7 ))
order by workitemresort0.REFER_TXT asc, WRK_IDO_desc ) row_
where rownum <= :8)
where rownum_ > :9
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|-----------|-----------------|-----------------|----------------|-------------------|----------|-----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 0 | 2 | 0.0000s | 0.0002s | 0 | 0 | 0 | 0 |
| Exec | 0 | 3 | 0.0100s | 0.0041s | 0 | 0 | 0 | 0 |
| Fetch | | 9 | 35.7400s | 39.6014s | 30 | 174,155 | 0 | 75 |
| Total | 0 | 14 | 35.7500s | 39.6059s | 30 | 174,155 | 0 | 75 |
| Per Fch | 0.0 | 1.6 | 3.9722s | 4.4007s | 3.3 | 19,350.6 | 0.0 | 8.3 |
| Per Row | 0.0 | 0.2 | 0.4767s | 0.5281s | 0.4 | 2,322.1 | 0.0 | 1.0 |

Figure 27 Continued: Example Six Formatted Trace File**Statement Plan**

met 2 times

| Rows | Row Source Operation | Object Id |
|------------|---|-----------|
| 75 | VIEW (cr=174,155 pr=30 pw=0 time=39.5978s) | |
| 150 | COUNT STOPKEY (cr=174,155 pr=30 pw=0 time=39.5975s) | |
| 150 | VIEW (cr=174,155 pr=30 pw=0 time=39.5973s) | |
| 150 | SORT ORDER BY STOPKEY (cr=174,155 pr=30 pw=0 time=39.5971s) | |
| 311 | HASH JOIN OUTER (cr=174,155 pr=30 pw=0 time=39.5924s) | |
| 311 | TABLE ACCESS BY INDEX ROWID WRK (cr=497 pr=30 pw=0 time=0.1412s) | 18356 |
| 311 | INDEX RANGE SCAN WRK_COUNT_INBOX_IDX (cr=198 pr=0 pw=0 time=0.0359s) | 18722 |
| 22,839,355 | INDEX RANGE SCAN WRK_REFER_COUNT_IDX (cr=173,658 pr=0 pw=0 time=19.6758s) | 18925 |

The output in Figure 27 shows that the SQL statement was parsed twice, executed three times, and performed nine fetches that performed a total of 174,185 I/Os (30 physical and 174,155,609 logical), taking 4.4007 seconds to return about 8 rows of data with each execution. Figure 27 also shows that the WRK_REFER_COUNT_IDX and WRK_COUNT_INBOX_IDX indexes and a hash join was chosen by the CBO.

Problem Cause

The SQL from Figure 27 was examined, and alternative queries were formulated in an effort to reduce the response time for the query below one second. Specifically, the addition of a FIRST_ROWS hint was found to have a substantially better response time because it caused the CBO to favor the WRK_REFER_X1 index instead of the WRK_REFER_COUNT_IDX index, and utilized a nested loops join instead of a hash join operation. Figure 28 shows a formatted 10046 trace file showing the execution plan from the hinted SQL.

Figure 28: Example Six Hinted SQL Execution Plan**Statement Text**

```
select /*+ FIRST_ROWS */ *
from (
select row_*, rownum rownum
from (
select workitemdb0.WRK_ID as WRK_IDO_, workitemdb0.LST_UPDT_TS as
LST_UPDT2_0, workitemdb0.USER_ID as USER_IDO_, workitemdb0.WRK_SRC_TXT as
WRK_SRC_4_0, workitemdb0.SUSP_TS as SUSP_TSO_, workitemdb0.DUE_TS as
DUE_TSO_, workitemdb0.CRTE_TS as CRTE_TSO_, workitemdb0.LST_UPDT_PGM_NM as
LST_UPDT8_0, workitemdb0.LST_UPDT_USERID as LST_UPDT9_0,
workitemdb0.STS_CD as STS_CDO_, workitemdb0.APP_ID as APP_IDO_,
workitemdb0.WRK_TYP_CD as WRK_TYP_CDO_, workitemdb0.WRK_QUE_ID as
WRK_QUE_IDO_, workitemdb0.UNQ_ID as UNQ_IDO_, workitemdb0.ASSIGN_TS as
ASSIGN_TSO_, workitemdb0.ACTION_CD as ACTION_CDO_, workitemdb0.RECD_TS as
RECD_TSO_, workitemdb0.WEIGHT as WEIGHTO_, workitemdb0.AUTO_ASSIGN_IND as
AUTO_AS19_0, workitemdb0.CRTE_ROW_PGM_NM as CRTE_RO20_0,
workitemdb0.CRTE_ROW_USERID as CRTE_RO21_0
from WFGEN3.WRK workitemdb0 left join WFGEN3.WRK_REFER workitemresort0 on
(workitemdb0.WRK_ID=workitemresort0.WRK_ID and workitemresort0.REFER_KEY_CD=
'PTYNM' )
where (workitemdb0.STS_CD in ( 'NEW', 'NEWREASGN', 'ACT', 'ACTSUSP', 'REOPEN' ))
and (workitemdb0.USER_ID
in ( 'SAB013' ))
order by workitemresort0.REFER_TXT asc, WRK_IDO_ desc ) row_
where rownum <= 50)
where rownum > 25
```

Figure 28 Continued: Example Six Hinted SQL Execution Plan

Statement Plan

met 1 time

| Rows | Row Source Operation | Object Id |
|------|--|-----------|
| 0 | VIEW (cr=112 pr=2 pw=0 time=0.0169s) | |
| 7 | COUNT STOPKEY (cr=112 pr=2 pw=0 time=0.0169s) | |
| 7 | VIEW (cr=112 pr=2 pw=0 time=0.0169s) | |
| 7 | SORT ORDER BY STOPKEY (cr=112 pr=2 pw=0 time=0.0168s) | |
| 7 | NESTED LOOPS OUTER (cr=112 pr=2 pw=0 time=0.0167s) | |
| 7 | TABLE ACCESS BY INDEX ROWID OBJ#(18356) (cr=80 pr=2 pw=0 time=0.0163s) | 18356 |
| 7 | INDEX RANGE SCAN OBJ#(18722) (cr=73 pr=2 pw=0 time=0.0163s) | 18722 |
| 7 | TABLE ACCESS BY INDEX ROWID OBJ#(18311) (cr=32 pr=0 pw=0 time=0.0003s) | 18311 |
| 29 | INDEX RANGE SCAN OBJ#(18460) (cr=24 pr=0 pw=0 time=0.0002s) | 18460 |

The execution plan in Figure 28 shows object IDs instead of object names. For clarity, Figure 29 shows the execution plan including object names as generated by the Hotsos SQL Test Harness `do.sql` script.

Figure 29: Example Six Hinted SQL Execution Plan

| Id | Operation | Name | Rows | Bytes | Cost |
|-----|-----------------------------|---------------------|------|-------|-------|
| 0 | SELECT STATEMENT | | 75 | 17175 | 18736 |
| * 1 | VIEW | | 75 | 17175 | 18736 |
| * 2 | COUNT STOPKEY | | | | |
| 3 | VIEW | | 714 | 150K | 18736 |
| * 4 | SORT ORDER BY STOPKEY | | 714 | 123K | 18736 |
| 5 | NESTED LOOPS OUTER | | 714 | 123K | 18717 |
| 6 | TABLE ACCESS BY INDEX ROWID | WRK | 695 | 97300 | 647 |
| * 7 | INDEX RANGE SCAN | WRK_COUNT_INBOX_IDX | 695 | | 14 |
| * 8 | TABLE ACCESS BY INDEX ROWID | WRK_REFER | 1 | 37 | 26 |
| * 9 | INDEX RANGE SCAN | WRK_REFER_X1 | 5 | | 3 |

As evidence that the use of the `FIRST_ROWS` hint improves the response time of this query, the Hotsos SQL Test Harness `diff.sql` script was used to compare the overall activity of two queries. Figure 30 shows the output of the `diff.sql` script.

Figure 30: Example Six Comparison of Original and Hinted SQL

| TYPE | NAME | ORIGINAL_SQL | FIRST_ROWS_HINT | DIFFERENCE |
|-------|------------------------------|--------------|-----------------|------------|
| Latch | cache buffers chains | 1845544 | 5051 | 1840493 |
| | library cache | 57976 | 238 | 57738 |
| | row cache objects | 8858 | 266 | 8592 |
| | shared pool | 21751 | 188 | 21563 |
| Stats | buffer is pinned count | 130 | 1095 | -965 |
| | consistent gets | 65624 | 576 | 65048 |
| | db block changes | 6 | 0 | 6 |
| | db block gets | 0 | 0 | 0 |
| | execute count | 5 | 6 | -1 |
| | index fast full scans (full) | 0 | 0 | 0 |
| | parse count (hard) | 4 | 2 | 2 |
| | parse count (total) | 5 | 6 | -1 |
| | physical reads | 65415 | 0 | 65415 |
| | physical writes | 0 | 0 | 0 |
| | redo size | 456 | 0 | 456 |
| | session logical reads | 65624 | 576 | 65048 |
| | session pga memory | 0 | 0 | 0 |
| | session pga memory max | 0 | 0 | 0 |
| | session uga memory | 0 | 0 | 0 |
| | session uga memory max | 65408 | 0 | 65408 |
| | sorts (disk) | 0 | 0 | 0 |
| | sorts (memory) | 1 | 1 | 0 |
| | sorts (rows) | 129 | 97 | 32 |
| | table fetch by rowid | 130 | 650 | -520 |
| | table scan blocks gotten | 0 | 0 | 0 |
| | table scans (long tables) | 0 | 0 | 0 |
| | table scans (short tables) | 0 | 0 | 0 |
| Time | elapsed time (centiseconds) | 8141 | 12 | 8129 |

The output in Figure 30 shows that there was 99% fewer logical I/Os and a much improved elapsed time for the hinted version of the query. Having determined that the FIRST_ROWS hint will provide the desired response time for this critical application query, the next challenge is how to implement that hint - given that application code cannot be easily modified.

Problem Resolution

Since the application code cannot easily be modified to include the FIRST_ROWS hint, the Oracle Stored Outlines feature was used to achieve the desired results. Implementing the stored outline is a four-step process:

1. Create an outline using the original “good” SQL, but with the “bad” execution plan as chosen by the CBO.
2. Create a second stored outline using “bad” hinted SQL, but with the “good” execution plan.
3. Delete the execution plan associated with the “good” SQL and replace it with the “good” execution plan from the hinted SQL.
4. Created a logon trigger so that the outline is available to application users (optional).

Steps 1 and 2 above are shown in the SQL in Figure 31, which creates two stored outlines, one for the original application SQL, and the other for the same SQL with the FIRST_ROWS hint.

Figure 31: Example Six Creation of Two Stored Outlines

```

-- outline with "good" SQL, but "bad" execution plan
create or replace outline ICS_IB_2053266053_ERASEME for category ICS_OLTP
on
select *
from (
select row_.*, rownum rownum_
from (
select workitemdb0.WRK_ID as WRK_ID0_, workitemdb0.LST_UPDT_TS as
LST_UPDT2_0_, workitemdb0.USER_ID as USER_ID0_, workitemdb0.WRK_SRC_TXT as
WRK_SRC_4_0_, workitemdb0.SUSP_TS as SUSP_TS0_, workitemdb0.DUE_TS as
DUE_TS0_, workitemdb0.CRTE_TS as CRTE_TS0_, workitemdb0.LST_UPDT_PGM_NM as
LST_UPDT8_0_, workitemdb0.LST_UPDT_USERID as LST_UPDT9_0_,
workitemdb0.STS_CD as STS_CD0_, workitemdb0.APP_ID as APP_ID0_,
workitemdb0.WRK_TYP_CD as WRK_TYP_CD0_, workitemdb0.WRK_QUE_ID as
WRK_QUE_ID0_, workitemdb0.UNQ_ID as UNQ_ID0_, workitemdb0.ASSIGN_TS as
ASSIGN_TS0_, workitemdb0.ACTION_CD as ACTION_CD0_, workitemdb0.RECD_TS as
RECD_TS0_, workitemdb0.WEIGHT as WEIGHT0_, workitemdb0.AUTO_ASSIGN_IND as
AUTO_AS19_0_, workitemdb0.CRTE_ROW_PGM_NM as CRTE_RO20_0_,
workitemdb0.CRTE_ROW_USERID as CRTE_RO21_0_
from WFGEN3.WRK workitemdb0 left join WFGEN3.WRK_REFER workitemresort0 on
(workitemdb0.WRK_ID=workitemresort0.WRK_ID and workitemresort0.REFER_KEY_CD=
:1 )
where (workitemdb0.STS_CD in( :2, :3, :4, :5, :6 )) and (workitemdb0.USER_ID
in ( :7 ))
order by workitemresort0.REFER_TXT asc, WRK_ID0_ asc ) row_
where rownum <= :8)
where rownum_ > :9
/

-- outline with "bad" SQL, but "good" execution plan
create or replace outline ICS_IB_2053266053 for category ICS_OLTP
on
select /*+ FIRST_ROWS */ *
from (
select row_.*, rownum rownum_
from (
select workitemdb0.WRK_ID as WRK_ID0_, workitemdb0.LST_UPDT_TS as
LST_UPDT2_0_, workitemdb0.USER_ID as USER_ID0_, workitemdb0.WRK_SRC_TXT as
WRK_SRC_4_0_, workitemdb0.SUSP_TS as SUSP_TS0_, workitemdb0.DUE_TS as
DUE_TS0_, workitemdb0.CRTE_TS as CRTE_TS0_, workitemdb0.LST_UPDT_PGM_NM as
LST_UPDT8_0_, workitemdb0.LST_UPDT_USERID as LST_UPDT9_0_,
workitemdb0.STS_CD as STS_CD0_, workitemdb0.APP_ID as APP_ID0_,
workitemdb0.WRK_TYP_CD as WRK_TYP_CD0_, workitemdb0.WRK_QUE_ID as
WRK_QUE_ID0_, workitemdb0.UNQ_ID as UNQ_ID0_, workitemdb0.ASSIGN_TS as
ASSIGN_TS0_, workitemdb0.ACTION_CD as ACTION_CD0_, workitemdb0.RECD_TS as
RECD_TS0_, workitemdb0.WEIGHT as WEIGHT0_, workitemdb0.AUTO_ASSIGN_IND as
AUTO_AS19_0_, workitemdb0.CRTE_ROW_PGM_NM as CRTE_RO20_0_,
workitemdb0.CRTE_ROW_USERID as CRTE_RO21_0_
from WFGEN3.WRK workitemdb0 left join WFGEN3.WRK_REFER workitemresort0 on
(workitemdb0.WRK_ID=workitemresort0.WRK_ID and workitemresort0.REFER_KEY_CD=
:1 )
where (workitemdb0.STS_CD in( :2, :3, :4, :5, :6 )) and (workitemdb0.USER_ID
in ( :7 ))
order by workitemresort0.REFER_TXT asc, WRK_ID0_ asc ) row_
where rownum <= :8)
where rownum_ > :9
/

```

Once the two outlines are created, the steps in Figure 32 can be used to associate the “good” execution plan with the “good” SQL by modifying the data in stored outline metadata tables that are owned by the OUTLN schema.

Figure 32: Example Six Updating Stored Outline Metadata

```
SQL> delete from outln.ol$hints where ol_name = 'ICS_IB_2053266053_ERASEME';

24 rows deleted.

SQL> delete from outln.ol$ where ol_name = 'ICS_IB_2053266053';

1 row deleted.

SQL> update outln.ol$ set ol_name = 'ICS_IB_2053266053'
  2 where ol_name = 'ICS_IB_2053266053_ERASEME';

1 row updated.

SQL> commit;

Commit complete.
```

The commands issued in Figure 32 leave us with a stored outline that has the original application SQL, but with the execution plan from the hinted SQL.

Note: It’s normally not recommended to make updates directly to Oracle metadata tables, but I opened a Service Request with Oracle Support on this very issue and was given the OK to make the types of updates to the OUTLN tables shown in Figure 32.

Finally, since this application has both OLTP and batch process users, a database logon trigger was created to apply the stored outline to the OLTP users only. This was done to prevent any unintended impacts to other non-OLTP application users. Figure 33 shows the database trigger that was used to accomplish this segregation.

Figure 33: Example Six Database Logon Trigger for Stored Outline

```
create or replace trigger sys.ics_stored_outln_trg
after logon on database declare
  v_user varchar2(30):=user;
begin
  if (v_user='icsuser') then
    execute immediate 'alter session set use_stored_outlines=ics_oltp';
  end if;
end ics_stored_outln_trg;
/
```

Once the stored outlines were in place, the application servers for the application were restarted to initiate new connections to the database, thus firing the database logon trigger. The application was tested and response time was seen to be much better that it had been before. However, to conclusively determine that the application was using the stored outline, two final tests were conducted. First, the DBA_OUTLINES data dictionary view was examined to determine whether the USED flag had switched from UNUSED (the default) to USED as shown in Figure 34.

Figure 34: Example Six Confirmation of Stored Outline Usage

```
SQL> select to_char(timestamp,'hh24:mi dd-MON-yyyy') "CREATED", used, category, name
2 from dba_outlines order by timestamp;
```

| CREATED | OWNER | USED | CATEGORY | NAME |
|-------------------|--------|------|----------|-------------------|
| 14:40 11-OCT-2008 | WFGEN3 | USED | ICS_OLTP | ICS_IB_2053266053 |

The output in Figure 34 indicates that the stored outline had been used at least once since its implementation. As a final confirmation of the effectiveness of the stored outline, another 10046 trace of the application was collected. Those traces are shown in Figure 35.

Figure 35: Example Six Formatted 10046 Trace Showing Stored Outline Usage

Statement Text

```
select *
from (
select row_.*, rownum rownum_
from (
select workitemdb0_.WRK_ID as WRK_IDO_, workitemdb0_.LST_UPDT_TS as
LST_UPDT2_O_, workitemdb0_.USER_ID as USER_IDO_, workitemdb0_.WRK_SRC_TXT as
WRK_SRC_4_O_, workitemdb0_.SUSP_TS as SUSP_TSO_, workitemdb0_.DUE_TS as
DUE_TSO_, workitemdb0_.CRTE_TS as CRTE_TSO_, workitemdb0_.LST_UPDT_PGM_NM as
LST_UPDT8_O_, workitemdb0_.LST_UPDT_USERID as LST_UPDT9_O_,
workitemdb0_.STS_CD as STS_CDO_, workitemdb0_.APP_ID as APP_IDO_,
workitemdb0_.WRK_TYP_CD as WRK_TYP_CDO_, workitemdb0_.WRK_QUE_ID as
WRK_QUE_IDO_, workitemdb0_.UNQ_ID as UNQ_IDO_, workitemdb0_.ASSIGN_TS as
ASSIGN_TSO_, workitemdb0_.ACTION_CD as ACTION_CDO_, workitemdb0_.RECD_TS as
RECD_TSO_, workitemdb0_.WEIGHT as WEIGHTO_, workitemdb0_.AUTO_ASSIGN_IND as
AUTO_AS19_O_, workitemdb0_.CRTE_ROW_PGM_NM as CRTE_RO20_O_,
workitemdb0_.CRTE_ROW_USERID as CRTE_RO21_O_
from WFGEN3.WRK workitemdb0_ left join WFGEN3.WRK_REFER workitemresort0 on
(workitemdb0_.WRK_ID=workitemresort0.WRK_ID and workitemresort0.REFER_KEY_CD=
:1 )
where (workitemdb0_.STS_CD in( :2, :3, :4, :5, :6 )) and (workitemdb0_.USER_ID
in ( :7 ))
order by workitemresort0.REFER_TXT asc, WRK_IDO_ desc ) row_
where rownum <= :8)
where rownum_ > :9
```

Statement Cumulative Statistics

| Call | Cache Misses | Count | - Seconds - | | Physical Reads | - Logical Reads - | | Rows |
|--------------|--------------|----------|----------------|----------------|----------------|-------------------|----------|-----------|
| | | | CPU | Elapsed | | Consistent | Current | |
| Parse | 0 | 1 | 0.0000s | 0.0001s | 0 | 0 | 0 | 0 |
| Exec | 0 | 1 | 0.0000s | 0.0007s | 0 | 0 | 0 | 0 |
| Fetch | | 3 | 1.9600s | 2.4862s | 0 | 38,047 | 0 | 75 |
| Total | 0 | 5 | 1.9600s | 2.4871s | 0 | 38,047 | 0 | 75 |
| Per Fch | 0.0 | 1.7 | 0.6533s | 0.8290s | 0.0 | 12,682.3 | 0.0 | 8.3 |
| Per Row | 0.0 | 0.2 | 0.0261s | 0.0332s | 0.0 | 507.3 | 0.0 | 1.0 |

Figure 35 Continued: Example Six Formatted 10046 Trace Showing Stored Outline Usage**Statement Plan**

met 1 time

| Rows | Row Source Operation | Object Id |
|-----------|--|-----------|
| 75 | VIEW (cr=38,047 pr=0 pw=0 time=2.4857s) | |
| 150 | COUNT STOPKEY (cr=38,047 pr=0 pw=0 time=2.4856s) | |
| 150 | VIEW (cr=38,047 pr=0 pw=0 time=2.4855s) | |
| 150 | SORT ORDER BY STOPKEY (cr=38,047 pr=0 pw=0 time=2.4853s) | |
| 311 | NESTED LOOPS OUTER (cr=38,047 pr=0 pw=0 time=2.4833s) | |
| 311 | TABLE ACCESS BY INDEX ROWID WRK (cr=162 pr=0 pw=0 time=0.0117s) | 18356 |
| 311 | INDEX RANGE SCAN WRK_COUNT_INBOX_IDX (cr=79 pr=0 pw=0 time=0.0108s) | 18722 |
| 311 | TABLE ACCESS BY INDEX ROWID WRK REFER (cr=37,885 pr=0 pw=0 time=0.0595s) | 18311 |
| 7,707,442 | INDEX RANGE SCAN WRK_REFER_X1 (cr=79 pr=0 pw=0 time=0.0108s) | 18460 |

The output in Figure 35 shows that the SQL was now performing 78% fewer total I/Os than the original query and the execution plan is now utilizing the WRK_REFER_X1 index with the desired NESTED LOOPS join operation – exactly as directed by the stored outline.

Conclusion

This paper has shown six examples of the application of *Method R* and the Hotsos tuning tools to solve real-world performance problems. The key to resolving each of these problems was clearly identifying the problem by gathering 10046 traces, determining the cause of the problem by reviewing formatted 10046 traces, output from Hotsos SQL Test Harness scripts, and other investigative work, then focusing on possible solutions for that specific issue. As examples, this paper showed how problems caused by poor row versus block selectivity, incorrect autoextend settings, table corruption, IS NULL predicates, improperly configured database connectors, and poor CBO join operations were all identified and corrected.

About the Author

Joe Johnson is an Oracle Certified Professional working as a Database Administration Specialist for American Family Insurance in Madison, Wisconsin, USA. Joe has over 15 years of Oracle DBA experience in a variety of industries, and has authored or co-authored four books and numerous articles on Oracle database administration topics. He can be contacted by emailing joe.johnson@yahoo.com.