Oracle Performance Diagnostic Guide (Version 3.20)
Query Tuning
9/11/2012

Welcome to the Oracle Performance Diagnostic Guide This guide is intended to help you resolve query tuning, hang/locking, and slow database issues. The guide is not an automated tool but rather seeks to show methodologies, techniques, common causes, and solutions to performance problems.

Most of the guide is finished but portions of the content under the Hang/Locking tab is still under development.
Your feedback is very valuable to us - please email your comments to: Vickie.Carbonneau@oracle.com

Contents

Feedback

We look forward to your feedback. Please email any comments, suggestion to help improve this guide, or any issues that you have encountered with the tool usage to Vickie.Carbonneau@oracle.com, Technical Advisor, Center of Expertise (CoE).

Query Tuning > Identify the Issue > Overview

To properly identify the issue we want to resolve, we must do three things:

- Recognize a query tuning issue
- Clarify the details surrounding the issue
- Verify that the issue is indeed the problem.

Recognize a Query Tuning Issue

What is a Query Tuning Issue?

A query tuning issue can manifest itself as:

- A particular SQL statement or group of statements that run slowly at a time when other statements run well
- One or more sessions are running slowly and most of the delay occurs during the execution of a particular SQL statement

You might have identified these queries from:

- benchmarking/testing
- user complaints
- statspack or AWR reports showing expensive SQL statements
- a query appearing to hang
- session consuming a large amount of CPU

These problems might appear after:

- schema changes
- changes in stats
- changes in data volumes
- changes in application
- database upgrades

Clarify the Issue

A clear problem statement is critical. You need to be clear on exactly what the problem is. It may be that in subsequent phases of working through the issue, the real problem becomes clearer and you have to revisit and re-clarify the issue.

To clarify the issue, you must know as much as possible of the following:

- The affected SQL statement.
- The sequence of events leading up to the problem
- Where/how was it noticed
- The significance of the problem
- What IS working
- What is the expected or acceptable result?
- What have you done to try to resolve the problem

As an example:

- A SQL statement performs poorly after re-gathering statistics.
- It was noticed by end users.
- It is making the application run slowly and preventing our system from taking orders.
- Everything else is fine and the problem did not show in our test environment.
- Normally, this statement runs in less than 2 seconds.
- We tried re-gathering stats, but it did not make any difference.

Why is this step mandatory?
Skipping this step will be risky because you might attack the wrong problem and waste significant time and effort. A clear problem statement is critical to begin finding the cause and solution to the problem.

Notes
- ODM Reference: Identify the Issue

How-To
- How to Identify Resource Intensive SQL for Tuning

Case Studies
- Resolving High CPU usage in Oracle Servers

Verify the Issue

Our objective in this step of the diagnostic process is to ensure the query that is thought to need tuning, is actually the query at the root of the performance problem. At this point, you need to collect data that verifies the existence of a problem.

To verify the existence of the issue you must collect :

- the SQL statement
- evidence of the poor performance of the query

Example:

The following query is slow:

Notes
- ODM Reference: Identify the Issue

```
SELECT order_id
 FROM po_orders
 WHERE branch_id = 6
```

Timing information was collected using SQLPlus as follows:

```
 SQL> set timing on
 SQL> SELECT order_id
      FROM po_orders
      WHERE branch_id = 6

       ORDER_ID
     ----------
            232

     Elapsed: 00:00:13.26
```

Further examples and advice on what diagnostic information will be needed to resolve the problem will be discussed in the DATA COLLECTION section.

Once the data is collected, you will review it to either verify there is a query tuning issue, or decide it is a different issue.

Why is this step mandatory?
If you skip this step, you might have identified the wrong query to tune and waste significant time and effort before you realize it. For example, the performance problem appears to be a 30 second delay to display a page; however, by verifying the issue, you see that the query you suspect to be a problem actually completes in 1 second. In this case, query tuning will not help solve this problem. Maybe, the problem lies with the network (e.g. latency or timeouts) or application server (e.g., high CPU utilization on the mid tier).

Special Considerations

- If you are on Oracle 10g or higher AND are licensed for the EM Tuning Pack, you should stop here and use the SQL Tuning Advisor to resolve your query tuning problem.

  Please see the following resources to get started with the SQL Tuning Advisor:
  10gR2 PerformanceTuning Guide, Automatic SQL Tuning
  White Paper: Optimizing the Optimizer
  Oracle 10g Manageability

- Database and application (query) tuning is an interactive process that requires a complete understanding of the environment where the database resides (database, operating system, application, etc). Standard Product Support Services provide solutions to bugs that may affect the performance of the database and its components. Support may also provide general recommendations to begin the tuning process. To perform a complete performance analysis, you must request one of the advanced services that Oracle Support has

available to tune your system. Visit http://www.oracle.com/support/assist/index.html or contact your Support Sales representative for further details on these services.

Next Step - Data Collection

When you have done the above, click "NEXT" to get some guidance on collecting data that will help to validate that you are looking at the right problem and that will help in diagnosing the cause of the problem.

Query Tuning > Identify the Issue >Data Collection

In this step, we will collect data to help verify whether the suspected query is the one that should be tuned. We should be prepared to identify the specific steps in the application that cause the slow query to execute. We will trace the database session while the application executes this query.

Note: Always try to collect data when the query ran well and when it ran poorly. Migrations and other changes sometimes cause queries to change. Having both the "good" and "bad" execution plans can help determine what might be going wrong and how to fix it.

Gather an Extended SQL Trace

The extended SQL trace (10046 trace at level 12) will capture execution statistics of all SQL statements issued by a session during the trace. It will show us how much time is being spent per statement, how much of the time was due to CPU or wait events, and what the bind values were. We will be able to verify if the "candidate" SQL statement is truly among the SQL issued by a typical session.

For detailed information on how to use the 10046 trace event, read Recommended Method for Obtaining 10046 trace for Tuning first

A summary of the steps needed to obtain the 10046 and TKProf are listed below:

Choose a session to trace

Target the most important / impacted sessions

- Users that are experiencing the problem most severely; e.g., normally the transaction is complete in 1 sec, but now it takes 30 sec.
- Users that are aggressively accumulating time in the database

The following queries will allow you to find the sessions currently logged into the database that have accumulated the most time on CPU or for certain wait events. Use them to identify potential sessions to trace using 10046.

These queries are filtering the sessions based on logon times less than 4 hours and the last call occurring within 30 minutes. This is to find more currently relevant sessions instead of long running ones that accumulate a lot of time but aren't having a performance problem. You may need to adjust these values to suit your environment.

Find Sessions with the Highest CPU Consumption

Documentation

- Understanding SQL Trace and TKProf

How-To

- How To Collect 10046 Trace Data
- Recommended Method for Obtaining 10046 trace for Tuning

SCRIPTS/TOOLS

- Trace Analyzer TRCANLZR - Interpreting Raw SQL Traces with Binds and/or Waits generated by EVENT 10046
- How To Generate TKProf reports
- Collect 10046 Traces Automatically with LTOM

```
-- sessions with highest CPU consumption
SELECT s.sid, s.serial#, p.spid as "OS PID",s.username, s.module, st.value/100 as
"CPU sec"
FROM v$sesstat st, v$statname sn, v$session s, v$process p
WHERE sn.name = 'CPU used by this session' -- CPU
AND st.statistic# = sn.statistic#
AND st.sid = s.sid
AND s.paddr = p.addr
AND s.last_call_et < 1800 -- active within last 1/2 hour
AND s.logon_time > (SYSDATE - 240/1440) -- sessions logged on within 4 hours
ORDER BY st.value;

       SID    SERIAL# OS PID       USERNAME
MODULE                                           CPU sec
---------- ---------- ------------ --------------------
---------------------------------------- ----------
       141       1125 15315        SYS                  sqlplus@coehq2 (TNS V1-
V3)                      8.25
       147        575 10577        SCOTT
SQL*Plus                                          258.08
       131        696 10578        SCOTT
SQL*Plus                                          263.17
       139        218 10576        SCOTT
SQL*Plus                                          264.08
       133        354 10583        SCOTT
SQL*Plus                                          265.79
       135        277 10586        SCOTT
SQL*Plus                                          268.02
```

Find Sessions with Highest Waits of a Certain Type

```
-- sessions with the highest time for a certain wait
SELECT s.sid, s.serial#, p.spid as "OS PID", s.username, s.module, se.time_waited
FROM v$session_event se, v$session s, v$process p
WHERE se.event = '&event_name'
AND s.last_call_et < 1800 -- active within last 1/2 hour
AND s.logon_time > (SYSDATE - 240/1440) -- sessions logged on within 4 hours
AND se.sid = s.sid
AND s.paddr = p.addr
ORDER BY se.time_waited;

SQL> /
Enter value for event_name: db file sequential read

       SID    SERIAL# OS PID       USERNAME
MODULE                                   TIME_WAITED
---------- ---------- ------------ --------------------
----------------------------------------- -----------
       141       1125 15315        SYS                  sqlplus@coehq2 (TNS V1-
V3)                           4
       147        575 10577        SCOTT
SQL*Plus                                          45215
       131        696 10578        SCOTT
SQL*Plus                                          45529
       135        277 10586        SCOTT
SQL*Plus                                          50288
       139        218 10576        SCOTT
SQL*Plus                                          51331
       133        354 10583        SCOTT
SQL*Plus                                          51428
```

10g or higher: Find Sessions with the Highest DB Time

```
-- sessions with highest DB Time usage
SELECT s.sid, s.serial#, p.spid as "OS PID", s.username, s.module, st.value/100 as
"DB Time (sec)"
, stcpu.value/100 as "CPU Time (sec)", round(stcpu.value / st.value * 100,2) as "%
CPU"
FROM v$sesstat st, v$statname sn, v$session s, v$sesstat stcpu, v$statname sncpu, v
$process p
WHERE sn.name = 'DB time' -- CPU
AND st.statistic# = sn.statistic#
AND st.sid = s.sid
AND  sncpu.name = 'CPU used by this session' -- CPU
AND stcpu.statistic# = sncpu.statistic#
AND stcpu.sid = st.sid
AND s.paddr = p.addr
AND s.last_call_et < 1800 -- active within last 1/2 hour
AND s.logon_time > (SYSDATE - 240/1440) -- sessions logged on within 4 hours
AND st.value > 0;

       SID    SERIAL# OS PID       USERNAME MODULE
DB Time (sec) CPU Time (sec)      % CPU
---------- ---------- ------------ -------- ------------------------------------
------------- -------------- ----------
       141       1125 15315        SYS      sqlplus@coehq2 (TNS V1-
V3)                      12.92           9.34      72.29


Note: sometimes DB Time can be lower than CPU Time when a session issues long-running
recursive calls. The DB Time statistic doesn't update until the top-level call is finished
(versus the CPU statistic that updates as each call completes).
```

Obtain a complete trace

- Ideally, start the trace as soon as the user logs on and begins the operation or transaction. Continue tracing until the operation is finished.
- Try to avoid starting or ending the trace in the middle of a call unless you know the call is not important to the solution

Collect the trace and generate a TKProf or TRCANLZR report

- A connected session
  - Start tracing on a connected session
  - Coordinate with the user to start the operation
  - Measure the client's response time for the operation

    The idea here is compare the time it takes to perform some function in the application from the user's perspective to the time it takes to execute the application's underlying SQL in the database for that functionality. If these two times are close, then the performance problem is in the database. Otherwise, the problem may be elsewhere.

  - Stop tracing
  - Gather the trace file from the "user_dump_dest" location (you can usually identify the file just by looking at the timestamp).

- Using a test script
  - Simply run the test script and collect the trace file from the "user_dump_dest" location (you can usually identify the file just by looking at the timestamp).

- Other Considerations
  - Shared Servers: Tracing shared servers could cause many separate trace files to be produced as the session moves to different Oracle processes during execution. Use the 10g utility, "trcsess" to combine these separate files into one.

- Generate a TRCANLZR report (see references for *Scripts and Tools*) or generate a TKProf report and sort the SQL statements in order of most elapsed time using the following command:

  tkprof <trace file name> <output file name> sort=fchela,exeela,prsela

Make sure trace file contains only data from the recent test

- If this session has been traced recently, there may be other traces mixed in the file with the recent trace collected
- We should extract only the trace data that is part of the recent tests. See the place in the sample trace below where it says "Cut away lines above this point".

Trace file from a long running process that has been traced intermittently over several days

```
. . .
*** 2006-07-24 13:35:05.642 <== Timestamp from a previous tracing
WAIT #8: nam='SQL*Net message from client' ela= 20479935 p1=1650815232 p2=1 p3=0
=====================
PARSING IN CURSOR #9 len=43 dep=0 uid=57 oct=3 lid=57 tim=1007742062095 hv=4018512766 ad='97039a58'
select e.empno, d.deptno<== Previous cursor that was traced
from emp e, dept d
END OF STMT
PARSE #9:c=630000,e=864645,p=10,cr=174,cu=0,mis=1,r=0,dep=0,og=4,tim=1007742062058
BINDS #9:
EXEC #9:c=0,e=329,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1007742062997
WAIT #9: nam='SQL*Net message to client' ela= 18 p1=1650815232 p2=1 p3=0
. . .
FETCH #9:c=10000,e=513,p=0,cr=1,cu=0,mis=0,r=15,dep=0,og=4,tim=1007742148898
WAIT #9: nam='SQL*Net message from client' ela= 2450 p1=1650815232 p2=1 p3=0
```

```
WAIT #9: nam='SQL*Net message to client' ela= 7 p1=1650815232 p2=1 p3=0
FETCH #9:c=0,e=233,p=0,cr=0,cu=0,mis=0,r=10,dep=0,og=4,tim=1007742152065
. . .

====> CUT AWAY LINES ABOVE THIS POINT - THEY AREN'T PART OF THIS TEST <====
*** 2006-07-24 18:35:48.850
<== Timestamp for the tracing we want (notice its about 5 hours later)


=====================
PARSING IN CURSOR #10 len=69 dep=0 uid=57 oct=42 lid=57 tim=1007783391548 hv=3164292706 ad='9915de10'
alter session set events '10046 trace name context forever, level 12'
END OF STMT
. . .
=====================
PARSING IN CURSOR #3 len=68 dep=0 uid=57 oct=3 lid=57 tim=1007831212596 hv=1036028368 ad='9306bee0'
select e.empno, d.dname<== Cursor that was traced
from emp e, dept d
where e.deptno = d.deptno
END OF STMT
PARSE #3:c=20000,e=17200,p=0,cr=6,cu=0,mis=1,r=0,dep=0,og=4,tim=1007831212566
BINDS #3:
EXEC #3:c=0,e=321,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1007831213512
WAIT #3: nam='SQL*Net message to client' ela= 15 p1=1650815232 p2=1 p3=0
WAIT #3: nam='db file sequential read' ela= 7126 p1=4 p2=11 p3=1
. . .
FETCH #3:c=10000,e=39451,p=12,cr=14,cu=0,mis=0,r=1,dep=0,og=4,tim=1007831253359
WAIT #3: nam='SQL*Net message from client' ela= 2009 p1=1650815232 p2=1 p3=0
WAIT #3: nam='SQL*Net message to client' ela= 10 p1=1650815232 p2=1 p3=0
FETCH #3:c=0,e=654,p=0,cr=1,cu=0,mis=0,r=13,dep=0,og=4,tim=1007831256674
WAIT #3: nam='SQL*Net message from client' ela= 13030644 p1=1650815232 p2=1 p3=0
STAT #3 id=1 cnt=14 pid=0 pos=1 obj=0 op='HASH JOIN (cr=15 pr=12 pw=0 time=39402 us)'
=====================
PARSING IN CURSOR #7 len=55 dep=0 uid=57 oct=42 lid=57 tim=1007844294588 hv=2217940283 ad='95037918'
alter session set events '10046 trace name context off' <== tracing turned off
END OF STMT
```

Make sure the trace is complete

- If the trace started or ended during a call, its best to rethink how the trace is started to ensure this doesn't happen. You can get an idea for the the amount of time attributed to the call that was in progress at the beginning or end of the trace by looking at the timestamps to find the total time spent prior to the first call and comparing it to the call's elapsed time (although if there were other fetch calls before the first one in the trace, you'll miss those) . The following trace file excerpt was taken by turning on the trace after the query had been executing for a few minutes.

```
*** 2006-07-24 15:00:45.538 <== Time when the trace was started
WAIT #3: nam='db file scattered read' ela= 18598 p1=4 p2=69417 p3=8 <== Wait
*** 2006-07-24 15:01:16.849 <== 10g will print timestamps if trace hasn't been written to in a while
WAIT #3: nam='db file scattered read' ela= 20793 p1=4 p2=126722 p3=7
. . .
*** 2006-07-24 15:27:46.076
WAIT #3: nam='db file sequential read' ela= 226 p1=4 p2=127625 p3=1 <== Yet more waits
WAIT #3: nam='db file sequential read' ela= 102 p1=4 p2=45346 p3=1
WAIT #3: nam='db file sequential read' ela= 127 p1=4 p2=127626 p3=1
WAIT #3: nam='db file scattered read' ela= 2084 p1=4 p2=127627 p3=16
. . .
*** 2006-07-24 15:30:28.536 <== Final timestamp before end of FETCH call
WAIT #3: nam='db file scattered read' ela= 5218 p1=4 p2=127705 p3=16 <== Final wait
WAIT #3: nam='SQL*Net message from client' ela= 1100 p1=1650815232 p2=1 p3=0
=====================
PARSING IN CURSOR #3 len=39 dep=0 uid=57 oct=0 lid=57 tim=1014506207489 hv=1173176699 ad='931230c8'
select count(*) from big_tab1, big_tab2 <== This is not a real parse call, just printed for convenience
END OF STMT

FETCH #3:c=0,e=11,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,tim=1014506207466 <== Completion of FETCH call
Notice the FETCH reports 11 microSec elapsed. This is wrong as you can see from timestamps -
It should be around 30 minutes. Maybe this is a feature?
```

Check if most of the elapsed time is spent waiting ***between*** calls

Waits for "SQL*Net Message from Client" between calls (usually FETCH calls) indicate a performance problem with the client (slow client or not using array operations) or network (high latencies, low bandwidth, or timeouts). Query tuning will not solve these kinds of problems. Evidence of waits *between* calls can be spotted by looking at the following:

1) In the TKProf, you will notice the total time spent in the database is small compared to the time waited by the client. You will also see the bulk of the time in "SQL*Net message from client" in the waits section, as shown below:

TKProf of a session where the client used an arraysize of 2 and caused many fetch calls

```
select empno, ename from  emp

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        8      0.00       0.00          0         14          0          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       10      0.00       0.00          0         14          0          14


Rows     Row Source Operation
-------  ---------------------------------------------------
     14  TABLE ACCESS FULL EMP (cr=14 pr=0 pw=0 time=377 us)


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                        8        0.00          0.00
  SQL*Net message from client                      8       29.36         78.39
```

Notice above: 8 fetch calls to return 14 rows. 78.39 seconds waiting for "SQL*Net message from client" for 8 waits. Each wait corresponds to each fetch call. The total database time was 377 microSeconds, but the total elapsed time to fetch all 14 rows was 78.39 seconds due to client waits. If you reduce the number of fetches, you will reduce the overall elapsed time. In any case, the database is fine, the problem is really external to the database.

2) To confirm whether the waits are due to a slow client, examine the 10046 trace for the SQL statement and look for WAITs in between FETCH calls, as follows:

```
PARSING IN CURSOR #2 len=29 dep=0 uid=57 oct=3 lid=57 tim=1016349402066 hv=3058029015 ad='94239ec0'
select empno, ename from emp
END OF STMT
PARSE #2:c=0,e=5797,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=1016349402036
EXEC #2:c=0,e=213,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1016349402675
WAIT #2: nam='SQL*Net message to client' ela= 12 p1=1650815232 p2=1 p3=0
FETCH #2:c=0,e=423,p=0,cr=7,cu=0,mis=0,r=1,dep=0,og=4,tim=1016349403494 <== Call Finished
WAIT #2: nam='SQL*Net message from client' ela= 1103179 p1=1650815232 p2=1 p3=0 <== Wait for client
WAIT #2: nam='SQL*Net message to client' ela= 10 p1=1650815232 p2=1 p3=0
FETCH #2:c=0,e=330,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1016350507608 <== Call Finished (2 rows)
WAIT #2: nam='SQL*Net message from client' ela= 29367263 p1=1650815232 p2=1 p3=0 <== Wait for client
WAIT #2: nam='SQL*Net message to client' ela= 9 p1=1650815232 p2=1 p3=0
FETCH #2:c=0,e=321,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1016379876558 <== Call Finished (2 rows)
WAIT #2: nam='SQL*Net message from client' ela= 11256970 p1=1650815232 p2=1 p3=0 <== Wait for client
WAIT #2: nam='SQL*Net message to client' ela= 10 p1=1650815232 p2=1 p3=0
. . .
FETCH #2:c=0,e=486,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=4,tim=1016409054527
WAIT #2: nam='SQL*Net message from client' ela= 18747616 p1=1650815232 p2=1 p3=0
STAT #2 id=1 cnt=14 pid=0 pos=1 obj=49049 op='TABLE ACCESS FULL EMP (cr=14 pr=0 pw=0 time=377 us)'
```

Notice: Between each FETCH call, there is a wait for the client. The client is slow and responds every 1 - 2 seconds.

If it appears that most waits occur in between calls for the *SQL*Net message from client* event, proceed to the "Slow Database" tab and navigate to this section for help in diagnosing these waits: Slow Database > Determine a Cause >Analysis > Choose a Tuning Strategy > Reduce Client Bottlenecks

Next Step - Analyze

When you have collected the data, click "NEXT" to receive guidance on analyzing the data and verify whether or not the suspected query is indeed the one to tune.

This step will analyze the trace file and TKProf data collected in the previous step to verify the suspected query is actually the one that should be tuned.

Verify the Problem Query using TKProf or Trace Analyzer (TRCANLZR)

At this stage, we will verify the suspected problem query using TKProf or TRCANLZR (recommended). TKProf and TRCANLZR will summarize the output of the SQL trace file to show us how much time each SQL statement took to run, the runtime execution plan (if the cursor was closed), and the waits associated with each statement (TRCANLZR will supply more detail and account for the time completely). We can quickly see the statements that were responsible for most of the time and hence should be considered for tuning.

If we see the that the top SQL statement in TKProf is the same one we suspect needs tuning, then we have verified the issue.

Data Required for Verification:

- TKProf or TRCANLZR output from the application that was traced in the previous step, *"Data Collection"*
- Measurement of the elapsed time to run the application from a user's point of view

Verification Steps:

1. ***Does total elapsed time in TKProf/TRCANLZR account for the application response time that was measured when the application was executed?***

If so, continue to the next question.

If not:

- ***Was the wrong session traced?***

Detailed Explanation
For example, if the application ran in 410 seconds, look in the "Overall Totals" section at the bottom of the TKProf to see what the total trace elapsed time was (assuming the trace file was started just before the application executed and was stopped just after the execution finished):

Documentation

- Understanding SQL Trace and TKProf

Special Topics

- Applications: How to use TKProf and Trace with Applications

Scripts and Tools

- Trace Analyzer TRCANLZR - Interpreting Raw SQL Traces with Binds and/or Waits generated by EVENT 10046
- Implementing and Using the PL/SQL Profiler
- Tracing PX session with a 10046 event or sql_trace

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu  elapsed  disk    query  current    rows
------- ------  -------- -------- ----- -------- --------  ------
Parse     1165      0.66     2.15     0        45        0       0
Execute   2926      1.23     2.92     0         0        0       0
Fetch     2945    117.03   398.23  5548   1699259       16   39654
------- ------  -------- -------- ----- -------- --------  ------
total     7036    118.92   403.31  5548   1699304       16   39654


OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

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

The total database time captured in this trace file was:
Total Non Recursive Time + Total Recursive Time = 403.31 sec
```

In this case, 403 seconds out of 410 seconds seen from the users point of view was spent in the database. Query tuning will indeed help this situation.

2. ***Does the time spent parsing, executing, and fetching account for most of the elapsed time in the trace.***

If so, continue to the next question.

If not, check client waits ("SQLNet Message from Client") time between calls

- ***Are the client waits occurring in between fetch calls for the same cursor ?***
  - ***If so, update the problem statement to note this fact and continue with the next question.***
  - ***If most of the time is spent waiting in between calls for different cursors, the bottleneck is in the client tier or network - SQL tuning may not improve the performance of the application.***
    This is no longer a query tuning issue but requires analysis of the client or network.

Detailed Explanation
The goal of query tuning is to reduce the amount of time a query takes to parse, execute, and/or fetch data. If the trace file shows that these operations occur quickly relative to the total elapsed time, then we may actually need to tune the client or network.

When the database is spending most of the time idle between executions of cursors, we suspect that a client or network is slow.

On the other hand, when most of the query's elapsed time is idle time between fetches of the same cursor, we suspect that the client is not utilizing bulk (array) fetches (we may see similar waits between executions of the same cursor when bulk inserts or updates aren't used).

The result of this is that it would be futile to tune a query that is actually spending most of its time outside of the database; we must know this before we start tuning the query.

3. ***Is the query we expect to tune shown at the top of the TKProf/TRCANLZR report?***

If so, continue to the next question.

If not:

- ***Was the SQL reported in TKProf/TRCANLZR as the highest elapsed time a PL/SQL procedure?***
    - ***Skip down the file until the first non-PL/SQL query is seen. If this query is the suspected query, then continue with the next question.***
    - ***Otherwise, the problem statement needs to change to either identify the PL/SQL or the first non-PL/SQL query found in the trace file. After updating the problem statement, continue with the next question.***
- ***Was the wrong session traced?***
- ***Was the session traced properly (started trace too late or finished too early)***

Do not continue until you review your data collection procedures to ensure you are collecting the data properly.

4. ***Does the query spend most of its time in the execute/fetch phases (not parse phase)?***

If so, you are done verifying that this query is the one that should be tuned.

If not, there may be a parsing problem that needs to be investigated. Normal query tuning techniques that alter the execution plan probably won't help. Update the problem statement to point out that we are aiming to improve the parse time. Proceed to investigate possible causes for this in this section of the guide:

Query Tuning > Determine a Cause >Analysis > Choose a Tuning Strategy > Parse Time Reduction Strategy

For example:

```
SELECT * FROM  ct_dn dn, ds_attrstore store

call     count       cpu  elapsed   disk    query  current     rows
------- ------  -------- -------- ------ -------- --------  -------
Parse      555    100.09   300.83      0        0        0        0
Execute    555      0.42     0.78      0        0        0        0
Fetch      555     14.04    85.03    513  1448514        0    11724
------- ------  -------- -------- ------ -------- --------  -------
total     1665    114.55   386.65    513  1448514        0    11724
```

The elapsed time spent parsing was 300.83 seconds compared to only 85.03 seconds for fetching. This query is having trouble parsing - tuning the query's execution plan will not give the greatest performance gain.

Next Step - Determine a Cause

If the analysis above has confirmed the query you want to tune, click "NEXT" to move to the next phase of this process where you will receive guidance to determine a cause for the slow query.

Would You Like to Stop and Log a Service Request?

We would encourage you to continue until at least the "Determine a Cause", "Data Collection" step, but If you would like to stop at this point and receive assistance from Oracle Support Services, please do the following:

- In the SR Creation Template, Question "Last Diagnostic Step Completed?", Please copy and paste the following:

  Last Diagnostic Step = Performance_Diagnostic_Guide.QTune.Issue_Identification.
  Data_Collection

- Enter the problem statement and how the issue has been verified (if performed)
- Gather the 10046 trace you collected and prepare to upload it to the service request
- Optionally, gather an RDA
- Gather other relevant information you may have such as explain plan output

The more data you collect ahead of time and upload to Oracle, the fewer round trips will be required for this data and the quicker the problem will be resolved.

Click here to log your service request

Query Tuning > Determine a Cause >Overview

At this point we have verified that an individual query needs to be tuned; now, we seek to determine the cause for this query's bad execution plan. To identify the specific cause we will need to collect data about the execution plan, the runtime statistics, and the underlying objects referenced in the query. Our approach to finding the cause will be:

1. Check the basics
   - Ensure the correct optimizer is used
   - Up-to-date statistics are collected for all objects
   - Basic parameter settings are appropriate
2. Choose a tuning strategy
   - Oracle 10g or higher: Use the SQL Tuning Assistant
   - High Parse Time: Resolve the high parse time
   - Bad and Good plan exist: Compare the "good" and "bad" execution plans to find the cause and apply fix
   - Only bad plan exists, thorough analysis desired: Analyze the plan to find the cause and apply fix
   - Only bad plan exists, fast resolution desired: Use triage methods to find a good plan quickly
3. Follow the steps in the tuning strategy to identify causes and their potential solutions
4. Choose a solution and implement it.
5. Verify that the solution solved the problem or more work is needed

Its very important to remember that every cause that is identified should be justified by the facts we have collected. If a cause cannot be justified, it should not be identified as a cause (i.e., we are not trying to *guess* at a solution).

This phase is very critical to resolving the query performance problem because accurate data about the query's execution plan and underlying objects are essential for us to determine a cause for the slow performance.

Gather the Query's Execution Plan [Mandatory]

An accurate execution plan is key for starting the query tuning process. The process of obtaining an execution plan varies depending on the database version, see the details below.

If you are using SQLT (see below) then this step may be omitted.

Reference
• Recommended Methods for Obtaining a Formatted Explain Plan • 10.2 Docs: DBMS_XPLAN • 10.1 Docs: DBMS_XPLAN • 9.2 Docs: DBMS_XPLAN

Scripts and Tools
• Script to Obtain an Execution Plan from V$SQL_PLAN

Prerequisites

- Create a plan table
  Use the utlxplan.sql script to create the table as instructed below.

  SQL> @?/rdbms/admin/utlxplan

  Note that the plan table format can change between versions so ensure that you create it using the utlxplan script from the current version.

- 10g and higher: Grant Privileges
  To use the DBMS_XPLAN.DISPLAY_CURSOR functionality, the calling user must have SELECT privilege on V_$SESSION, V_$SQL_PLAN_STATISTICS_ALL, V_$SQL, and V_$SQL_PLAN, otherwise it will show an appropriate error message.

Database Version 8.1.7

a. Generate the execution plan:
   SQL> EXPLAIN PLAN FOR
   < your query goes here >

b. Display the execution plan:

   #### Serial Plans
   To obtain a formatted execution plan for serial plans:

   SQL> set lines 130
   SQL> set head off
   SQL> spool
   SQL> @?/rdbms/admin/utlxpls
   SQL> spool off

   #### Parallel Plans
   To obtain a formatted execution plan for parallel plans:

```
SQL> set lines 130
SQL> set head off
SQL> spool
SQL> @?/rdbms/admin/utlxplp
SQL> spool off
```

Database Version 9.0.x

The actual value of bind variables will influence the execution plan that is generated (due to "bind peeking"); its very important to obtain the actual execution plan for the query that is having a performance problem. Use the appropriate method below.

1. ***Preferred Approach***

This approach gathers the actual execution plan (not the EXPLAINed one); one must use one of these methods if the query has bind variables (e.g., :b1 or :custID) in order to get accurate execution plans.

- ***For a complete example on how to set bind values and statistics level to run a query, see the section below entitled, "Construct a Test Script"***
  .
- ***If the SQL has been executed, and you know the hash value of the SQL, you can pull the plan from V$SQL_PLAN_STATISTICS or V$SQL_PLAN (if statistics_level = typical) as described in Note 260942.1 .***

2. ***Alternate Approach***

Use this approach if you are unable to capture the plan using the preferred approach. This approach may be used to collect plans reliably from queries that don't have bind variables.

a. ***Generate the execution plan:***
   SQL> EXPLAIN PLAN FOR
   < your query goes here >

b. ***Display the execution plan:***

   Serial Plans
   To obtain a formatted execution plan for serial plans:

   ```
   SQL> set lines 130
   SQL> set head off
   SQL> spool
   SQL> @?/rdbms/admin/utlxpls
   SQL> spool off
   ```

   Parallel Plans

To obtain a formatted execution plan for parallel plans:

SQL> set lines 130
SQL> set head off
SQL> spool
SQL> @?/rdbms/admin/utlxplp
SQL> spool off

Database Version 9.2.x

The actual value of bind variables will influence the execution plan that is generated (due to "bind peeking"); its very important to obtain the actual execution plan for the query that is having a performance problem. Use the appropriate method below.

1. ***Preferred Approach***

This approach gathers the actual execution plan (not the EXPLAINed one) and will provide extremely useful information on actual and estimated row counts.

One must use one of these methods if the query has bind variables (e.g., :b1 or :custID) in order to get accurate execution plans.

- ***If possible, execute the query while the parameter, "STATISTICS_LEVEL" is set to ALL in a session. Warning: Do not set this for the entire instance!***
  For a complete example on how to set bind values and statistics level to run a query, see the section below entitled, "Construct a Test Script"
  .
- ***If the SQL has been executed, and you know the hash value of the SQL, you can pull the plan from V$SQL_PLAN_STATISTICS or V$SQL_PLAN (if statistics_level = typical) as described in Note 260942.1 . .***

2. ***Alternate Approach***

Use this approach if you are unable to capture the plan using the preferred approach. This approach may be used to collect plans reliably from queries that don't have bind variables.

a. ***Generate the execution plan:***
   SQL> EXPLAIN PLAN FOR
   < your query goes here >

b. ***Display the execution plan:***
   SQL> set lines 130
   SQL> set head off
   SQL> spool myfile.lst
   SQL> alter session set cursor_sharing=EXACT;
   SQL> select plan_table_output from table(dbms_xplan.display('PLAN_TABLE',null,'ALL'));

SQL> spool off

Database Version 10.1.x

The actual value of bind variables will influence the execution plan that is generated (due to "bind peeking"); its very important to obtain the actual execution plan for the query that is having a performance problem. Use the appropriate method below.

1. ***Preferred Approach***

This approach uses DBMS_XPLAN to gather the actual execution plan (not the EXPLAINed one) and will provide extremely useful information on actual and estimated row counts.

One must use one of these methods if the query has bind variables (e.g., :b1 or :custID) in order to get accurate execution plans.

- ***If possible, execute the query while the parameter, "STATISTICS_LEVEL = ALL" is set for your session.***
  Warning: Do not set this for the entire instance!

  e.g:
  a. ***Execute the Query and gather plan statistics:***
     SQL> alter session set statistics_level = all;
     SQL> select col1, col2 etc.....

  b. ***Display the execution plan with plan statistics (for last executed cursor):***
     SQL> set linesize 150
     SQL> set pagesize 2000
     SQL> select * from TABLE(dbms_xplan.display_cursor('NULL, NULL , 'RUNSTATS_LAST'))

- ***To get the plan of the last executed SQL issue the following:***

  SQL> set linesize 150
  SQL> set pagesize 2000
  SQL> select * from table(dbms_xplan.display_cursor(null,null, 'ALL'));

- ***If the SQL has been executed, and you know the SQL_ID value of the SQL, you can pull the plan from from the library cache as shown:***

  SQL> set linesize 150
  SQL> set pagesize 2000
  SQL> select * from TABLE(dbms_xplan.display_cursor('&SQL_ID', &CHILD,'ALL'));

  If the cursor happened to be executed when plan statistics were gathered, then use "RUNSTATS_LAST" instead of just "ALL".

  sql_id:

specifies the sql_id value for a specific SQL statement, as shown in V$SQL.SQL_ID, V$SESSION.SQL_ID, or V$SESSION.PREV_SQL_ID. If no sql_id is specified, the last executed statement of the current session is shown.
cursor_child_no:
specifies the child number for a specific sql cursor, as shown in V$SQL.CHILD_NUMBER or in V$SESSION.SQL_CHILD_NUMBER, V$SESSION.PREV_CHILD_NUMBER.

- ***For a complete example on how to set bind values and statistics level to run a query, see the section below entitled, "Construct a Test Script"***
  .

2. ***Alternate Approach***

Alternate Approach
Use this approach if you are unable to capture the plan using the preferred approach. This approach may be used to collect plans reliably from queries that don't have bind variables.

a. ***Generate the execution plan:***
   SQL> EXPLAIN PLAN FOR
   < your query goes here >

b. ***Display the execution plan:***
   SQL> set lines 130
   SQL> set head off
   SQL> spool
   SQL> alter session set cursor_sharing=EXACT;
   SQL> select plan_table_output from table(dbms_xplan.display('PLAN_TABLE',null,'ALL'));
   SQL> spool off

Database Version 10.2.x

The actual value of bind variables will influence the execution plan that is generated (due to "bind peeking"); its very important to obtain the actual execution plan for the query that is having a performance problem. Use the appropriate method below.

1. ***Preferred Approach***

This approach uses DBMS_XPLAN to gather the actual execution plan (not the EXPLAINed one) and will provide extremely useful information on actual and estimated row counts.

One must use one of these methods if the query has bind variables (e.g., :b1 or :custID) in order to get accurate execution plans.

- ***If possible, execute the query with the hint, " gather_plan_statistics" to capture runtime statistics. Or, use the parameter, "STATISTICS_LEVEL = ALL" for your session.***
  Warning: Do not set this for the entire instance!

  e.g:
  a. ***Execute the Query and gather plan statistics:***
     SQL> select /*+ gather_plan_statistics */ col1, col2 etc.....

  b. ***Display the execution plan with plan statistics (for last executed cursor):***
     SQL> set linesize 150
     SQL> set pagesize 2000
     SQL> select * from TABLE(dbms_xplan.display_cursor('NULL, NULL , 'ALLSTATS LAST'))

- ***To get the plan of the last executed SQL issue the following:***

  SQL> set linesize 150
  SQL> set pagesize 2000
  SQL> select * from table(dbms_xplan.display_cursor(null,null, 'ALL'));

- ***If the SQL has been executed, and you know the SQL_ID value of the SQL, you can pull the plan from from the library cache as shown:***

  SQL> set linesize 150
  SQL> set pagesize 2000
  SQL> select * from TABLE(dbms_xplan.display_cursor('&SQL_ID', &CHILD,'ALL'));

  If the cursor happened to be executed when plan statistics were gathered, then use "ALL ALLSTATS" instead of just "ALL".

  sql_id:
  specifies the sql_id value for a specific SQL statement, as shown in V$SQL.SQL_ID, V $SESSION.SQL_ID, or V$SESSION.PREV_SQL_ID. If no sql_id is specified, the last executed statement of the current session is shown.
  cursor_child_no:
  specifies the child number for a specific sql cursor, as shown in V$SQL.CHILD_NUMBER or in V$SESSION.SQL_CHILD_NUMBER, V$SESSION.PREV_CHILD_NUMBER.

- ***For a complete example on how to set bind values and statistics level to run a query, see the section below entitled, "Construct a Test Script"***
  .

2. ***Alternate Approach***

Use this approach if you are unable to capture the plan using the preferred approach. This approach may be used to collect plans reliably from queries that don't have bind variables.

a. ***Generate the execution plan:***
   SQL> EXPLAIN PLAN FOR
   < your query goes here >

b. ***Display the execution plan:***
   SQL> set lines 130
   SQL> set head off
   SQL> spool
   SQL> alter session set cursor_sharing=EXACT;
   SQL> select plan_table_output from table(dbms_xplan.display('PLAN_TABLE',null,'ALL'));
   SQL> spool off

Important: Obtain Plans for Good and Bad Performing Queries

It is extremely helpful to collect as much of the data in this section as possible when the query performs poorly and when it performs well. For example, if the database is migrating from 9.2 to 10g and both systems are available, obtain the execution plans from both systems to compare a good plan to a bad plan.

If the old system is no longer available, you may need to do the following to get the old, good plan:

- use the parameter *optimizer_features_enable* = to revert the optimizer's behavior to the older one
- Import the old statistics or set them to match the other system
- Ensure the optimizer mode is set to the old system (e.g., if migrating from the rule based optimizer, then set: *optimizer_mode = rule*

Gather Comprehensive Information about the Query (Mandatory)

SQLT (SQLTXPLAIN) gathers comprehensive data about a particular query. This data can be used to examine a query's underlying objects, view the query's execution plan and dig deeper into the causes for the optimizer's execution plan decisions.

Scripts and Tools
• Downloading and Installing SQLT

1. ***Install SQLT and use the SQLTXecute, SQLTXTract, or SQLTxplain options***

SQLT requires a schema in the database where the query to be tuned is executed. This schema will be used for the tables used by SQLT. The installation needs to be done only once. The SQLXecute option is recommended because it gathers a more complete set of diagnostics. SQLTXtract can be used when a statement is not easily placed into a file and/or it has already executed and is in the library cache. SQLTXplain is more limited and is therefore the last resort.

For detailed installation instructions, please see the "instructions.txt" file in the distribution ZIP file for SQLT (click on the reference provided to download). In summary, this is how to install it:

Uncompress sqlt.zip file into a dedicated directory in the server, and run SQL*Plus from that directory connecting as a user with SYSDBA privilege.

e.g. Start SQL*Plus, then:
SQL> connect / as sysdba
SQL> @sqcreate.sql

Note:

- ***If this query contains bind values and the database is at version 9.0.x or higher, its possible that the plan collected by SQLTXplain is NOT a typical one due to bind peeking. This is one of the reasons we recommend using SQLTXecute or SQLTXtract; however if there is no other choice, SQLXplain still gathers valuable diagnostics and should be used.***

2. ***Run SQLT against the query that needs to be tuned***

- ***This will collect information about each table or view in the query, including statistics/ histograms gathered, columns, and view dependencies***
- ***The execution plan and predicate information will be gathered***
- ***If SQLTXecute is used, additional runtime statistics will be gathered including actual row counts per plan step.***
- ***AWR SQL history***
- ***A CBO (event 10053) trace will be gathered***
- ***The final output will be produced as an HTML file***
- ***Example usage:***

sqlplus <usr>/<pwd>
start sqltxplain.sql <name of text file containing one SQL statement to be analyzed>;

e.g.,

sqlplus apps/apps;
start sqltxplain.sql sql5.txt;

3. ***Gather the resulting trace file***

Is this step optional?
If you do not use SQLT you will be missing a lot of detail about the tables in the query and the 10053 trace. The *analysis* step that follows will refer to this data often; by using SQLT, you will gather the data upfront rather than piecemeal (with some inital, minimal effort installing the SQLT tables).

Gather Historical Information about the Query

SPREPSQL.SQL and AWRSQLRPT.SQL gather historical costs, elapsed times, statistics, and execution plans about a specific query. They can help identify when an execution plan changed and what a better performing execution plan looked like.

SCRIPTS/TOOLS
- Using Statspack to Report Execution Plans

NOTE 1: You may skip this entire step if you used SQLT since SQLT gathers this data for you.

NOTE 2: A prerequisite to using SPREPSQL.SQL is to ensure that Statspack snapshots are being collected at level 6 or greater (AWR will automatically collect the data). See the reference document on the right.

1. ***Find the "hash value" or "SQL ID" for the query***

- ***One way to find the hash value or SQL ID is to look for the query in the Statspack or AWR output under one of the "Top SQL" sections.***
- ***Another way is to look in the raw 10046 trace file collected from a session during the "issue verification" phase, find the SQL statement and look for the line associated with that statement containing "hv=". For example,***

```
PARSING IN CURSOR #2 len=86 dep=0 uid=54 oct=3 lid=54 tim=1010213476218 hv=710622186
ad='9d3ad468'
select TO_CHAR(hiredate,:dfmt)
from emp
where sal > :salary
and deptno = :b3
END OF STMT
```

The hash value is found in the listing above is: 710622186.

2. ***Run sqrepsql.sql or awrsqlrpt.sql to look for a point in time when the query might have performed well***

- ***When prompted for a begin and end snapshot time, enter a time when you knew the query performed poorly; it may also help to collect another report when you knew the query performed well.***
- ***For example, using sprepsql.sql :***

```
sqlplus perfstat/pwd;
@?/rdbms/admin/sprepsql.sql


Completed Snapshots Snap Snap
Instance  DB Name   Id    Snap Started      Level Comment
--------- --------- ----- ---------------- ----- --------------
DB9iR2    DB9IR2    125   18 Aug 2005 21:49     5
. . .
150 03 Apr 2006 16:51 7
151 03 Apr 2006 16:51 7


Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for begin_snap: 150
Begin Snapshot Id specified: 150


Enter value for end_snap: 151
End Snapshot Id specified: 151


Specify the Hash Value
~~~~~~~~~~~~~~~~~~~~~~
Enter value for hash_value: 710622186
Hash Value specified is: 710622186


Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~
The default report file name is sp_150_151_710622186. To use this name,
press <return> to continue, otherwise enter an alternative.
Enter value for report_name:
```

3. ***Gather the resulting trace file***

Is this step optional?
It is optional, but there is a huge potential benefit if you find an older, better execution plan stored in the repository. With the better plan, you can compare it to the current bad plan and focus on what has changed and how to fix it.

Construct a Test Script

In this step a test script will be created that can be used to run the query with any required bind values and diagnostic events turned on and off. The test script will be valuable as a benchmark while we make changes to the query. At this point in the process, we just want to create the script and make sure it represents the same performance behavior and execution plan as the original query from the application. Please note that the test script is not the same thing as a *test case* that is submitted to Oracle. We are NOT trying to reproduce this at Oracle; we want to run the test script on the actual system where the performance problem is occurring.

1. ***Extract a test case from the extended SQL trace collected in the Issue Verification phase (where the query was verified to be the biggest bottleneck).***

- ***Look for the query of interest, pay close attention to the cursor number***
- ***Find the bind values for your query, for example:***

```
PARSING IN CURSOR #1 len=90 dep=0 uid=54 oct=3 lid=54 tim=1004080714263 hv=710622186 ad='9f040c28'
select TO_CHAR(hiredate,:dfmt) <-------------- bind variable in position 0
from emp
where sal >:salary <------------------------- bind variable in position 1
and deptno = :b3 <--------------------------- bind variable in position 2
END OF STMT
PARSE #1:c=10000,e=2506,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=1004080714232
BINDS #1: <---------------------------------- IMPORTANT!, Must be the same # as the cursor above
(#1)
kkscoacd
Bind#0 <------------------------------------- section for bind position 0
oacdty=01 mxl=32(20) mxlc=00 mal=00 scl=00 pre=00
oacflg=03 fl2=1000000 frm=01 csi=31 siz=80 off=0
kxsbbbfp=ffffffff7b12bef0 bln=32 avl=10 flg=05
value="mm-dd-yyyy" <------------------------- bind value for ":dfmt" variable
Bind#1 <------------------------------------- section for bind position 1
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=03 fl2=1000000 frm=00 csi=00 siz=0 off=32
kxsbbbfp=ffffffff7b12bf10 bln=22 avl=02 flg=01
value=10 <----------------------------------- bind value for ":salary" variable
Bind#2 <------------------------------------- section for bind position 2
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=03 fl2=1000000 frm=00 csi=00 siz=0 off=56
kxsbbbfp=ffffffff7b12bf28 bln=22 avl=02 flg=01
value=20 <----------------------------------- bind value for ":b3" variable
```

- ***Determine the bind variable types and values. Referring to the example above, we can associate the bind variables, definitions, and values as such:***

Bind Variable Name in Query	Bind ID in the Trace file	Bind datatype info in the trace	Datatype	Bind value	Bind variable declaration in SQLPlus
:dfmt	Bind #0	oacdty=01 mxl=32(20) --> varchar2, length of 32	varchar2(32)	mm-dd-yyyy	variable dfmt varchar2(32)
:salary	Bind #1	oacdty=02 mxl=22(22) --> number, length not important	number	10	variable salary number

:b3	Bind #2	oacdty=02 mxl=22(22) --> number, length not important	number	20	variable b3 number

- *Create a test script that incorporates the query with the bind values, for example:*

```
set time on
set timing on

spool test_script

-- define the variables in SQLPlus
variable dfmt varchar2(32)
variable salary number
variable b3 number

-- Set the bind values
begin :dfmt := 'mm-dd-yyyy'; :salary := 10; :b3 := 20;
end;
/

-- Set statistics level to high
alter session set statistics_level = all;

-- Turn on the trace
alter session set events '10046 trace name context forever, level 12';

-- Run the query
select TO_CHAR(hiredate,:dfmt)
from emp
where sal > :salary
and deptno = :b3;

select 'end of script' from dual;

-- Turn off the trace
alter session set events '10046 trace name context off';

-- 10g: uncomment the following to obtain an execution plan

-- select * from table(dbms_xplan.display_cursor(NULL,NULL,'RUNSTATS_LAST'));

-- Reduce statistics level
alter session set statistics_level = typical;

spool off
```

Gather the resulting spool file called "test_script.lst".

2. ***Run the test script and gather the extended SQL trace that was produced in the user_dump_dest directory.***

For example if the test script similar to the one above was named "test.sql", do the following to run it and gather the resulting trace:

```
sqlplus scott/tiger @test.sql

SQL> show parameter user_dump_dest

NAME            TYPE    VALUE
--------------- ------- ----------------------------------------------------
user_dump_dest  string  /u01/app/oracle/product/DB10gR2/admin/DB10gR2/udump
```

3. ***Obtain a TKProf report of the extended SQL trace that was produced***

- ***Generate a TKProf report and sort the SQL statements in order of most elapsed time using the following command:***

```
tkprof <trace file name> <output file name> sort=fchela,exeela,prsela
```

Compare the execution plan and other execution statistics (physical reads, logical reads, rows returned per execution) of the test script query to the one collected in the Issue Verification phase. If they are comparable, then the test script is valid. If not, its possible that the application had set session-level parameters that changed the execution plan. Additional tips and techniques for constructing a good test script are found in this document.

Is this step optional?
It is optional, but the time to build this test script is usually very short and provides you with a test harness to test the query accurately and repeatedly. Typically, query tuning issues are resolved on the whole much faster by investing in this step.

Next Step - Analyze

In the following step, you will receive guidance on interpreting the data you collected to determine the cause for the performance problem; click "NEXT" to continue.

The data collected in the previous step will be analyzed in this step to determine a cause. It is very important to ensure the data has been collected as completely as possible and for good as well as bad plans.

This process always starts by sanity checking the statistics, optimizer mode, and important initialization parameters and follows with the choice of a tuning strategy that matches your problem and objectives.

Always Check: Optimizer Mode, Statistics, and Parameters

Ensure that the CBO is used, statistics have been gathered properly, and initialization parameters are reasonably set before looking into the details of the execution plan and how to improve it.

Reference Notes
• Gathering Statistics - Best Practices • Interpreting SQLTXPLAIN output • Compare estimated vs. actual cardinality • Which optimizer is being used? • Rule Based Optimizer -- Changing Query Access Paths • Database Initialization Parameters and Configuration for Oracle Applications 11i

Ensure the cost based optimizer is used

The use of the CBO is essential for this tuning effort since the RBO is no longer supported.

1. ***Data Required For Analysis***

- ***Source: Execution plan (collected in "Data Collection", part A)***
  8.1.7 and 9.0.1: Look at the query's "cost", if its is NULL then the RBO was used.
  9.2.x and higher: Look for the text, "Note: rule based optimization" after the plan is displayed to see if the RBO was used.

2. ***Common Observations and Causes***

If the collected data shows the RBO is used, see the table below to find common causes and reasons related to the choice of optimizer:

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Use of the rule based optimizer (RBO)

The RBO is being deprecated in favor of the cost based optimizer (CBO). No specific tuning advice on the RBO will be given in this document. See the references in the sidebar for additional information.

What to look for

Review the execution plan (collected in "Data Collection", part A)
8.1.7 and 9.0.1: Look for the cost column to have NULL values
9.2.x and higher: Look for the text, "Note: rule based optimization" after the plan is displayed.

Cause Identified: No statistics gathered (pre10g)

Oracle will default to the RBO when none of the objects in the query have any statistics. In 10g and to some extent in 9.2, Oracle will use the CBO with dynamic sampling and avoid the RBO.

Cause Justification
The execution plan will not display estimated cardinality or cost if RBO is used. In general, RBO will be used in the following cases (see references for more detail):

No "exotic" (post 8.x) features like partitioning, IOTs, parallelism, etc AND:

- *Pre 9.2.x:*
  - OPTIMIZER_MODE = CHOOSE. Confirm by looking at TKProf, "Optimizer Mode: CHOOSE" for the query
  - and, no statistics on ANY table. Confirm this by looking at each table in SQLTXPLAIN and checking for a NULL value in the "LAST ANALYZED" column
- *9.2x + :*
  - OPTIMIZER_MODE = CHOOSE or RULE
  - and, dynamic sampling disabled (set to level 0 via hint or parameter)
  - and, no statistics on ANY table. Confirm this by looking at each table in SQLTXPLAIN and checking for a NULL value in the "LAST ANALYZED" column

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- *ensuring the sample size is large enough*
- *ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)*
- *ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)*
- *if possible, gather global partition stats*

L *Effort Details*

Low effort; easily scripted and executed.

M *Risk Details*

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be

explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Parameter "optimizer mode" set to RULE

The optimizer_mode parameter will cause Oracle to use the RBO even if statistics are gathered on some or all objects in the query. If a feature such as parallel execution or partitioning is used, then the query will switch over to the CBO.

Cause Justification
The execution plan will not display estimated cardinality or cost if RBO is used. In general, RBO will be used in the following cases (see references for more detail):

No "exotic" (post 8.x) features like partitioning, IOTs, parallelism, etc AND:
- *Pre 9.2.x: optimizer_mode = choose and no statistics on ANY table*
- *9.2x + : optimizer_mode = choose or rule and dynamic sampling disabled*

Solution Identified: Migrate from the RBO to the CBO

The RBO is no longer supported and many features since 8.0 do not use it. The longer term strategy for Oracle installations is to use the CBO. This will ensure the highest level of support and the most efficient plans when using new features.

M *Effort Details*

Migrating to the CBO can be a high or low effort task depending on the amount of risk you are willing to tolerate. The lowest effort involves simply changing the "OPTIMIZER_MODE" initialization parameter and gathering statistics on objects, but the less risky approaches take more effort to ensure execution plans don't regress.

M *Risk Details*

Risk depends on the effort placed in localizing the migration (to a single query, session, or application at a time). The highest risk for performance regressions involve using the init.ora "OPTIMIZER_MODE" parameter.

Solution Implementation

The most cautious approach involves adding a hint to the query that is performing poorly. The hint can be "FIRST_ROWS_*" or "ALL_ROWS" depending on the expected number of rows. If the query can't be changed, then it may be possible to limit the change to CBO to just a certain session using a LOGON trigger.

See the following links for more detail:

Moving from RBO to the Query Optimizer

Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques, see the section "Avoiding Plan Regressions after Database Upgrades"

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Ensure fresh and accurate table / index statistics exist

Accurate statistics on all tables and indexes in the query are essential for the CBO to produce good execution plans.

1. ***Data Required For Analysis***

The data listed here is required for analyzing the causes in the table below.

- ***Source: Execution plan (gathered in "Data Collection", part A)***
  - ***Actual number of rows returned by the query or an execution plan that shows actual and estimated rows per plan step.***
  - ***Estimated number of rows returned by the query ("Estim Card" or similar) from the execution plan***
  - ***Determine if there is a large discrepancy between the actual and estimated rows***
- ***Source: SQLTXPLAIN report, table statistics***
  - ***Examine the "Tables" and "Index" sections, column "Last Analyzed" to determine if the tables and all indexes were analyzed.***
  - ***Compare the columns "Num Rows" and "Sample Size" in the "Tables" section to see how much of the table was sampled for statistics collection.***
  - ***Examine the "Tables" and "Index" sections, column "User Stats" to determine if stats were entered directly rather than analyzed.***
  - ***Examine the "Column Statistics", "Num Buckets", if this is 1, then no histograms were gathered.***

2. Common Observations and Causes

The following table shows common problems and causes related to object statistics:

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

The CBO's estimate of the number of rows returned differs significantly from the actual number of rows returned

Accurate statistics for tables and indexes is the most important factor for ensuring the CBO generates good execution plans. When statistics are missing (and the CBO has to guess) or insufficient, the CBO's estimate for the number of rows returned from each table and the query as a whole can be wrong. If this happens, the CBO may chose a poor execution plan because it based its decision on incorrect estimates.

What to look for

- ***Using the SQLTXPLAIN report, look at the estimated rows returned by the query("Estim Card") for the top-most line of the execution plan***
- ***Compare the estimated rows to the actual rows returned by the query. If they differ by an order of magnitude or more, the CBO may be affected by inadequate statistics.***

Cause Identified: Missing or inadequate statistics

- ***Missing Statistics***
  - ***Statistics were never gathered for tables in the query***
  - ***Gathering was not "cascaded" down to indexes***
- ***Inadequate sample size***
  - ***The sample size was not sufficient to allow the CBO to compute selectivity values accurately***
  - ***Histograms not collected on columns involved in the query predicate that have skewed values***

Cause Justification
One or more of the following may justify the need for better statistics collection:

- ***Missing table statistics: DBA_TABLES.LAST_ANALYZED is NULL***
- ***Missing index statistics: For indexes belonging to each table: DBA_INDEX.LAST_ANALYZED is NULL***
- ***Inadequate sample size for tables: DBA_TABLES.SAMPLE_SIZE / # of rows in the table < 5%***
- ***Inadequate sample size for indexes: DBA_INDEXES.SAMPLE_SIZE / # of rows in the table < 30%***
- ***Histograms not collected: for each table in the query, no rows in DBA_TAB_HISTOGRAMS for the columns having skewed data***
- ***Inadequate number of histograms buckets: For each table in the query, less than 255 rows in DBA_TAB_HISTOGRAMS for the columns having skewed data***

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new

very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- *Gathering Statistics for the Cost Based Optimizer*
- *Gathering Schema or Database Statistics Automatically - Examples*
- *Histograms: An Overview*
- *Best Practices for automatic statistics collection on 10g*
- *How to check what automatic statistics collection is scheduled on 10g*
- *Statistics Gathering: Frequency and Strategy Guidelines*

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- *Collect and Display System Statistics (CPU and IO) for CBO usage*
- *Scaling the System to Improve CBO optimizer*

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Unrealistic Statistics

Values in DBA_TABLES do not match what is known about the table.

What to look for

In ALL_TABLES, the value of NUM_ROWS is actually much larger or smaller than SELECT COUNT(*) FROM table_name.

Cause Identified: Unreasonable table stat values were manually set

Someone either miscalculated or misused DBMS_STATS.SET_STATISTICS

Cause Justification

- ***Check the SQLTXPLAIN report, "Table" or "Index" columns, look for the column "User Stats". If this is YES", then the stats were entered directly by users through the DBMS_STATS.SET_*_STATS procedure.***
- ***Outrageous statistics values are usually associated with very inaccurate estimated cardinality for the query. You can also examine the statistics by looking at things like the number of rows and comparing them to the actual number of rows in the table (SQLTXPLAIN will list both for each table and index).***

  One approach to confirming this is to export the current statistics on certain objects, gather fresh statistics and compare the two (avoid doing this on a production system).

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
```

```
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: The tables have undergone extreme DML changes; stats are old

The table has changed dramatically since the stats were collected due to large DML activity.

Cause Justification
You can determine if significant DML activity has occurred against certain tables in the query by looking in the SQLTXPLAIN report and comparing the "Current COUNT" with the "Num Rows". If there is a large difference, the statistics are stale.

You can also look in the DBA_TAB_MODIFICATIONS table to see how much DML has occurred against tables since statistics were last gathered.

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Histograms were collected on skewed data columns but computed cardinality is still incorrect

The computed cardinality is not correct for a column that is known to contain skewed data despite the fact that histograms have been collected at the maximum bucket size.

What to look for

Actual rows returned by the query do not match what the top-most line for the execution plan reports under "Estim Card" in the SQLTXPLAIN report.

Cause Identified: Long VARCHAR2 strings are exact up to the 32 character position

The histogram endpoint algorithm for character strings looks at the first 32 characters only. If those characters are exactly the same for many columns, the histograms will not be accurate.

Cause Justification
Observe the histogram endpoint values for the column in the SQLTXPLAIN report under the heading "Table Histograms". Many, if not all, endpoint values will be indistinguishable from each other.

Solution Identified: Use Hints to Get the Desired Plan

Hints will override the CBO's choices (depending on the hint) with a desired change to the execution plan. When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to sub-optimal plans.

M ***Effort Details***

Determining the exact hints to arrive at a certain execution plan may be easy or difficult depending on the degree to which the plan needs to be changed.

L ***Risk Details***

Hints are applied to a single query so their effect is localized to that query and has no chance of widespread changes (except for widely used views with embedded hints).
For volatile tables, there is a risk that the hint will enforce a plan that is no longer optimal.

Solution Implementation

See the following resources for advice on using hints.

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Data skewing is such that the maximum bucket resolution doesn't help

The histogram buckets must have enough resolution to catch the skewed data. This usually means at least two endpoints must have the same value in order to be detected as a "popular" (skewed) value. Skewing goes undetected when the number of samples in each bucket is so large that truly skewed values are buried inside the bucket.

Cause Justification
Check the following for the column suspected of having skewed data:

1. ***A crude way to confirm there is skewed data is by running this query:***
   SELECT AVG(col1)/((MIN(col1)+MAX(col1))/2) skew_factor FROM table1
   col1, table1 refers to the column/table that has skewed data.
2. ***Examine the output of the query for skewing; when the "skew_factor" is much less than or much greater than 1.0, there is some skewing.***
3. ***Look at the endpoint values for the column in SQLTXPLAIN ("Table Histograms" section) and check if "popular" values are evident in the bucket endpoints (a popular value will have the same endpoint repeated in 2 or more buckets - these are skewed values)***
4. ***If the histogram has 254 buckets and doesn't show any popular buckets, then this cause is justified.***

Solution Identified: Use Hints to Get the Desired Plan

Hints will override the CBO's choices (depending on the hint) with a desired change to the execution plan. When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to sub-optimal plans.

M ***Effort Details***

Determining the exact hints to arrive at a certain execution plan may be easy or difficult depending on the degree to which the plan needs to be changed.

L ***Risk Details***

Hints are applied to a single query so their effect is localized to that query and has no chance of widespread changes (except for widely used views with embedded hints).
For volatile tables, there is a risk that the hint will enforce a plan that is no longer optimal.

Solution Implementation

See the following resources for advice on using hints.

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Manually set histogram statistics to reflect the skewing in the column's data

The histogram will need to be manually defined according to the following method

1. *Find the values where skewing occurs most severely*
2. *Use DBMS_STATS.SET_TABLE_STATS to enter the endpoints and endpoint values representing the skewed data values*

H *Effort Details*

High effort; It will take some effort to determine what the endpoint values should be and then set them using DBMS_STATS.

M *Risk Details*

Medium risk; By altering statistics manually, there is a chance that a miscalculation or mistake may affect many queries in the system. The change may also destabilize good plans.]

Solution Implementation

Details for this solution are not yet available.

Related documents:
Interpreting Histogram Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Ensure reasonable initialization parameters are set

The CBO uses the values of various initialization parameters to estimate the cost of various operations in the execution plan. When certain parameters are improperly set, they can cause the cost estimates to be inaccruate and cause suboptimal plans.

1. ***Data Required For Analysis***

The data listed here is required for analyzing the causes in the table below.

- ***Source: SQLTXPLAIN report, Optimizer Trace section, "Parameters Used by the Optimizer"***

2. ***Common Observations and Causes***

The following table shows common problems and causes related to object statistics:

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Parameter settings affecting table access paths and joins

Certain initialization parameters may be set too aggressively to obtain better plans with certain queries. These parameters may adversely affect other queries and cause them to favor full table scans and merge or hash joins instead of index access with nested loop joins.

What to look for

- ***Parameter settings affecting the optimizer are set to non-default values***
- ***The CBO chooses an access path, join order, or operation that is sub-optimal in comparison to another better plan (e.g., CBO chooses a full table scan instead of an index)***

Cause Identified: Parameters causing full table scans and merge/hash joins

The following parameters are known to affect the CBO's cost estimates :
- ***optimizer_index_cost_adj set much higher than 100***
- ***db_file_multiblock_read_count set too high (greater than 1MB / db_block_size)***
- ***optimizer_mode=all_rows***

Cause Justification
Full table scans, merge/hash joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L ***Effort Details***

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H ***Risk Details***

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Parameters causing index scans and nested loop joins

The following parameters are known to bias the CBO towards index scans and nested loop joins :
- ***optimizer_index_cost_adj set much lower than 100***
- ***db_file_multiblock_read_count set too low (smaller than 1MB / db_block_size)***
- ***optimizer_index_caching set too high***
- ***optimizer_mode=first_rows (or first_rows_N)***

Cause Justification
Index scans and nested loop joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L ***Effort Details***

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H ***Risk Details***

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Init.ora parameters not set for Oracle Applications 11i

Oracle Applications 11i requires certain database initialization parameters to be set according to specific recommendations

Cause Justification
Oracle Applications 11i in use and init.ora parameters not set accordingly

Solution Identified: Set Database Initialization Parameters for Oracle Applications 11i

Oracle Applications 11i have strict requirements for database initialization parameters that must be followed. The use of these parameters generally result in much better performance for the queries used by Oracle Apps. This is a minimum step required when tuning Oracle Apps.

L ***Effort Details***

Low effort; simply set the parameters as required.

L ***Risk Details***

Low risk; these parameters have been extensively tested by Oracle for use with the Apps.

Solution Implementation

See the notes below.

Database Initialization Parameters and Configuration for Oracle Applications 11i
bde_chk_cbo.sql - Reports Database Initialization Parameters related to an Apps 11i instance

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Choose a Tuning Strategy

Determining the cause for a query performance problem can be approached in various ways. Three approaches are presented here:

Questions that influence the choice of strategy

Documentation
- 10g: Automatic SQL Tuning
- SQL Tuning Overview
- The Query Optimizer
- Parameter: OPTIMIZER_DYNAMIC_SAMPLING
- Parameter: OPTIMIZER_FEATURES_ENABLE
- Parameter: OPTIMIZER_INDEX_CACHING
- Parameter: OPTIMIZER_INDEX_COST_ADJ
- Hint: PARALLEL
- Hint: MERGE

The answers to the following questions will help you choose an appropriate tuning approach.

1. ***Do you have an execution plan when the query ran well?***

Rationale: If you have a "good" execution plan in addition to the current "bad" one, then you can use the "Execution Plan Comparison" strategy to find where the plans differ. Once you know where they differ, you can modify the query to produce a good plan, or focus on the particular place where they differ and determine the cause for the difference

2. ***Are you mostly interested in solving this problem quickly rather than getting to the cause?***

Rationale:If you have an urgent need to make the query perform well and you aren't interested in the underlying cause of the problem, you can use the "Quick Solution" strategy to give the CBO more information and possibly obtain a better execution plan.

Once you obtain a better plan, you have the option of performing a plan comparison and then a deeper analysis to find the root cause, or influence the CBO in whatever way possible to obtain the plan.

3. ***Does the query spend most of its time in the execute/fetch phases (not parse phase)?***

Rationale: If the query spends most its time parsing, normal query tuning techniques that alter the execution plan to reduce logical I/O during execute or fetch calls probably won't help. The focus of the tuning should be in reducing parse times; see the "Parse Reduction" strategy.

For example, here is an excerpt from a TKProf for a query:

```
SELECT * FROM  ct_dn dn, ds_attrstore store . . .

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ----------  ----------  ----------
Parse      555    100.09     300.83          0          0           0           0
Execute    555      0.42       0.78          0          0           0           0
Fetch      555     14.04      85.03        513    1448514           0       11724
------- ------  -------- ---------- ---------- ----------  ----------  ----------
total     1665    114.55     386.65        513    1448514           0       11724
```

The elapsed time spent parsing was 300.83 seconds compared to only 85.03 seconds for fetching. This query is having trouble parsing - tuning the query's execution plan to reduce the number of buffers read during the fetch call will not give the greatest performance gain (in fact only about 85 out of 386 seconds could be improved in the fetch call).

Using Oracle 10g ==> Use the SQL Tuning Advisor

- Hint: NO_MERGE
- Hint: PUSH_PRED
- Hint: PUSH_SUBQ
- Hint: UNNEST
- Using Plan Stability (Stored Outlines)
- Stored Outline Quick Reference

How To

- Diagnosing Query Tuning Problems
- Troubleshooting Oracle Applications Performance Issues
- How to Tune a Query that Cannot be Modified
- How to Move Stored Outlines for One Application from One Database to Another
- SQLTXPLAIN report: How to determine if an index should be created
- How to compare actual and estimated cardinalities in each step of the execution plan

Reference Notes

- Interpreting Explain plan
- Using Plan Stability (Stored Outlines)
- Stored Outline Quick Reference
- Diagnosing Why a Query is Not Using an Index
- Affect of Number of Tables on Join Order Permutations
- Checklist for Performance Problems with Parallel Execution
- Why did my query go parallel?

Oracle 10g is able to perform advanced SQL tuning analysis using the SQL Tuning Advisor (and related Access Advisor). This is the preferred way to begin a tuning effort if you are using Oracle 10g.

Note: You must be licensed for the "Tuning Pack" to use these features.

High Parse Times ==> Parse Time Reduction Strategy

Reduction in high parse times requires a different approach to query tuning than the typical goals of reducing logical I/O or inefficient execution plans.

When to use: Use this approach when you have determined the query spends most of its time in the parse phase (this was done when you verified the issue).

Data Required for Analysis

- Source: TKProf
  - parse elapsed time, overall elapsed time
  - parse time spent on CPU; Example of a query with high parse CPU
  - parse time spent waiting (not in CPU); Example of a query with high parse Waits
- If the parse time spent on CPU is more than 50% of the parse elapsed time, then the parse time is dominated by CPU; otherwise it is dominated by waits.

See the appropriate section below based on the data collected.

1. ***CPU time dominates the parse time***

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

High CPU usage during HARD parse

High CPU usage during hard parses are often seen with large statements involving many objects or partitioned objects.

What to look for

1. ***Check if the statement was hard parsed***
2. ***Compare parse cpu time to parse elapsed time to see if parse cpu time is more than 50%***

Cause Identified: Dynamic sampling is being used for the query and impacting the parse time

Dynamic sampling is performed by the CBO (naturally at parse time) when it is either requested via hint or parameter, or by default because statistics are missing. Depending on the level of the dynamic sampling, it may take some time to complete - this time is reflected in the parse time for the statement.

Cause Justification

- ***The parse time is responsible for most of the query's overall elapsed time***
- ***The execution plan output of SQLTXPLAIN, the UTLXPLS script, or a 10053 trace will show if dynamic sampling was used while optimizing the query.***

Solution Identified: Alternatives to Dynamic Sampling

If the parse time is high due to dynamic sampling, alternatives may be needed to obtain the desired plan without using dynamic sampling.

M ***Effort Details***

Medium effort; some alternatives are easy to implement (add a hint), whereas others are more difficult (determine the hint required by comparing plans)

L ***Risk Details***

Low risk; in general, the solution will affect only the query.

Solution Implementation

Some alternatives to dynamic sampling are:

1. ***In 10g or higher, use the SQL Tuning Advisor (STA) to generate a profile for the query (in fact, its unlikely you'll even set dynamic sampling on a query that has been tuned by the STA)***
2. ***Find the hints needed to implement the plan normally generated with dynamic sampling and modify the query with the hints***
3. ***Use a stored outline to capture the plan generated with dynamic sampling***

For very volatile data (in which dynamic sampling was helping obtain a good plan), an approach can be used where an application will choose one of several hinted queries depending on the state of the data (i.e., if data recently deleted use query #1, else query #2).

Documents for hints:

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Documents for stored outlines / plan stability:

Using Plan Stability

Stored Outline Quick Reference

How to Tune a Query that Cannot be Modified

How to Move Stored Outlines for One Application from One Database to Another

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has many IN LIST parameters / OR statements

The CBO may take a long time to cost a statement with dozens of IN LIST / OR clauses.

Cause Justification

- *The parse time is responsible for most of the query's overall elapsed time*
- *The query has a large set of IN LIST values or OR clauses.*

Solution Identified: Implement the NO_EXPAND hint to avoid transforming the query block

In versions 8.x and higher, this will avoid the transformation to separate query blocks with UNION ALL (and save parse time) while still allowing indexes to be used with the IN-LIST ITERATOR operation. By avoiding a large number of query blocks, the CBO will save time (and hence the parse time will be shorter) since it doesn't have to optimize each block.

L *Effort Details*

Low effort; hint applied to a query.

L *Risk Details*

Low risk; hint applied only to the query and will not affect other queries.

Solution Implementation

See the reference documents.

Optimization of large inlists/multiple OR`s

NO_EXPAND Hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Partitioned table with many partitions

The use of partitioned tables with many partitions (more than 1,000) may cause high parse CPU times while the CBO determines an execution plan.

Cause Justification

1. *The parse time is responsible for most of the query's overall elapsed time*
2. *Determine total number of partitions for all tables used in the query.*
3. *If the number is over 1,000, this cause is likely*

Solution Identified: 9.2.0.x, 10.0.0: Bug 2785102 - Query involving many partitions (>1000) has high CPU/ memory use

A query involving a table with a large number of partitions takes a long time to parse, causes rowcache contention, and high CPU consumption. The case of this bug involved a table with greater than 10000 partitions and global statistics ere not gathered.

M *Effort Details*

Medium effort; application of a patchset.

L *Risk Details*

Low risk; patchsets generally are low risk because they have been regression tested.

Solution Implementation

Apply patchset 9.2.0.4

Workaround:
Set "_improved_row_length_enabled"=false

Additional bug information:
Bug 2785102

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. *Wait time dominates the parse time*

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

High wait time during HARD parse

High wait time during hard parses usually occur due to contention for resources or are related to very large queries.

What to look for

1. *Check if the statement was hard parsed (See TKProf, "Misses in the library cache" for the statement - if this is equal to one or higher, then this statement was hard parsed)*
2. *Examine the waits in the TKProf for this statement; you will see waits like "SQL*Net more data FROM client" or waits related to latches, library cache locks or pins.*

Cause Identified: Waits for large query texts to be sent from the client

A large query (containing lots of text) may take several round trips to be sent from the client to the server; each trip takes time (especially on slow networks).

Cause Justification

1. *High parse wait times occur any time, not just during peak load or during certain times of the day*
2. *Most other queries do not have high parse wait times at the same time as the query you are trying to tune*
3. *TKProf shows "SQL*Net more data from client" wait events.*
4. *Raw 10046 trace shows "SQL*Net more data from client" waits just before the PARSE call completes*
5. *Slow network ping times due to high latency networks make these waits worse*

Solution Identified: Use PL/SQL REF CURSORs to avoid sending query text to the server across the network

The performance of parsing a large statement may be improved by encapsulating the SQL in a PL/SQL package and then obtaining a REF CURSOR to the resultset. This will avoid sending the SQL statement across the network and will only require sending bind values and the PL/SQL call.

M *Effort Details*

Medium effort; a PL/SQL package will need to be created and the client code will need to be changed to call the PL/SQL and obtain a REF CURSOR.

L *Risk Details*

Low risk; there are changes to the client code as well as the PL/SQL code in the database that must be tested thoroughly, but the changes are not widespread and won't affect other queries.

Solution Implementation

See the documents below.

How to use PL/SQL REF Cursors to Return Result Sets

Using Cursor Variables (REF CURSORs)

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Only Bad Plan Available, Fast Solution Desired ==> Quick Solution Strategy

The goal of this strategy is to change some high-level settings of the optimizer and see if a better plan results (e.g., change the optimizer_mode or use dynamic sampling). Then, use hints or other means to make the CBO generate the better plan. The better plan can be used to find an underlying cause later.

1. ***Assumptions***

- ***Oracle 10g or higher: You have already tried the SQL Tuning Advisor (STA). Do not use these techniques if you are licensed for the STA and haven't tried using it first.***
- ***You have read the section above, "Always Check: Optimizer Mode, Statistics, and Parameters" and have ensured that statistics are being gathered properly. In summary, we want to ensure the following:***
  - ***The CBO is being used***
  - ***Statistics have been gathered for all objects, tables as well as indexes***
  - ***The sample size is as large as possible - "COMPUTE" if possible***
  - ***Histograms have been gathered on any columns in the query predicate that may have skewed data***
  - ***Global partition stats are gathered on a partitioned table***

2. ***Preparations***

You will need the query text and bind values (if applicable) to run on the system where the query is slow (or a test system where this problem is reproduced).

The use of a test script is very valuable for the techniques in this section. See the instructions in "Determine a Cause" > Data Collection > D. Construct a Test Script

3. ***Discover a Better Execution Plan - Basic Techniques***

The following changes should be tried first - they are likely to give the best results in the shortest time.

1. ***Change the optimizer mode: If the optimizer mode is currently ALL_ROWS, use FIRST_ROWS_N (choose a value for N that reflects the number of rows that the user wants to see right away) or vice versa. The optimizer mode may be changed via hint or parameter, for example:***
   - ***Hint:***

```
SELECT /*+ first_rows_1 */  col1, col2, ...
FROM table1...
WHERE col1 = 1 AND ...
```

   - ***Parameter:***

```
ALTER SESSION SET optimizer_mode = first_rows_1;
SELECT  col1, col2, ...
FROM table1...
WHERE col1 = 1 AND ...
```

2. ***Dynamic Sampling: This will sample the number of rows returned by the query and determine very accurate selectivity estimates that often lead to good execution plans. There are two ways to use it:***
   - ***Hint:***

```
SELECT /*+ dynamic_sampling(5) */  col1, col2, ...
FROM table1...
WHERE col1 = 1 AND ...
```

   - ***Parameter:***

```
ALTER SESSION SET optimizer_dynamic_sampling = 5;
SELECT  col1, col2, ...
FROM table1...
WHERE col1 = 1 AND ...
```

   The dynamic sampling levels generally control how large the sample size will be. A setting of 10 will cause all rows to be sampled from the tables in the query - if the tables are large this will take some time. Its a good idea to start with a level of 5 and increase it until the performance of the query improves.

   Note: The query should be run TWICE to truly determine if the performance is better. This is because the first time the query is parsed, the dynamic sampling effort may take considerable time and doesn't reflect the execution plan's performance. The second run will indicate if the

execution plan is improved and hence performance is better.

3. *OPTIMIZER_FEATURES_ENABLE parameter: If this query performed better in an older version (say prior to a migration), then use this parameter to "rollback" the optimizer to the older version. In Oracle 10g and higher, this can be set at the session level which is preferred over system-wide level.*

4. *Replace bind variables with literals : Sometimes bind peeking doesn't occur for all bind variables (may happen for certain complex queries). Substituting the literals will allow the CBO to have accurate values.*

   Note: the use of literals is strictly for this test; it is not recommended to use literals on production OLTP applications due to concurrency issues.

4. *Discover a Better Execution Plan - Additional Techniques*

Try these changes if the basic techniques did not result in a better execution plan.

1. *If the query returns FEW rows*
   - *ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 10 (or lower)*
   - *ALTER SESSION SET OPTIMIZER_INDEX_CACHING = 90 (or higher)*

2. *If the query returns MANY rows*
   - *ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 1000 (or higher)*
   - *ALTER SESSION SET OPTIMIZER_INDEX_CACHING = 0*
   - *PARALLEL hint; Use parallelism if sufficient CPU and I/O bandwidth are available to run the statement (along with other sessions concurrently)*

3. *If the query uses views, try the following hints:*
   - *PUSH_PRED, NO_MERGE*
   - *MERGE*

4. *If the query has a subquery, try the following hints:*
   - *PUSH_SUBQ*
   - *UNNEST*

Note: In Oracle 10gR2, you can set an initialization parameter in a hint so it affects only the query being tuned. This undocumented hint is called "OPT_PARAM". It may be used as follows:
For text values (e.g., 'TRUE'): OPT_PARAM('initialization parameter name' 'parameter value - text')
For numeric values: OPT_PARAM('initialization parameter' 99)

For example:

```
SELECT /*+ OPT_PARAM('optimizer_index_adj' 10) */ col1, col2
FROM table1 WHERE col1 = 10;
```

This hint will be documented in later versions.

5. ***Implement the New Good Plan***

This section will identify ways to implement the new, good plan you discovered through the techniques above.

- ***If you are able to modify the query or application...***

    1. ***Examine the good plan and find the difference with the bad plan.***
       Often, this will be something simple like the use of an index or a different join order.
       See the "Execution Plan Comparison" strategy section for more details.
    2. ***Find a hint for the query to make the CBO generate the good plan.***
       Sometimes it is helpful to obtain stored outlines for the bad execution plan and the good one to compare and see what hints may be needed.

       If you aren't able to find a suitable hint, try the method below using stored outlines.

- ***If you are NOT able to modify the query (third party application, etc)...***

    - ***Use stored outlines to "lock in" a good plan***
        1. ***Use session-based initialization parameters to change the execution plan***
        2. ***Capture a stored outline for the query (use ALTER SESSION CREATE_STORED_OUTLINES command)***
        3. ***Verify the stored outline causes the query to perform well***
        4. ***Test the stored outline on a test system***
        5. ***Implement the stored outline in production***

    - ***Use initialization parameters to influence the CBO***
        1. ***Use a LOGON trigger or change the application to set its session parameters to values that improve the query's performance. This approach is not recommended in most cases because it may cause undesirable changes to other queries that perform well.***

6. ***Follow-up***

If the problem was not solved or if you need to find the underlying cause, see the "Plan Analysis" strategy for a more rigorous way to help identify the underlying cause.

If you would like to log a service request, see the section below for more details.

Good and Bad Execution Plan Available => Execution Plan Comparison Strategy

NOTE: This section is still under construction.

The goal of this strategy is to compare the good and bad execution plans, find the differences, and look for ways to make the bad plan change so it becomes like the good one.

When to use: Use this approach when:

- a good plan is available
- a reasonably quick solution is required
- determining an underlying cause is not required (due to time constraints)
- the query may be modified (hints, etc)

1. ***Review the "Always Check:..." section to ensure the CBO has enough information to make a good choice***
2. ***Obtain a "good" plan and a "bad" plan***

Ideally, you have collected all of the data described in the data collection step for both good and bad plans. With this information, it will be possible to compare both plans.

Note: Oracle 10gR2's PLAN_TABLE has a column called "OTHER_XML" which contains the hints needed to produce the plan. If the good plan is from 10gR2, view the contents of this column to extract the hints for the query. You can skip the rest of this procedure since the hints are complete, or you can continue and eliminate all of the hints but the ones actually needed to enforce the plan.

3. ***Walk through the plan steps and compare***

The remainder of the steps will compare the join orders, join types, access methods, and other operations between the two execution plans. This comparison is done by "walking" through the execution step-by-step in the order the plan is executed by Oracle.

The SQLTXPLAIN report's execution plan output has a column called, "Exec Order" which will guide you on the proper sequence of steps (the plan starts with step 1). For example:

Id	Exec Order	Explain Plan Operation
0:	5	SELECT STATEMENT
1:	4	SORT GROUP BY
2:	3	HASH JOIN
3:	1	INDEX FAST FULL SCAN TOWNER.TEAMS_LINKS_IDX_001
4:	2	INDEX FAST FULL SCAN TOWNER.UOIS_IDX_003

In this case, the execution will start with the step that has the "Explain Plan Operation" as "INDEX FAST FULL SCAN TOWNER.TEAMS_LINKS_IDX_001", followed by "INDEX FAST FULL SCAN TOWNER. UOIS_IDX_003"

4. *Comparing and changing the join order*

Comparing the Join Orders

The join order is one of the most important factors for query performance for queries involving many tables. The join order for the good plan may be obtained in one of two ways:

- *10053 Trace File: Using the 10053 trace for the good plan, find the final join order chosen by the CBO. The join order is simply read from top to bottom (from Table #0 to Table # N). For example:*

```
Best so far: TABLE#: 0 CST: 8502 CDN: 806162 BYTES: 72554580
Best so far: TABLE#: 1 CST: 40642 CDN: 806162 BYTES: 112056518
```

To map the table #s to actual names, you'll need to find the line at the top of the join order, similar to this:

```
Join order[1]: EMP [ E] DEPT [ D]
```

Table #0 is the furthest left table, Table #1 is the next to the right and so on.

In this case, the order is EMP, DEPT

- *SQLTXPLAIN File: Run through the plan in the order of the execution (SQLTXPLAIN has a column called "Exec Order" which tells you this), and take note of the order of the tables as you "walk" through the plan.*

Id	Exec Order	Explain Plan Operation
0:	5	SELECT STATEMENT
1:	4	SORT GROUP BY
2:	3	HASH JOIN
3:	1	INDEX FAST FULL SCAN TOWNER.TEAMS_LINKS_IDX_001
4:	2	INDEX FAST FULL SCAN TOWNER.UOIS_IDX_003

In this case, two indexes are scanned instead of the actual tables, so we need to find the tables they correspond to by looking in the SQLTXPLAIN report:

Table Owner.Table Name	Index Owner.Index Name	Used	Index Type	Uniqueness	Indexed Columns
TOWNER.TEAMS_LINKS	TOWNER.TEAMS_LINKS_DEST_I		NORMAL	NONUNIQUE	DEST_VALUE DEST_TYPE
TOWNER.TEAMS_LINKS	TOWNER.TEAMS_LINKS_IDX_001	3	NORMAL	NONUNIQUE	LINK_TYPE DEST_VALUE SRC_VALUE
TOWNER.TEAMS_LINKS	TOWNER.TEAMS_LINKS_IDX_002		NORMAL	NONUNIQUE	LINK_TYPE SRC_VALUE DEST_VALUE
TOWNER.TEAMS_LINKS	TOWNER.TEAMS_LINKS_PK		NORMAL	UNIQUE	SRC_VALUE DEST_VALUE LINK_TYPE

...	...		...	...	...
TOWNER.UOIS	TOWNER.UOIS_IDX_002		NORMAL	NONUNIQUE	CONTENT_TYPE CONTENT_STATE
TOWNER.UOIS	TOWNER.UOIS_IDX_003	4	NORMAL	NONUNIQUE	UOI_ID CONTENT_STATE
TOWNER.UOIS	TOWNER.UOIS_METADATA_STATE_DT		NORMAL	NONUNIQUE	METADATA_STATE_DT
...	...		...	...	...
TOWNER.UOIS	TOWNER.UOI_UCT_I		NORMAL	NONUNIQUE	CONTENT_TYPE

Now we can resolve that the table order is: 1) TOWNER.TEAM_LINKS, 2) TOWNER.UOIS

Compare the order of the good plan to the bad plan. If they differ, use the "ORDERED" or "LEADING" hints to direct the CBO.

Changing the Join Order
For example, in the case above, we can make the join order of the good plan with the the following hint:

```
SELECT /*+ ordered */ TL.COL1, TU.COL2 FROM TOWNER.TEAM_LINKS TL, TOWNER.UOIS TU WHERE ...
```

5. ***Compare data access methods of both plans***
6. ***Compare join types of both plans***
7. ***Identify major operations that differ between plans***
8. ***Make the CBO generate the "good" plan***

Once the difference between the plans has been found, the query will need to be modified with a hint to cause the good plan to be generated. If its not possible to change the query, then alternative ways to change the query may be needed, such as:

- ***Use stored outlines to "lock in" a good plan***
  1. ***Use session-based initialization parameters to change the execution plan***
  2. ***Capture a stored outline for the query (use ALTER SESSION CREATE_STORED_OUTLINES command)***
  3. ***Verify the stored outline causes the query to perform well***
  4. ***Test the stored outline on a test system***
  5. ***Implement the stored outline in production***
- ***Use initialization parameters to influence the CBO***
  1. ***Use a LOGON trigger or change the application to set its session parameters to values that improve the query's performance. This approach is not recommended in most cases because it may cause undesirable changes to other queries that perform well.***

Only Bad Plan Available, Thorough Analysis Desired ==> Plan Analysis Strategy

Review the query text, access paths, join orders, and join methods for common problems; implement the solutions to these problems. This is the default approach to query tuning issues and is the main subject of this phase. In summary, this approach will focus on :

- Statistics and Parameters: Ensure statistics are up-to-date and parameters are reasonable
- SQL statement structure: Look for constructs known to confuse the optimizer
- Data access paths: Look for inefficient ways of accessing data knowing how many rows are expected
- Join orders and join methods: Look for join orders where large rowsources are at the beginning; look for inappropriate use of join types
- Other operations: Look for unexpected operations like parallelism or lack of partition elimination

When to use: Use this approach when:

- a good plan is not available
- an urgent solution is not required
- determining an underlying cause is desired
- the query may be modified (hints, etc)

- - - - - A. Examine the SQL Statement

Look at the query for common mistakes that occur. Understand the volume of data that will result when the query executes.

1. Ensure no join predicates are missing

Missing join predicates cause cartesian products that are very inefficient and should be avoided. They are usually the result of a mistake in the SQL. You can identify this easily by looking at each table in the FROM clause and ensuring the WHERE clause contains a join condition to one or more of the other tables in the FROM clause. For example,

```
SELECT e.ename, d.dname FROM scott.emp e, scott.dept d WHERE e.empno < 1000
```

Should be rewritten as:

```
SELECT e.ename, d.dname FROM scott.emp e, scott.dept d WHERE e.empno < 1000 AND e.deptno = d.deptno
```

2. ***Look for unusual predicates***

Unusual predicates are usually present due to query generators that do not take into account the proper use of the CBO. These predicates may cause the CBO to inaccurately estimate the selectivity of the predicate. Some unusual predicates are:

- ***Repeated identical predicates: E.g., WHERE col1 =1 and col1 =1 and col1 =1 and ...., this will cause the CBO underestimate the selectivity of the query. Assuming that the selectivity of "col1 = 1" is 0.2, the clause above would have its selectivity = 0.2 * 0.2 * 0.2 = 0.008. This would mean that the estimated cardinality would be much smaller than it actually is and may chose an index path when a full table scan would have been a better choice or the CBO might decide to begin a join order using a table that returns many more rows than is estimated.***
- ***Join predicates where both sides are identical: E.g. WHERE d.deptno = d.deptno, this has the effect of 1) removing NULLs from the query (even on a NULLable column) and 2) underestimating the cardinality.***

The unusual predicates should be removed if they are causing unexpected results (no NULLs) or bad execution plans due to incorrect cardinality estimates. If there is no way to change the query, it might be beneficial to upgrade to version 9.2 or greater to take advantage of the CBO's awareness of these kinds of predicates.

Other common problems are listed below:

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Bad performance using Fine Grained Access Control (FGAC)

A query performs well unless FGAC is used. Whenever FGAC is avoided, the performance improves.

What to look for

1. ***Compare the execution plans when FGAC was used to when it wasn't. There should be a difference in plans***
2. ***The plan with the FGAC use should have additional predicates generated automatically and visible in the "access predicate" or "filter predicate" output of the execution plan (in SQLTXPLAIN)***

Cause Identified: Index needed for columns used with fine grained access control

The use of FGAC will cause additional predicates to be generated. These predicates may be difficult for the CBO to optimize or they may require the use of new indexes.

Cause Justification

1. ***Query performance improves when FGAC is not used***
2. ***Manually adding the FGAC-generated predicates to the base query will reproduce the problem.***
3. ***Event 10730 shows the predicate added by FGAC and it matches the predicate seen in the execution plan's access and filter predicates***

Solution Identified: Create an index on the columns involved in the FGAC

FGAC introduces additional predicates that may require an index on the relevant columns. In some cases, a function-based index may be needed.

L ***Effort Details***

Low effort; just add an index or recreate an index to include the columns used in the security policy.

L ***Risk Details***

The index should have little negative impact except where a table already has many indexes and this index causes DML to take longer than desired.

Solution Implementation

TBD

TBD

TBD

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Bug 5195882

Queries In FGAC Using Full Table Scan Instead Of Index Access

This bug prevents view merging when PL/SQL functions and views are involved - which is common when FGAC is used. The inability to merge views leads to bad execution plans.

Cause Justification

1. ***Query performance improves when FGAC is not used***
2. ***Manually adding the FGAC-generated predicates to the base query will reproduce the problem.***
3. ***Event 10730 shows the predicate added by FGAC and it matches the predicate seen in the execution plan's access and filter predicates***

Solution Identified: Apply patch for bug 5195882 or use the workaround

Patch and workaround available.

M ***Effort Details***

Requires a patch application. The workaround is lower effort, but side effects are unknown.

M ***Risk Details***

If applying the one-off patch, it carries the risk typically associated with one-off patches.
Patchset 10.2.0.3 has the fix for this bug and is lower risk since patchsets are rigorously tested.

Solution Implementation

Contact Oracle Support Services for the patch.

Workaround:
Set optimizer_secure_view_merging=false

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

3. Look for constructs known to cause problems

Certain constructs are known to cause problems with the CBO. Some of these include:

- *Large IN lists / OR statements*
- *Outer Joins*
- *Hierarchical queries*
- *Views, inline views, or subqueries*

See the table below for common causes related to SQL constructs.

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Bad Execution Plans with Large IN Lists / OR statements
Large IN lists / OR statements are sometimes difficult for the CBO to cost accurately. *What to look for* *Query contains clauses of the form:* • *...WHERE col1 IN (1, 2, 3, 4, 5, ...)* • *...WHERE col1 = 1 OR col1 = 2 OR col1 = 3 ...*

Cause Identified: CBO costs a full table scan cheaper than a series of index range scans
The CBO determines that it is cheaper to do a full table scan than to expand the IN list / OR into separate query blocks where each one uses an index. *Cause Justification* *Full table scans in the execution plan instead of a set of index range scans (one per value) with a CONCATENATION operation.*

Solution Identified: Implement the USE_CONCAT hint to force the optimizer to use indexes and avoid a full table scan

This hint will force the use of an index (supplied with the hint) instead of using a full table scan (FTS). For certain combinations of IN LIST and OR predicates in queries with tables of a certain size, the use of an index may be far superior to an FTS on a large table.

L ***Effort Details***

Low; simply use the hint in the statement (assuming you can alter the statement).

L ***Risk Details***

Low; will only affect the single statement.

Solution Implementation

See the notes below.

Using the USE_CONCAT hint with IN/OR Statements

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

- - - - - B. Examine Data Access Paths

The choice of access path greatly affects the performance of queries. If the query has a predicate that will reduce the number of rows from a table, then the use of an index is usually beneficial (hopefully, indexes exist for the columns in the predicate). On the other hand, if there is no predicate to filter the rows from a table or the predicate is not very selective and many rows are expected, a full table scan may be a better choice over an index scan.

It is important to understand the actual number of rows expected for each plan step and compare it to the CBO's estimate so that you can determine if FTS or index access makes more sense.

Data Required for Analysis

- Source: Execution plan (gathered in "Data Collection", part A)
  - Actual number of rows returned by the query or an execution plan that shows actual and estimated rows per plan step.
  - Estimated number of rows returned by the query ("Estim Card" or similar) from the execution plan
  - Determine if there is a large discrepancy between the actual and estimated rows

1. ***Few rows expected to be returned by the query (typically OLTP apps)***

If you know that few rows will be returned by the query or by a particular predicate, then you should expect to see an index access method to retrieve those rows. Look for places in the plan where the actual cardinality is low (the estimated one may be high due to an inaccurate CBO estimate) but an index is not used.

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Query is not using an index, or is using the "wrong" one

Either an index is not available for one or more columns in the query's predicate(s), or an available index is not chosen by the CBO.

What to look for

The execution plan shows that an index is not used (FTS is used instead typically) to access rows from a table for a particular predicate's column(s) AND

- ***The column(s) do not have an index***
  or
- ***An existing index that contains the column is not used***

Cause Identified: No index available for columns in the predicate

No indexes have been defined for one or more columns in the query predicate. Oracle only has a full table scan access method available in this case.

Cause Justification

1. ***For each column in the query's WHERE clause, check that there is an index available. In some cases, multiple columns from a table will be in the WHERE clause - ideally, there is an index defined with these columns as the leading columns of the index.***
2. ***Examine the execution plan in the SQLTXPLAIN report and look for predicates using the "FILTER()" function rather than the "ACCESS()" function. Predicates obtained via ACCESS() were obtained using an index (more efficiently and directly), whereas those obtained via FILTER() where obtained by applying a condition to a row source after the data was obtained.***

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- ***A column in the predicate is not indexed; if it were, a full table scan would be avoided***
- ***The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective***
- ***For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better***

M ***Effort Details***

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M ***Risk Details***

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following

document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: 10g+ : Use the SQL Access Advisor for Index Recommendations

The SQL Access Advisor recommends bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

L *Effort Details*

Low effort; available through Enterprise Manager's GUI or via command line PL/SQL interface.

M *Risk Details*

Medium risk; changes to indexes should be tested in a test system before implementing in production because they may affect many other queries.

Solution Implementation

Please see the following documents:

SQL Access Advisor

Tuning Pack Licensing Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Incorrect selectivity estimate

The CBO needs additional information for estimating the selectivity of the query (in maybe just one of the plan steps). Usually this is due to predicate clauses that have some correlation. The CBO assumes that filter predicates are independent of each other; when ANDed, these predicates reduce the number of rows returned (increased selectivity). However, when these predicates are not independent (e.g., a query that filtered on the city name and postal code), more rows are returned than the CBO estimates. This leads to inaccurate cost estimates and inefficient plans.

Cause Justification
The estimated vs. actual cardinality for the query or for individual plan steps differ significantly.

Solution Identified: Use Hints to Get the Desired Plan

Hints will override the CBO's choices (depending on the hint) with a desired change to the execution plan. When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to sub-optimal plans.

M ***Effort Details***

Determining the exact hints to arrive at a certain execution plan may be easy or difficult depending on the degree to which the plan needs to be changed.

L ***Risk Details***

Hints are applied to a single query so their effect is localized to that query and has no chance of widespread changes (except for widely used views with embedded hints).
For volatile tables, there is a risk that the hint will enforce a plan that is no longer optimal.

Solution Implementation

See the following resources for advice on using hints.

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use Plan Stability to Set the Desired Execution Plan

Plan stability preserves execution plans in stored outlines. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If the use of the outline is enabled for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints.

The performance of a statement is improved without modifying the statement (assuming an outline can be created with the hints that generate a better plan).

M ***Effort Details***

Medium effort; Depending on the circumstance, sometimes an outline for a query is easily generated and used. The easiest case is when a better plan is generated simply by changing an initialization parameter and an outline is captured for the query.
In other cases, it is difficult to obtain the plan and capture it for the outline.

L ***Risk Details***

Low risk; the outline will only affect the associated query. The outline should be associated with a category that enables one to easily disable the outline if desired.

Solution Implementation

See the documents below:

Using Plan Stability

Stored Outline Quick Reference

How to Tune a Query that Cannot be Modified

How to Move Stored Outlines for One Application from One Database to Another

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use dynamic sampling to obtain accurate selectivity estimates

The purpose of dynamic sampling is to improve server performance by determining more accurate estimates for predicate selectivity and statistics for tables and indexes. The statistics for tables and indexes include table block counts, applicable index block counts, table cardinalities, and relevant join column statistics. These more accurate estimates allow the optimizer to produce better performing plans.

You can use dynamic sampling to:

- ***Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation.***
- ***Estimate statistics for tables and relevant indexes without statistics.***
- ***Estimate statistics for tables and relevant indexes whose statistics are too out of date to trust.***

L ***Effort Details***

Low effort; Dynamic sampling can be turned on at the instance, session, or query level.

M ***Risk Details***

Medium risk; Depending on the level, dynamic sampling can consume system resources (I/O bandwidth, CPU) and increase query parse time. Its is best used as an intermediate step to find a better execution plan which can then be hinted or captured with an outline.

Solution Implementation

See the documents below:

When to Use Dynamic Sampling

How to Use Dynamic Sampling to Improve Performance

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Available Indexes are too unselective.

None of the available indexes are selective enough to be useful.

Cause Justification
TBD

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- ***A column in the predicate is not indexed; if it were, a full table scan would be avoided***
- ***The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective***
- ***For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better***

M ***Effort Details***

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M ***Risk Details***

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following

document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: 10g+ : Use the SQL Access Advisor for Index Recommendations

The SQL Access Advisor recommends bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

L *Effort Details*

Low effort; available through Enterprise Manager's GUI or via command line PL/SQL interface.

M *Risk Details*

Medium risk; changes to indexes should be tested in a test system before implementing in production because they may affect many other queries.

Solution Implementation

Please see the following documents:

SQL Access Advisor

Tuning Pack Licensing Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Implicit data type conversion in the query

If the datatypes of two values being compared are different, then Oracle has to implement type conversion on one of the values to enable comparisons to be made. This is called implicit type conversion. Typically this causes problems when developers store numbers in character columns. At runtime oracle is forced to convert one of the values and (due to fixed rules) places a to_number around the indexed character column. Adding any function to an indexed column prevents use of the index. The fact that Oracle has to do this type conversion is an indication of a design problem with the application. Because conversion is performed on EVERY ROW RETRIEVED, this will also result in a performance hit.

Cause Justification
An index exists that satisfies the predicate, but the execution plan's predicate info shows a data type conversion and an "ACCESS" operation.

Solution Identified: Eliminate implicit data type conversion

Eliminating implicit data type conversions will allow the CBO to use an index if its available and potentially improve performance.

M ***Effort Details***

Medium effort. Either the query will need to be re-written to use the same datatype that is stored in the table, or the table and index will need to be modified to reflect the way its used in queries.

M ***Risk Details***

Medium; The risk is low if only the query is changed. If the table and index are modified, other queries may be affected. The change should be thoroughly tested before implementing in production.

Solution Implementation

Related documents:

Avoid Transformed Columns in the WHERE Clause

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: No index has the required columns as leading columns of the index

Oracle usually needs to have the leading columns of the index supplied in the query predicate. In some versions, a "skip scan" access method is possible if an index's leading columns are not in the predicate, but this method is only useful in special cases (where the leading columns have few distinct values).

Cause Justification
TBD

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- ***A column in the predicate is not indexed; if it were, a full table scan would be avoided***
- ***The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective***
- ***For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better***

M ***Effort Details***

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M ***Risk Details***

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following

document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: 10g+ : Use the SQL Access Advisor for Index Recommendations

The SQL Access Advisor recommends bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

L *Effort Details*

Low effort; available through Enterprise Manager's GUI or via command line PL/SQL interface.

M *Risk Details*

Medium risk; changes to indexes should be tested in a test system before implementing in production because they may affect many other queries.

Solution Implementation

Please see the following documents:

SQL Access Advisor

Tuning Pack Licensing Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: A function is used on a column in the query's predicate which prevents the use of an index

A function on a column in the predicate will prevent the use of an index unless a function-based index is available. For example:

use: WHERE a.order_no = b.order_no

rather than:

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

Cause Justification
If the query is performing a full table scan or is using an undesirable index, examine the query's predicate for columns involved in functions.

Solution Identified: Create a function-based index

Function-based indexes provide an efficient mechanism for evaluating statements that contain functions in their WHERE clauses. The value of the expression is computed and stored in the index. When it processes INSERT and UPDATE statements, however, Oracle must still evaluate the function to process the statement. The use of a function-based index will often avoid a full table scan and lead to better performance (when a small number of rows from a rowsource are desired).

L ***Effort Details***

Low; requires the creation of an index using the function used in the query and setting an initialization parameter.

L ***Risk Details***

The function-based index will typically be used by a very small set of queries. There is some risk of a performance regression when performing bulk DML operations due to the application of the index function on each value inserted into the index.

Solution Implementation

Related documents:

Function-based Indexes

Using Function-based Indexes for Performance

When to Use Function-Based Indexes

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Re-write the query to permit the use of an existing index

Rewrite the query to avoid the use of SQL functions in predicate clauses or WHERE clauses. Any expression using a column, such as a function having the column as its argument, causes the optimizer to ignore the possibility of using an index on that column, even a unique index, unless there is a function-based index defined that can be used.

M ***Effort Details***

Medium effort; assuming the query can be modified, it involves rewriting it to avoid the use of functions. Often, this could mean changing the way the data is stored which would involve changes to the underlying table, indexes, and client software.

M ***Risk Details***

Medium risk; if just the query is changed, the risk is low. However, if the query change is accompanied by changes in tables, indexes, and client software, other queries may suffer regressions (although in general, this change will improve the design across the board). An impact analysis should be performed and the changes should be thoroughly tested.

Solution Implementation

See the related documents:

Avoid Transformed Columns in the WHERE Clause

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: The index's cluster factor is too high

When an index is used to access a table's blocks, the optimizer takes into account the cost of accessing the table in addition to the cost of accessing the index. This is computed using something called the cluster factor. The cluster factor is a measure of how closely the rows in the index are ordered relative to the order of the rows in the table. When the rows in the index are ordered closely with those in the table, the cluster factor is low and thus access to the table's blocks will be less expensive since adjacent rows of the index will be found in the table's blocks that are likely already cached. If the rows in the table are not well ordered compared to the order of the index (cluster factor will be high), then access to the table will be much more expensive. The CBO will estimate this cost using the cluster factor; thus, indexes with high cluster factors tend to appear more costly to the CBO and may not be chosen.

Cause Justification

In the 10053 trace, compare the cost of the chosen access path to the index access path that is desired. The index access cost is calculated as follows:

Total index access cost = index cost + table cost

*Index cost = # of Levels + (index selectivity * Index leaf blocks)*
*Table cost = table selectivity * cluster factor*

From the table cost equation, you can see that a large cluster factor will easily dominate the total index access cost and will lead the CBO to chose a different index or a full table scan.

Solution Identified: Load the data in the table in key order

When the table's data is inserted in the same order as one of its indexes (the one of use to the query that needs tuning), it will cost less to access the table based on the rows identified by the index. This will be reflected in the clustering factor and the CBO's cost estimate for using the index.

H *Effort Details*

High effort; it is usually non-trivial to recreate a table or change the insert process so that rows are inserted according to a particular order. Sometimes its not even possible to do because of the nature of the application.

H *Risk Details*

High risk; although the change in the way rows are stored in the table may benefit a certain query using a particular index, it may actually cause other queries to perform worse if they benefited from the former order. An impact analysis should be performed and the application tested prior to implementing in production.

Solution Implementation

The simplest way to reorder the table is to do the following:

*CREATE TABLE new AS SELECT * FROM old ORDER BY b,d;*

Then, rename NEW to OLD.

If the table is loaded via SQLLOAD or a custom loader, it may be possible to change the way the input files are loaded.

Related documents:
Clustering Factor

Tuning I/O-related waits

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use an Index-Organized Table (IOT)

Index-organized tables provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Presence of non-key columns of a row in the B-tree leaf block itself avoids an additional block access. Also, because rows are stored in primary key order, range access by the primary key (or a valid prefix) involves minimum block accesses.

L ***Effort Details***

An IOT is easily created using the CREATE TABLE command. There may be some downtime costs when building the IOT (exporting data from the old table, dropping the old table, creating the new table).

M ***Risk Details***

Medium risk. IOTs are not a substitute for tables in every case. Very large rows can cause the IOT to have deep levels in the B-tree which increase I/Os.
Since the IOT is organized along one key order, it may not provide a competitive cluster factor value for secondary indexes created on it. The value of the IOT should be tested against all of the queries that reference the table.

Solution Implementation

See the documents below:

Benefits of Index-Organized Tables

Managing Index-Organized Tables

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has a hint that is preventing the use of indexes

The query has one of the following hints: INDEX_**, NO_INDEX, FULL, AND_EQUAL. These hints may be set to choose no indexes, or an inferior index the CBO would not have chosen.

In some cases, the FULL hint may be used to suppress the use of all indexes on a table. Existing hints should be viewed with some skepticism when tuning (their presence doesn't mean they were optimal in the first place or that they're still relevant).

Cause Justification
Query contains an access path hint and performs a full table scan or uses an index that does not perform well.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Index hint is being ignored

Index hints may be ignored due to syntax errors in the hints, forgetting to use table aliases, or because it may be semantically impossible to use the index (due to selected join orders or types)

Cause Justification
Hint is specified in the query but execution plan shows it is not being used.

Solution Identified: Correct common problems with hints

There are various reasons why a hint may be ignored. Please see the resources below for guidance.

M ***Effort Details***

Medium effort; The effort to correct a hint problem could range from a simple spelling correction to trying to find a workaround for semantic error that makes the use of a hint impossible.

L ***Risk Details***

Low risk; the hint will only affect the query of interest.

Solution Implementation

See the related documents:

Why is my hint ignored?

How To Avoid Join Method Hints Being Ignored

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

CBO expects the query to return many rows, but only first few rows are actually desired by users

The query will return many rows, but users are only interested in the first few (usually less than 100) rows. This is very common for OLTP applications or web-based applications as opposed to batch or reporting applications.

In this case, the optimizer needs to know that the query will be used in this way in order to generate a better plan.

What to look for

- ***Users are only interested in the first few rows (typically less than 100)***
- ***The query can return many more rows than the first few rows desired***
- ***The optimizer mode is ALL_ROWS or CHOOSE***

Cause Identified: Incorrect OPTIMIZER_MODE being used

The OPTIMIZER_MODE is used to tell the CBO whether the application desires to use all of the rows estimated to be returned by the query or just a small number. This will affect how the CBO approaches the execution plan and how it estimates the costs of access methods and join types.

Cause Justification
OPTIMIZER_MODE is ALL_ROWS or CHOOSE

Look for the SQL in V$SQL and calculate the following:

Avg Rows per Execution = V$SQL.ROWS_PROCESSED / V$SQL.EXECUTIONS

If this value is typically less than 1000 rows, then the optimizer may need to know how many rows are typically desired per execution.

Solution Identified: Use the FIRST_ROWS or FIRST_ROWS_N optimizer mode

The FIRST_ROWS or FIRST_ROWS_K optimizer modes will bias the CBO to look for plans that cost less when a small number of rows are expected. This often produces better plans for OLTP applications because rows are fetched quickly.

L ***Effort Details***

The change involves hints or initialization parameters

M ***Risk Details***

The risk depends on the scope of the change. if just a hint is used, then the risk of impacting other queries is low; whereas if the initialization parameter is used, the impact may be widespread.

Solution Implementation

See the following links for more detail:

FIRST_ROWS(n) hint description

OPTIMIZER_MODE initialization parameter

Fast response optimization (FIRST_ROWS variants)

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Few rows are expected but many rows are returned

Few rows are expected but many rows are returned.

What to look for

Many rows are returned (greater than a few percent of the total rows)

Cause Identified: Cartesian product is occurring due to missing join predicates

Some tables in the query are missing join predicates. When this happens, Oracle will return a cartesian product of the tables resulting in many rows being returned (and generally undesirable results).

Cause Justification

- ***Tables in the FROM clause do not have the proper join clauses. You may need to consult the data model to determine the correct way to join the tables.***
- ***Rows in the result set have many columns***

Solution Identified: Add the appropriate join predicate for the query

Review the join predicates and ensure all required predicates are present

M ***Effort Details***

Medium effort; depending on the complexity of the query and underlying data model, identifying the missing predicate may be easy or difficult.

L ***Risk Details***

Low risk; the additional predicate affects only the query. If not specified properly, the additional predicate may not return the expected values.

Solution Implementation

Requires understanding of the joins and data model to troubleshoot. The solution is simply to add a join predicate.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. ***Many rows expected to be returned by the query (typically decision support apps)***

When many rows are expected, full table scans or full index scans are usually more appropriate than index scans (unique or range). If you know that many rows need to be returned or processed, look for inappropriate use of indexes.

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Full index scan used against a large index

An INDEX FULL SCAN access method is used against a large table (not an INDEX FAST FULL SCAN).

What to look for

- ***The execution plan shows INDEX SCAN on a table (view the plan using SQLTXPLAIN).***
- ***The table is large (see the NUM_ROWS and BLOCKS value in the SQLTXPLAIN report)***

Cause Identified: Optimizer mode or hint set to FIRST_ROWS or FIRST_ROWS_K

When optimizer mode is set to FIRST_ROWS or FIRST_ROWS_K, the optimizer will favor the use of indexes to retrieve rows quickly. This mode will result in a very inefficient plan If many rows are actually desired from the query.

Cause Justification

- ***The optimizer mode may be set in a hint, such as "/*+ FIRST_ROWS_1 */ "***
- ***The optimizer mode may be set in an initialization parameter, such as "OPTIMIZER_MODE=FIRST_ROWS_1". Sometimes a session may have its initialization parameters set through a LOGON trigger - the 10053 trace will show whether this parameter was set or not.***
- ***The TKProf will show the optimizer mode used for each statement***

Solution Identified: Try using the ALL_ROWS hint

If most of the rows from the query are desired (not just the first few that are returned), then the ALL_ROWS hint may allow the CBO to find better execution plans than the FIRST_ROWS_N mode which will produce plans that return rows promptly, but will not be as efficient for retrieving all of the rows.

L ***Effort Details***

Simply add the hint to the query.

L ***Risk Details***

The hint will affect only the query where its applied.

Solution Implementation

The hint syntax is:
/*+ ALL_ROWS */

For reference, see:
ALL_ROWS hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: INDEX FULL SCAN used to avoid a sort operation

The CBO will cost the effort needed to returns rows in order (due to an ORDER BY). Sometimes the estimated cost of using a FULL INDEX SCAN (rows returned in key order) will be cheaper than doing a sort. This estimation may be incorrect and lead to a bad use of the INDEX FULL SCAN operation.

Cause Justification

1. *The execution plan shows the operation, "INDEX FULL SCAN"*
2. *The predicate corresponding to the "INDEX FULL SCAN" operation shows the columns - those columns are the ones used in the ORDER BY clause*

You might be able to quickly confirm if not using this index helps, by modifying the test query to use the "/+ NO_INDEX(...) */" hint.*

Solution Identified: Use the NO_INDEX or ALL_ROWS hint

If the CBO's choice of using a particular index was incorrect (assuming statistics were properly collected), it may be possible to improve the plan by using the following hints:

- *NO_INDEX : suppress the use of the index; this is usually enough to change the plan to avoid the FULL INDEX SCAN*
- *ALL_ROWS : if FIRST_ROWS_N is being used, the use of the index may be attractive to the CBO for returning rows in order quickly. If this isn't really desired (you want ALL of the rows in the shortest time), the ALL_ROWS hint will help the CBO cost the sort better.*

L *Effort Details*

Low effort; if the query can be modified, adding the hint is trivial.

L *Risk Details*

Low risk; only affects the query being tuned.

Solution Implementation

See the documents below:

When will an ORDER BY statement use an Index

NO_INDEX hint

ALL_ROWS hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use PGA_AGGREGATE_TARGET to optimize session memory

The use of an INDEX FULL SCAN operation may be due to a small SORT_AREA_SIZE. The CBO will consider the cost of satisfying the ORDER BY using the INDEX FULL SCAN if there is insufficient PGA memory for sorting. In Oracle8i administrators sized the PGA by carefully adjusting a number of initialization parameters, such as, SORT_AREA_SIZE, HASH_AREA_SIZE, BITMAP_MERGE_AREA_SIZE, and CREATE_BITMAP_AREA_SIZE, etc.

Beginning with 9i, Oracle provides an option to completely automate the management of PGA memory. Administrators merely need to specify the maximum amount of PGA memory available to an instance using a newly introduced initialization parameter PGA_AGGREGATE_TARGET. The database server automatically distributes this memory among various active queries in an intelligent manner so as to ensure maximum performance benefits and the most efficient utilization of memory. Furthermore, Oracle9i can adapt itself to changing workload thus utilizing resources efficiently regardless of the load on the system. The amount of the PGA memory available to an instance can be changed dynamically by altering the value of the PGA_AGGREGATE_TARGET parameter making it possible to add to and remove PGA memory from an active instance online.

L *Effort Details*

The auto-PGA management feature may be activated easily. Some tuning of this will be needed, but it is not difficult.

M *Risk Details*

Medium risk; the change will affect the entire instance, but in general, many queries should see their performance improve as memory is allocated more intelligently to the PGA (as long as the overall amount isn't set too small).

Solution Implementation

Refer to the following documents:

PGA Memory Management

Automatic PGA Memory Management in 9i

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

The query processes or returns many rows

The query either returns many rows or there are steps in the execution plan that must operate on very large tables

What to look for

- *Large number of rows returned.*
- *The execution plan shows full table scan operations on large tables, hash / merge joins, or sorts / sort aggregate to compute some values.*

Cause Identified: Missing filter predicate

A missing filter predicate may cause many more rows to be processed or returned than would otherwise. If the large number of rows is unexpected, a filter predicate may have been forgotten when the query was written.

Cause Justification
Examine the predicate (WHERE clause) to see if any tables are missing a filter condition. Discuss or observe how the data from this query is used by end-users. See if end-users need to filter data on their client or only use a few rows out of the entire result set.

Solution Identified: Review the intent of the query and ensure a predicate isn't missing

If the number of rows returned is unexpectedly high, its possible that part of the predicate is missing. With a smaller number of rows returned, the CBO may choose an index that can retrieve the rows quickly.

L *Effort Details*

Medium effort; usually requires coordination with developers to examine the query

L *Risk Details*

Low risk; the solution applies to the query and won't affect other queries.

Solution Implementation

Review the predicate and ensure it isn't missing a filter or join criteria.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: A large number of rows must be processed for this query

The query must indeed process many rows and must be tuned for processing large amounts of data

Cause Justification
There is a business need for the large volume of data.

Solution Identified: Use parallel execution / parallel DML

If sufficient resources exist, the work can be split using parallel execution (PX) to complete the work in a short time. PX should be considered as a solution after the query has been thoroughly tuned- it shouldn't be the first choice in speeding up a query.

PX works best in cases where a large number of rows must be processed in a timely manner, such as data warehousing or batch operations. OLTP applications with short transactions (a few seconds) are not good candidates for PX.

M *Effort Details*

Medium effort; it is fairly simple to use parallel execution for a query, but some research and testing may need to be done regarding available resources to ensure PX performs well and doesn't exhaust machine resources.

M *Risk Details*

Medium risk; the use of PX may affect all users on the machine and other queries (if a table or index's degree was changed).

Solution Implementation

See the documents below.

Using Parallel Execution

Viewing Parallel Execution with EXPLAIN PLAN

Parallel Execution Hints on Views

Troubleshooting Documents:

Checklist for Performance Problems with Parallel Execution

How To Verify Parallel Execution is running

Why doesn't my query run in parallel?

Restrictions on Parallel DML

Find Parallel Statements which are Candidates for tuning

Why didn't my parallel query use the expected number of slaves?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the

following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Ensure array processing is used

Array processing allows Oracle to process many rows at the same time. It is most commonly used when fetching so that rather than fetch one row at a time and send each one back to the client, Oracle will fetch a set of them and return the set back to the client in one call (usually 10 or more). Array processing is a more efficient way to manage queries that involve many rows and significant performance improvements occur when using it. Large array sizes mean that Oracle can do more work per call to the database and often greatly reduces time spent waiting for context switching, network latency, block pinning, and logical reads.

L *Effort Details*

Low effort; set at the session level in the client.

L *Risk Details*

Low risk; very large array fetch sizes may use a large amount of PGA memory as well as cause a perceived degradation in performance for queries that only need a few rows at a time.

Solution Implementation

Depends on the language used by the client.

SQLPlus Arraysize variable

*Pro*C / C++ : Host Arrays*

*Pro*C / C++ : Using Arrays for Bulk Operations*

PL/SQL : Bulk Binds

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use materialized views and query rewrite to use data that has already been summarized

A materialized view is like a query with a result that is materialized and stored in a table. When a user query is found compatible with the query associated with a materialized view, the user query can be rewritten in terms of the materialized view. This technique improves the execution of the user query, because most of the query result has been pre-computed. The query transformer looks for any materialized views that are compatible with the user query and selects one or more materialized views to rewrite the user query. The use of materialized views to rewrite a query is cost-based. That is, the query is not rewritten if the plan generated without the materialized views has a lower cost than the plan generated with the materialized views.

M ***Effort Details***

Medium effort; creating the materialized view is not difficult, but some considerations must be given to whether and how it should be created and maintained (fast refresh vs. complete, refresh interval, storage requirements).

M ***Risk Details***

Medium risk; the CBO will rewrite a query to use the materialized view instead of accessing the base tables in the query. Some queries that are performing well may change and use the materialized view (generally this should be an improvement). The implementation must be thoroughly tested before deploying to production.

Solution Implementation

See the documents below:

Basic Materialized Views

What are Materialized Views?

Using Materialized Views

Advanced Materialized Views

Basic Query Rewrite

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

- - - - - C. Examine the Join Order and Join Types

The join order can have a huge impact on the performance of a query. The optimal join order is usually one where the fewest rows are returned earliest in the plan. The CBO tries to start join orders with tables that it believes will only return one row. If this estimate is wrong, the wrong table may be chosen and the performance of the query may be impacted.

The choice of join type is also important. Nested loop joins are desirable when just a few rows are desired quickly and join columns are indexed. Hash joins are typically very good at joining large tables, returning many rows, or joining columns that don't have indexes.

Data Required for Analysis

- Source: Execution plan (gathered in "Data Collection", part A)
  - Actual number of rows returned by the query or an execution plan that shows actual and estimated rows per plan step.
  - Estimated number of rows returned by the query ("Estim Card" or similar) from the execution plan
  - Determine if there is a large discrepancy between the actual and estimated rows
- Source: SQLTXPLAIN report (gathered in "Data Collection", part B)
  - Execution plan showing the execution order in the "Exec Order" column (usable only if SQLTXPLAIN's plan matches the one collected in "Data Collection", part A)
- Source: SQLTXPLAIN report, Optimizer Trace section, "Parameters Used by the Optimizer"

1. Join Order Issues

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Incorrect join order; the first few tables being joined return more rows than later tables

The actual rows returned from tables earliest in the join order are much higher than tables joined later. It is very critical to start the join order with the table that will return the fewest rows and then join to the table returning the next fewest rows, and so on. The table returning the fewest rows is not necessarily the smallest table because even a huge table may only return one row when predicates of the query are applied to the table. Conversely, a smaller table may have no predicates applied to it and return many rows.

What to look for

The estimated vs. actual cardinality for one or more tables in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the each table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan steps). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Cause Identified: Incorrect selectivity / cardinality estimate for the first table in a join

The CBO is not estimating the cardinality of the first table in the join order. This could drastically affect the performance of the query because this error will cascade into subsequent join orders and lead to bad choices for the join type and access paths.
The estimate may be bad due to missing statistics (see the Statistics and Parameters section above) or a bad assumption about the predicates of the query having non-overlapping data. Oracle is unable to use statistics to detect overlapping data values in complex predicates without the use of "dynamic sampling".

Cause Justification
The estimated vs. actual cardinality for the first table in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the first table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan step). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use hints to choose a desired join order

Hints may be used for guiding the CBO to the correct join order. There are two hints available:

- ***ORDERED : The join order will be implemented based on the order of the tables in the FROM clause (from left to right, left being the first table in the join order). This gives complete control over the join order and overrides the LEADING hint below.***
- ***LEADING : The join order will start with the specified tables; the rest of the join order will be generated by the CBO. This is useful when you know the plan is improved by just starting with one or two tables and the rest are set properly by the CBO.***

L ***Effort Details***

Low effort; the hint is easily applied to the query. The LEADING hint is the easiest to use as it requires specifying just the start of the join.

Sometimes the CBO will not implement a join order even with a hint. This occurs when the requested join order is semantically impossible to satisfy the query.

L ***Risk Details***

Low risk; the hint will only affect the specific SQL statement.

Solution Implementation

See the reference documents below:

ORDERED hint

LEADING hint

Using Optimizer Hints

Why is my hint ignored?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Incorrect join selectivity / cardinality estimate

The CBO must estimate the cardinality of each join in the plan. The estimate will be used in each subsequent join for costing the various types of joins (and makes a significant impact to the cost of nested loop joins). When this estimate is wrong, the costing of subsequent joins in the plan may be very inaccurate.

Cause Justification
The estimated vs. actual cardinality for one or more tables in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the each table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan steps). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Solution Identified: Use hints to choose a desired join order

Hints may be used for guiding the CBO to the correct join order. There are two hints available:

- ***ORDERED : The join order will be implemented based on the order of the tables in the FROM clause (from left to right, left being the first table in the join order). This gives complete control over the join order and overrides the LEADING hint below.***
- ***LEADING : The join order will start with the specified tables; the rest of the join order will be generated by the CBO. This is useful when you know the plan is improved by just starting with one or two tables and the rest are set properly by the CBO.***

L ***Effort Details***

Low effort; the hint is easily applied to the query. The LEADING hint is the easiest to use as it requires specifying just the start of the join.

Sometimes the CBO will not implement a join order even with a hint. This occurs when the requested join order is semantically impossible to satisfy the query.

L ***Risk Details***

Low risk; the hint will only affect the specific SQL statement.

Solution Implementation

See the reference documents below:

ORDERED hint

LEADING hint

Using Optimizer Hints

Why is my hint ignored?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***

- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use dynamic sampling to obtain accurate selectivity estimates

The purpose of dynamic sampling is to improve server performance by determining more accurate estimates for predicate selectivity and statistics for tables and indexes. The statistics for tables and indexes include table block counts, applicable index block counts, table cardinalities, and relevant join column statistics. These more accurate estimates allow the optimizer to produce better performing plans.

You can use dynamic sampling to:
- *Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation.*
- *Estimate statistics for tables and relevant indexes without statistics.*
- *Estimate statistics for tables and relevant indexes whose statistics are too out of date to trust.*

L *Effort Details*

Low effort; Dynamic sampling can be turned on at the instance, session, or query level.

M *Risk Details*

Medium risk; Depending on the level, dynamic sampling can consume system resources (I/O bandwidth, CPU) and increase query parse time. Its is best used as an intermediate step to find a better execution plan which can then be hinted or captured with an outline.

Solution Implementation

See the documents below:

When to Use Dynamic Sampling

How to Use Dynamic Sampling to Improve Performance

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Initialization parameter "OPTIMIZER_MAX_PERMUTATIONS" is too low for the number of tables in the join

When a large number of tables are joined together, the CBO may not be able to try all permutations because the parameter "OPTIMIZER_MAX_PERMUTATIONS" is too low. Some join orders might have been better than the chosen one if the CBO had been given enough chances to try costing them.

Cause Justification
If the value of "OPTIMIZER_MAX_PERMUTATIONS" is less than the factorial of the number of tables in a join (e.g., if number of tables in the join is 5, 5 factorial is 5*4*3*2*1 or 120), this may be the cause for the bad join order.

Solution Identified: Increase the value of "OPTIMIZER_MAX_PERMUTATIONS" or "OPTIMIZER_SEARCH_LIMIT"

Queries with more than 6 tables (depending on the database version) may require the optimizer to cost more join order permutations than the default settings allow. These additional permutations may yield a lower cost and better performing plan.

Note: in version 10g or later, this parameter is obsolete.

L ***Effort Details***

Low effort; simply an initialization parameter change.

L ***Risk Details***

Low risk; will generally result in better plans and can be tested at the session level. The highest risk is in increasing parse times for queries with more than 6 tables in a join. The increased parse time will be attributed to CPU usage while the optimizer looks for additional join orders.

Solution Implementation

See the links below.

Related documents:
Affect of Number of Tables on Join Order Permutations
Relationship between OPTIMIZER_MAX_PERMUTATIONS and OPTIMIZER_SEARCH_LIMIT

Parameter is obsolete in 10g:
Upgrade Guide, Appendix A

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. Join Type Issues

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Query or rowsource returns many rows, but nested loop join chosen

Nested loop joins enable fast retrieval of a small number of rows. They do not perform well when many rows will be retrieved.

What to look for

NA

Cause Identified: Query has a USE_NL hint that is not appropriate

The query has a USE_NL hint that may have been improperly specified (specifies the wrong inner table) or is now obsolete.

Cause Justification
The query contains a USE_NL hint and performs better without the hint or with a USE_HASH or USE_MERGE hint.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has a USE_NL, FIRST_ROWS, or FIRST_ROWS_K hint that is favoring NL

- *The query has a USE_NL hint that may have been improperly specified (specifies the wrong inner table) or is now obsolete.*
- *The query has a FIRST_ROWS or FIRST_ROWS_K hint that is causing the CBO to favor index access and NL join types*

Remove the hints or avoid the use of the index by adding a NOINDEX() hint; NL joins will usually not be "cost competitive" when indexes are not available to the CBO.

Cause Justification
The query contains a USE_NL hint and performs better without the hint or with a USE_HASH or USE_MERGE hint.

Solution Identified: Remove hints that are influencing the choice of index

*Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).*

L *Effort Details*

Low effort; simply remove the suspected hint, assuming you can modify the query.

L *Risk Details*

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

- - - - - D. Examine Other Operations (parallelism, etc)

1. ***Parallel Execution***

Data Required for Analysis:

- ***Source: Execution plan showing parallel information; collected in "Data Collection", part A.***

Note: This list shows some common observations and causes but is not a complete list. If you do not find a possible cause in this list, you can always open a service request with Oracle to investigate other possible causes. Please see the section below called, "Open a Service Request with Oracle Support Services".

Degree of parallelism is wrong or unexpected

This problem occurs when the requested degree of parallelism is different from what Oracle actually uses for the query.

What to look for

- ***Execution plan shows fewer slaves allocated than requested***
- ***_px_trace reports a smaller degree of parallelism than requested***

Cause Identified: No parallel slaves available for the query

No parallel slaves were available so the query executed in serial mode.

Cause Justification
Event 10392, level 1 shows that the PX coordinator was enable to get enough slaves (at least 2).

Additional Information:
Why didn't my parallel query use the expected number of slaves?

Solution Identified: Additional CPUs are needed

Additional CPUs may be needed to allow enough sessions to use PX. If manual PX tuning is used, you will have to increase the value of PARALLEL_MAX_SERVERS after adding the CPUs.

M ***Effort Details***

Medium effort; adding CPUs may involve downtime depending on the high availability architecture employed.

L ***Risk Details***

Low risk; adding additional CPUs should only improve performance and scalability in this case.

Solution Implementation

Hardware addition, no details provided here.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the

following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Parallelism occurs but is not desired

The query runs best in serial mode or you are trying to avoid parallel mode due to a lack of available resources when the query is executed by many users simultaneously. The query is observed to be executing in parallel.

What to look for

- *Execution plan shows parallel operations*
- *You see parallel slaves associated with your session*

Cause Identified: Hints or configuration settings causing parallel plans

The CBO will attempt to use parallel operations if the following are set or used:

- *Parallel hint: parallel(t1, 4)*
- *ALTER SESSION FORCE PARALLEL*
- *Setting a degree of parallel and/or the number of instances on a table or index in a query*

Cause Justification
Examine the 10053 trace and check the parallel degree for tables and presence of hints in the query. The presence of any of these is justification for this cause.

Additional Information:
Summary of Parallelization Rules

Solution Identified: Remove parallel hints

The statement is executing in parallel due to parallel hints. Removing these hints may allow the statement to run serially.

L *Effort Details*

Low effort; simply remove the hint from the statement.

L *Risk Details*

Low risk, only affects the statement.

Solution Implementation

Remove one or more hints of the type:

- *PARALLEL*
- *PARALLEL_INDEX*
- *PQ_DISTRIBUTE*

If one of the tables has a degree greater than 1, the query may still run in parallel.

Hint information:
Hints for Parallel Execution

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Alter a table or index's degree of parallelism

A table or index in the query has its degree (of parallelism) set higher than 1. This may be one factor causing the query to execute in parallel. If the parallel plan is not performing well, a serial plan may be obtained by changing the degree.

L *Effort Details*

Low effort; the object may be changed with an ALTER command.

M *Risk Details*

Medium risk; other queries may be running in parallel due to the degree setting and will revert to a serial plan. An impact analysis should be performed to determine the effect of this change on other queries.

The ALTER command will invalidate cursors that depend on the table or index and may cause a spike in library cache contention - the change should be done during a period of low activity.

Solution Implementation

See the documents below.

Parallel clause for the CREATE and ALTER TABLE / INDEX statements

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Query executed serially, parallel plan was desired.

The query was not running in parallel. The performance of the query was slower due to serial execution.

What to look for

After executing the query, Check V$PQ_SESSTAT.LAST_QUERY for the statistic "queries parallelized". If it is 0 then the query did not run in parallel.

Cause Identified: No parallel slaves available for the query

No parallel slaves were available so the query executed in serial mode.

Cause Justification
Event 10392, level 1 shows that the PX coordinator was enable to get enough slaves (at least 2).

Additional Information:
Why didn't my parallel query use the expected number of slaves?

Solution Identified: Additional CPUs are needed

Additional CPUs may be needed to allow enough sessions to use PX. If manual PX tuning is used, you will have to increase the value of PARALLEL_MAX_SERVERS after adding the CPUs.

M ***Effort Details***

Medium effort; adding CPUs may involve downtime depending on the high availability architecture employed.

L ***Risk Details***

Low risk; adding additional CPUs should only improve performance and scalability in this case.

Solution Implementation

Hardware addition, no details provided here.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Open a Service Request with Oracle Support Services

If you would like to stop at this point and receive assistance from Oracle Support Services, please do the following:

- Please copy and paste the following into the SR:

  Last Diagnostic Step = Performance_Diagnostic_Guide.QTune.Cause_Determination.
  Data_Analysis

- Enter the problem statement and how the issue has been verified
- Upload into the SR:
  - Extended SQL trace (event 10046 trace) (Identify the Issue, Data Collection)
  - Execution Plan (Determine a Cause, Data Collection, part A)
  - SQLTXPLAIN report (Determine a Cause, Data Collection, part B)
  - Any other data collected (e.g., awrsqlrpt report)
  - (optionally) RDA collection

The more data you collect ahead of time and upload to Oracle, the fewer round trips will be required for this data and the quicker the problem will be resolved.

Click here to log your service request

Cause / Solution Reference

The reference page contains a summary of common causes and solutions.

Optimizer Mode, Statistics, and Initialization Parameters

1. *Optimizer Mode*

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: No statistics gathered (pre10g)

Oracle will default to the RBO when none of the objects in the query have any statistics. In 10g and to some extent in 9.2, Oracle will use the CBO with dynamic sampling and avoid the RBO.

Cause Justification
The execution plan will not display estimated cardinality or cost if RBO is used. In general, RBO will be used in the following cases (see references for more detail):

No "exotic" (post 8.x) features like partitioning, IOTs, parallelism, etc AND:

- ***Pre 9.2.x:***
  - OPTIMIZER_MODE = CHOOSE. Confirm by looking at TKProf, "Optimizer Mode: CHOOSE" for the query
  - and, no statistics on ANY table. Confirm this by looking at each table in SQLTXPLAIN and checking for a NULL value in the "LAST ANALYZED" column
- ***9.2x + :***
  - OPTIMIZER_MODE = CHOOSE or RULE
  - and, dynamic sampling disabled (set to level 0 via hint or parameter)
  - and, no statistics on ANY table. Confirm this by looking at each table in SQLTXPLAIN and checking for a NULL value in the "LAST ANALYZED" column

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very

accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- *Gathering Statistics for the Cost Based Optimizer*
- *Gathering Schema or Database Statistics Automatically - Examples*
- *Histograms: An Overview*
- *Best Practices for automatic statistics collection on 10g*
- *How to check what automatic statistics collection is scheduled on 10g*
- *Statistics Gathering: Frequency and Strategy Guidelines*

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- *Collect and Display System Statistics (CPU and IO) for CBO usage*
- *Scaling the System to Improve CBO optimizer*

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Parameter "optimizer mode" set to RULE

The optimizer_mode parameter will cause Oracle to use the RBO even if statistics are gathered on some or all objects in the query. If a feature such as parallel execution or partitioning is used, then the query will switch over to the CBO.

Cause Justification
The execution plan will not display estimated cardinality or cost if RBO is used. In general, RBO will be used in the following cases (see references for more detail):

No "exotic" (post 8.x) features like partitioning, IOTs, parallelism, etc AND:

- *Pre 9.2.x: optimizer_mode = choose and no statistics on ANY table*
- *9.2x + : optimizer_mode = choose or rule and dynamic sampling disabled*

Solution Identified: Migrate from the RBO to the CBO

The RBO is no longer supported and many features since 8.0 do not use it. The longer term strategy for Oracle installations is to use the CBO. This will ensure the highest level of support and the most efficient plans when using new features.

M ***Effort Details***

Migrating to the CBO can be a high or low effort task depending on the amount of risk you are willing to tolerate. The lowest effort involves simply changing the "OPTIMIZER_MODE" initialization parameter and gathering statistics on objects, but the less risky approaches take more effort to ensure execution plans don't regress.

M ***Risk Details***

Risk depends on the effort placed in localizing the migration (to a single query, session, or application at a time). The highest risk for performance regressions involve using the init.ora "OPTIMIZER_MODE" parameter.

Solution Implementation

The most cautious approach involves adding a hint to the query that is performing poorly. The hint can be "FIRST_ROWS_*" or "ALL_ROWS" depending on the expected number of rows. If the query can't be changed, then it may be possible to limit the change to CBO to just a certain session using a LOGON trigger.

See the following links for more detail:

Moving from RBO to the Query Optimizer

Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques, see the section "Avoiding Plan Regressions after Database Upgrades"

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Incorrect OPTIMIZER_MODE being used

The OPTIMIZER_MODE is used to tell the CBO whether the application desires to use all of the rows estimated to be returned by the query or just a small number. This will affect how the CBO approaches the execution plan and how it estimates the costs of access methods and join types.

Cause Justification
OPTIMIZER_MODE is ALL_ROWS or CHOOSE

Look for the SQL in V$SQL and calculate the following:

Avg Rows per Execution = V$SQL.ROWS_PROCESSED / V$SQL.EXECUTIONS

If this value is typically less than 1000 rows, then the optimizer may need to know how many rows are typically desired per execution.

Solution Identified: Use the FIRST_ROWS or FIRST_ROWS_N optimizer mode

The FIRST_ROWS or FIRST_ROWS_K optimizer modes will bias the CBO to look for plans that cost less when a small number of rows are expected. This often produces better plans for OLTP applications because rows are fetched quickly.

L ***Effort Details***

The change involves hints or initialization parameters

M ***Risk Details***

The risk depends on the scope of the change. if just a hint is used, then the risk of impacting other queries is low; whereas if the initialization parameter is used, the impact may be widespread.

Solution Implementation

See the following links for more detail:

FIRST_ROWS(n) hint description

OPTIMIZER_MODE initialization parameter

Fast response optimization (FIRST_ROWS variants)

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Optimizer mode or hint set to FIRST_ROWS or FIRST_ROWS_K

When optimizer mode is set to FIRST_ROWS or FIRST_ROWS_K, the optimizer will favor the use of indexes to retrieve rows quickly. This mode will result in a very inefficient plan If many rows are actually desired from the query.

Cause Justification

- ***The optimizer mode may be set in a hint, such as "/*+ FIRST_ROWS_1 */ "***
- ***The optimizer mode may be set in an initialization parameter, such as "OPTIMIZER_MODE=FIRST_ROWS_1". Sometimes a session may have its initialization parameters set through a LOGON trigger - the 10053 trace will show whether this parameter was set or not.***
- ***The TKProf will show the optimizer mode used for each statement***

Solution Identified: Try using the ALL_ROWS hint

If most of the rows from the query are desired (not just the first few that are returned), then the ALL_ROWS hint may allow the CBO to find better execution plans than the FIRST_ROWS_N mode which will produce plans that return rows promptly, but will not be as efficient for retrieving all of the rows.

L ***Effort Details***

Simply add the hint to the query.

L ***Risk Details***

The hint will affect only the query where its applied.

Solution Implementation

The hint syntax is:
/*+ ALL_ROWS */

For reference, see:
ALL_ROWS hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. Statistics in General

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Missing or inadequate statistics

- *Missing Statistics*
  - *Statistics were never gathered for tables in the query*
  - *Gathering was not "cascaded" down to indexes*
- *Inadequate sample size*
  - *The sample size was not sufficient to allow the CBO to compute selectivity values accurately*
  - *Histograms not collected on columns involved in the query predicate that have skewed values*

Cause Justification
One or more of the following may justify the need for better statistics collection:

- *Missing table statistics: DBA_TABLES.LAST_ANALYZED is NULL*
- *Missing index statistics: For indexes belonging to each table: DBA_INDEX.LAST_ANALYZED is NULL*
- *Inadequate sample size for tables: DBA_TABLES.SAMPLE_SIZE / # of rows in the table < 5%*
- *Inadequate sample size for indexes: DBA_INDEXES.SAMPLE_SIZE / # of rows in the table < 30%*
- *Histograms not collected: for each table in the query, no rows in DBA_TAB_HISTOGRAMS for the columns having skewed data*
- *Inadequate number of histograms buckets: For each table in the query, less than 255 rows in DBA_TAB_HISTOGRAMS for the columns having skewed data*

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- *ensuring the sample size is large enough*
- *ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)*
- *ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)*
- *if possible, gather global partition stats*

L *Effort Details*

Low effort; easily scripted and executed.

M *Risk Details*

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
```

```
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following

document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Unreasonable table stat values were manually set

Someone either miscalculated or misused DBMS_STATS.SET_STATISTICS

Cause Justification

- *Check the SQLTXPLAIN report, "Table" or "Index" columns, look for the column "User Stats". If this is YES", then the stats were entered directly by users through the DBMS_STATS.SET_*_STATS procedure.*
- *Outrageous statistics values are usually associated with very inaccurate estimated cardinality for the query. You can also examine the statistics by looking at things like the number of rows and comparing them to the actual number of rows in the table (SQLTXPLAIN will list both for each table and index).*

  One approach to confirming this is to export the current statistics on certain objects, gather fresh statistics and compare the two (avoid doing this on a production system).

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- *ensuring the sample size is large enough*
- *ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)*
- *ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)*
- *if possible, gather global partition stats*

L *Effort Details*

Low effort; easily scripted and executed.

M *Risk Details*

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- *Gathering Statistics for the Cost Based Optimizer*
- *Gathering Schema or Database Statistics Automatically - Examples*
- *Histograms: An Overview*
- *Best Practices for automatic statistics collection on 10g*
- *How to check what automatic statistics collection is scheduled on 10g*
- *Statistics Gathering: Frequency and Strategy Guidelines*

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- *Collect and Display System Statistics (CPU and IO) for CBO usage*
- *Scaling the System to Improve CBO optimizer*

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: The tables have undergone extreme DML changes; stats are old

The table has changed dramatically since the stats were collected due to large DML activity.

Cause Justification
You can determine if significant DML activity has occurred against certain tables in the query by looking in the SQLTXPLAIN report and comparing the "Current COUNT" with the "Num Rows". If there is a large difference, the statistics are stale.

You can also look in the DBA_TAB_MODIFICATIONS table to see how much DML has occurred against tables since statistics were last gathered.

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

Gathering Statistics for the Cost Based Optimizer
Gathering Schema or Database Statistics Automatically - Examples
Histograms: An Overview
Best Practices for automatic statistics collection on 10g
How to check what automatic statistics collection is scheduled on 10g
Statistics Gathering: Frequency and Strategy Guidelines

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

Collect and Display System Statistics (CPU and IO) for CBO usage
Scaling the System to Improve CBO optimizer

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

3. Histograms

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Long VARCHAR2 strings are exact up to the 32 character position

The histogram endpoint algorithm for character strings looks at the first 32 characters only. If those characters are exactly the same for many columns, the histograms will not be accurate.

Cause Justification
Observe the histogram endpoint values for the column in the SQLTXPLAIN report under the heading "Table Histograms". Many, if not all, endpoint values will be indistinguishable from each other.

Solution Identified: Use Hints to Get the Desired Plan

Hints will override the CBO's choices (depending on the hint) with a desired change to the execution plan. When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to sub-optimal plans.

M *Effort Details*

Determining the exact hints to arrive at a certain execution plan may be easy or difficult depending on the degree to which the plan needs to be changed.

L *Risk Details*

Hints are applied to a single query so their effect is localized to that query and has no chance of widespread changes (except for widely used views with embedded hints).
For volatile tables, there is a risk that the hint will enforce a plan that is no longer optimal.

Solution Implementation

See the following resources for advice on using hints.

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Data skewing is such that the maximum bucket resolution doesn't help

The histogram buckets must have enough resolution to catch the skewed data. This usually means at least two endpoints must have the same value in order to be detected as a "popular" (skewed) value. Skewing goes undetected when the number of samples in each bucket is so large that truly skewed values are buried inside the bucket.

Cause Justification
Check the following for the column suspected of having skewed data:

1. ***A crude way to confirm there is skewed data is by running this query:***
   SELECT AVG(col1)/((MIN(col1)+MAX(col1))/2) skew_factor FROM table1
   col1, table1 refers to the column/table that has skewed data.
2. ***Examine the output of the query for skewing; when the "skew_factor" is much less than or much greater than 1.0, there is some skewing.***
3. ***Look at the endpoint values for the column in SQLTXPLAIN ("Table Histograms" section) and check if "popular" values are evident in the bucket endpoints (a popular value will have the same endpoint repeated in 2 or more buckets - these are skewed values)***
4. ***If the histogram has 254 buckets and doesn't show any popular buckets, then this cause is justified.***

Solution Identified: Use Hints to Get the Desired Plan

Hints will override the CBO's choices (depending on the hint) with a desired change to the execution plan. When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to sub-optimal plans.

M ***Effort Details***

Determining the exact hints to arrive at a certain execution plan may be easy or difficult depending on the degree to which the plan needs to be changed.

L ***Risk Details***

Hints are applied to a single query so their effect is localized to that query and has no chance of widespread changes (except for widely used views with embedded hints).
For volatile tables, there is a risk that the hint will enforce a plan that is no longer optimal.

Solution Implementation

See the following resources for advice on using hints.

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Manually set histogram statistics to reflect the skewing in the column's data

The histogram will need to be manually defined according to the following method

1. *Find the values where skewing occurs most severely*
2. *Use DBMS_STATS.SET_TABLE_STATS to enter the endpoints and endpoint values representing the skewed data values*

H *Effort Details*

High effort; It will take some effort to determine what the endpoint values should be and then set them using DBMS_STATS.

M *Risk Details*

Medium risk; By altering statistics manually, there is a chance that a miscalculation or mistake may affect many queries in the system. The change may also destabilize good plans.]

Solution Implementation

Details for this solution are not yet available.

Related documents:
Interpreting Histogram Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

4. Parameters

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Parameters causing full table scans and merge/hash joins

The following parameters are known to affect the CBO's cost estimates :
- *optimizer_index_cost_adj set much higher than 100*
- *db_file_multiblock_read_count set too high (greater than 1MB / db_block_size)*
- *optimizer_mode=all_rows*

Cause Justification
Full table scans, merge/hash joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L *Effort Details*

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H *Risk Details*

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Parameters causing index scans and nested loop joins

The following parameters are known to bias the CBO towards index scans and nested loop joins :
- ***optimizer_index_cost_adj set much lower than 100***
- ***db_file_multiblock_read_count set too low (smaller than 1MB / db_block_size)***
- ***optimizer_index_caching set too high***
- ***optimizer_mode=first_rows (or first_rows_N)***

Cause Justification
Index scans and nested loop joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L ***Effort Details***

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H ***Risk Details***

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Init.ora parameters not set for Oracle Applications 11i

Oracle Applications 11i requires certain database initialization parameters to be set according to specific recommendations

Cause Justification
Oracle Applications 11i in use and init.ora parameters not set accordingly

Solution Identified: Set Database Initialization Parameters for Oracle Applications 11i

Oracle Applications 11i have strict requirements for database initialization parameters that must be followed. The use of these parameters generally result in much better performance for the queries used by Oracle Apps. This is a minimum step required when tuning Oracle Apps.

L ***Effort Details***

Low effort; simply set the parameters as required.

L ***Risk Details***

Low risk; these parameters have been extensively tested by Oracle for use with the Apps.

Solution Implementation

See the notes below.

> ***Database Initialization Parameters and Configuration for Oracle Applications 11i***
> ***bde_chk_cbo.sql - Reports Database Initialization Parameters related to an Apps 11i instance***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Initialization parameter "OPTIMIZER_MAX_PERMUTATIONS" is too low for the number of tables in the join

When a large number of tables are joined together, the CBO may not be able to try all permutations because the parameter "OPTIMIZER_MAX_PERMUTATIONS" is too low. Some join orders might have been better than the chosen one if the CBO had been given enough chances to try costing them.

Cause Justification
If the value of "OPTIMIZER_MAX_PERMUTATIONS" is less than the factorial of the number of tables in a join (e.g., if number of tables in the join is 5, 5 factorial is 5*4*3*2*1 or 120), this may be the cause for the bad join order.

Solution Identified: Increase the value of "OPTIMIZER_MAX_PERMUTATIONS" or "OPTIMIZER_SEARCH_LIMIT"

Queries with more than 6 tables (depending on the database version) may require the optimizer to cost more join order permutations than the default settings allow. These additional permutations may yield a lower cost and better performing plan.

Note: in version 10g or later, this parameter is obsolete.

L ***Effort Details***

Low effort; simply an initialization parameter change.

L ***Risk Details***

Low risk; will generally result in better plans and can be tested at the session level. The highest risk is in increasing parse times for queries with more than 6 tables in a join. The increased parse time will be attributed to CPU usage while the optimizer looks for additional join orders.

Solution Implementation

See the links below.

Related documents:
Affect of Number of Tables on Join Order Permutations
Relationship between OPTIMIZER_MAX_PERMUTATIONS and OPTIMIZER_SEARCH_LIMIT

Parameter is obsolete in 10g:
Upgrade Guide, Appendix A

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Access Path

1. Index Not Used

This table lists common causes for cases where the CBO did NOT choose an index (and an index was thought to be optimal).

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Parameters causing full table scans and merge/hash joins

The following parameters are known to affect the CBO's cost estimates :
- *optimizer_index_cost_adj set much higher than 100*
- *db_file_multiblock_read_count set too high (greater than 1MB / db_block_size)*
- *optimizer_mode=all_rows*

Cause Justification
Full table scans, merge/hash joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L *Effort Details*

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H *Risk Details*

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: CBO costs a full table scan cheaper than a series of index range scans

The CBO determines that it is cheaper to do a full table scan than to expand the IN list / OR into separate query blocks where each one uses an index.

Cause Justification
Full table scans in the execution plan instead of a set of index range scans (one per value) with a CONCATENATION operation.

Solution Identified: Implement the USE_CONCAT hint to force the optimizer to use indexes and avoid a full table scan

This hint will force the use of an index (supplied with the hint) instead of using a full table scan (FTS). For certain combinations of IN LIST and OR predicates in queries with tables of a certain size, the use of an index may be far superior to an FTS on a large table.

L ***Effort Details***

Low; simply use the hint in the statement (assuming you can alter the statement).

L ***Risk Details***

Low; will only affect the single statement.

Solution Implementation

See the notes below.

Using the USE_CONCAT hint with IN/OR Statements

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: No index available for columns in the predicate

No indexes have been defined for one or more columns in the query predicate. Oracle only has a full table scan access method available in this case.

Cause Justification

1. ***For each column in the query's WHERE clause, check that there is an index available. In some cases, multiple columns from a table will be in the WHERE clause - ideally, there is an index defined with these columns as the leading columns of the index.***
2. ***Examine the execution plan in the SQLTXPLAIN report and look for predicates using the "FILTER()" function rather than the "ACCESS()" function. Predicates obtained via ACCESS() were obtained using an index (more efficiently and directly), whereas those obtained via FILTER() where obtained by applying a condition to a row source after the data was obtained.***

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- ***A column in the predicate is not indexed; if it were, a full table scan would be avoided***
- ***The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective***
- ***For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better***

M ***Effort Details***

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M ***Risk Details***

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: 10g+ : Use the SQL Access Advisor for Index Recommendations

The SQL Access Advisor recommends bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

L ***Effort Details***

Low effort; available through Enterprise Manager's GUI or via command line PL/SQL interface.

M ***Risk Details***

Medium risk; changes to indexes should be tested in a test system before implementing in production because they may affect many other queries.

Solution Implementation

Please see the following documents:

SQL Access Advisor

Tuning Pack Licensing Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Available Indexes are too unselective.

None of the available indexes are selective enough to be useful.

Cause Justification
TBD

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- ***A column in the predicate is not indexed; if it were, a full table scan would be avoided***
- ***The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective***
- ***For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better***

M ***Effort Details***

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M ***Risk Details***

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: 10g+ : Use the SQL Access Advisor for Index Recommendations

The SQL Access Advisor recommends bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

L *Effort Details*

Low effort; available through Enterprise Manager's GUI or via command line PL/SQL interface.

M *Risk Details*

Medium risk; changes to indexes should be tested in a test system before implementing in production because they may affect many other queries.

Solution Implementation

Please see the following documents:

SQL Access Advisor

Tuning Pack Licensing Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Implicit data type conversion in the query

If the datatypes of two values being compared are different, then Oracle has to implement type conversion on one of the values to enable comparisons to be made. This is called implicit type conversion. Typically this causes problems when developers store numbers in character columns. At runtime oracle is forced to convert one of the values and (due to fixed rules) places a to_number around the indexed character column. Adding any function to an indexed column prevents use of the index. The fact that Oracle has to do this type conversion is an indication of a design problem with the application. Because conversion is performed on EVERY ROW RETRIEVED, this will also result in a performance hit.

Cause Justification
An index exists that satisfies the predicate, but the execution plan's predicate info shows a data type conversion and an "ACCESS" operation.

Solution Identified: Eliminate implicit data type conversion

Eliminating implicit data type conversions will allow the CBO to use an index if its available and potentially improve performance.

M ***Effort Details***

Medium effort. Either the query will need to be re-written to use the same datatype that is stored in the table, or the table and index will need to be modified to reflect the way its used in queries.

M ***Risk Details***

Medium; The risk is low if only the query is changed. If the table and index are modified, other queries may be affected. The change should be thoroughly tested before implementing in production.

Solution Implementation

Related documents:

Avoid Transformed Columns in the WHERE Clause

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: No index has the required columns as leading columns of the index

Oracle usually needs to have the leading columns of the index supplied in the query predicate. In some versions, a "skip scan" access method is possible if an index's leading columns are not in the predicate, but this method is only useful in special cases (where the leading columns have few distinct values).

Cause Justification
TBD

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- ***A column in the predicate is not indexed; if it were, a full table scan would be avoided***
- ***The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective***
- ***For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better***

M ***Effort Details***

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M ***Risk Details***

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following

document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: 10g+ : Use the SQL Access Advisor for Index Recommendations

The SQL Access Advisor recommends bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

L *Effort Details*

Low effort; available through Enterprise Manager's GUI or via command line PL/SQL interface.

M *Risk Details*

Medium risk; changes to indexes should be tested in a test system before implementing in production because they may affect many other queries.

Solution Implementation

Please see the following documents:

SQL Access Advisor

Tuning Pack Licensing Information

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: A function is used on a column in the query's predicate which prevents the use of an index

A function on a column in the predicate will prevent the use of an index unless a function-based index is available. For example:

use: WHERE a.order_no = b.order_no

rather than:

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

Cause Justification
If the query is performing a full table scan or is using an undesirable index, examine the query's predicate for columns involved in functions.

Solution Identified: Create a function-based index

Function-based indexes provide an efficient mechanism for evaluating statements that contain functions in their WHERE clauses. The value of the expression is computed and stored in the index. When it processes INSERT and UPDATE statements, however, Oracle must still evaluate the function to process the statement. The use of a function-based index will often avoid a full table scan and lead to better performance (when a small number of rows from a rowsource are desired).

L ***Effort Details***

Low; requires the creation of an index using the function used in the query and setting an initialization parameter.

L ***Risk Details***

The function-based index will typically be used by a very small set of queries. There is some risk of a performance regression when performing bulk DML operations due to the application of the index function on each value inserted into the index.

Solution Implementation

Related documents:

Function-based Indexes

Using Function-based Indexes for Performance

When to Use Function-Based Indexes

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Re-write the query to permit the use of an existing index

Rewrite the query to avoid the use of SQL functions in predicate clauses or WHERE clauses. Any expression using a column, such as a function having the column as its argument, causes the optimizer to ignore the possibility of using an index on that column, even a unique index, unless there is a function-based index defined that can be used.

M ***Effort Details***

Medium effort; assuming the query can be modified, it involves rewriting it to avoid the use of functions. Often, this could mean changing the way the data is stored which would involve changes to the underlying table, indexes, and client software.

M ***Risk Details***

Medium risk; if just the query is changed, the risk is low. However, if the query change is accompanied by changes in tables, indexes, and client software, other queries may suffer regressions (although in general, this change will improve the design across the board). An impact analysis should be performed and the changes should be thoroughly tested.

Solution Implementation

See the related documents:

Avoid Transformed Columns in the WHERE Clause

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: The index's cluster factor is too high

When an index is used to access a table's blocks, the optimizer takes into account the cost of accessing the table in addition to the cost of accessing the index. This is computed using something called the cluster factor. The cluster factor is a measure of how closely the rows in the index are ordered relative to the order of the rows in the table. When the rows in the index are ordered closely with those in the table, the cluster factor is low and thus access to the table's blocks will be less expensive since adjacent rows of the index will be found in the table's blocks that are likely already cached. If the rows in the table are not well ordered compared to the order of the index (cluster factor will be high), then access to the table will be much more expensive. The CBO will estimate this cost using the cluster factor; thus, indexes with high cluster factors tend to appear more costly to the CBO and may not be chosen.

Cause Justification

In the 10053 trace, compare the cost of the chosen access path to the index access path that is desired. The index access cost is calculated as follows:

Total index access cost = index cost + table cost

*Index cost = # of Levels + (index selectivity * Index leaf blocks)*
*Table cost = table selectivity * cluster factor*

From the table cost equation, you can see that a large cluster factor will easily dominate the total index access cost and will lead the CBO to chose a different index or a full table scan.

Solution Identified: Load the data in the table in key order

When the table's data is inserted in the same order as one of its indexes (the one of use to the query that needs tuning), it will cost less to access the table based on the rows identified by the index. This will be reflected in the clustering factor and the CBO's cost estimate for using the index.

H ***Effort Details***

High effort; it is usually non-trivial to recreate a table or change the insert process so that rows are inserted according to a particular order. Sometimes its not even possible to do because of the nature of the application.

H ***Risk Details***

High risk; although the change in the way rows are stored in the table may benefit a certain query using a particular index, it may actually cause other queries to perform worse if they benefited from the former order. An impact analysis should be performed and the application tested prior to implementing in production.

Solution Implementation

The simplest way to reorder the table is to do the following:

*CREATE TABLE new AS SELECT * FROM old ORDER BY b,d;*

Then, rename NEW to OLD.

If the table is loaded via SQLLOAD or a custom loader, it may be possible to change the way the input files are loaded.

Related documents:
Clustering Factor

Tuning I/O-related waits

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use an Index-Organized Table (IOT)

Index-organized tables provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Presence of non-key columns of a row in the B-tree leaf block itself avoids an additional block access. Also, because rows are stored in primary key order, range access by the primary key (or a valid prefix) involves minimum block accesses.

L ***Effort Details***

An IOT is easily created using the CREATE TABLE command. There may be some downtime costs when building the IOT (exporting data from the old table, dropping the old table, creating the new table).

M ***Risk Details***

Medium risk. IOTs are not a substitute for tables in every case. Very large rows can cause the IOT to have deep levels in the B-tree which increase I/Os.
Since the IOT is organized along one key order, it may not provide a competitive cluster factor value for secondary indexes created on it. The value of the IOT should be tested against all of the queries that reference the table.

Solution Implementation

See the documents below:

Benefits of Index-Organized Tables

Managing Index-Organized Tables

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has a hint that is preventing the use of indexes

The query has one of the following hints: INDEX_**, NO_INDEX, FULL, AND_EQUAL. These hints may be set to choose no indexes, or an inferior index the CBO would not have chosen.

In some cases, the FULL hint may be used to suppress the use of all indexes on a table. Existing hints should be viewed with some skepticism when tuning (their presence doesn't mean they were optimal in the first place or that they're still relevant).

Cause Justification
Query contains an access path hint and performs a full table scan or uses an index that does not perform well.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Index hint is being ignored

Index hints may be ignored due to syntax errors in the hints, forgetting to use table aliases, or because it may be semantically impossible to use the index (due to selected join orders or types)

Cause Justification
Hint is specified in the query but execution plan shows it is not being used.

Solution Identified: Correct common problems with hints

There are various reasons why a hint may be ignored. Please see the resources below for guidance.

M ***Effort Details***

Medium effort; The effort to correct a hint problem could range from a simple spelling correction to trying to find a workaround for semantic error that makes the use of a hint impossible.

L ***Risk Details***

Low risk; the hint will only affect the query of interest.

Solution Implementation

See the related documents:

Why is my hint ignored?

How To Avoid Join Method Hints Being Ignored

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Incorrect OPTIMIZER_MODE being used

The OPTIMIZER_MODE is used to tell the CBO whether the application desires to use all of the rows estimated to be returned by the query or just a small number. This will affect how the CBO approaches the execution plan and how it estimates the costs of access methods and join types.

Cause Justification
OPTIMIZER_MODE is ALL_ROWS or CHOOSE

Look for the SQL in V$SQL and calculate the following:

Avg Rows per Execution = V$SQL.ROWS_PROCESSED / V$SQL.EXECUTIONS

If this value is typically less than 1000 rows, then the optimizer may need to know how many rows are typically desired per execution.

Solution Identified: Use the FIRST_ROWS or FIRST_ROWS_N optimizer mode

The FIRST_ROWS or FIRST_ROWS_K optimizer modes will bias the CBO to look for plans that cost less when a small number of rows are expected. This often produces better plans for OLTP applications because rows are fetched quickly.

L *Effort Details*

The change involves hints or initialization parameters

M *Risk Details*

The risk depends on the scope of the change. if just a hint is used, then the risk of impacting other queries is low; whereas if the initialization parameter is used, the impact may be widespread.

Solution Implementation

See the following links for more detail:

FIRST_ROWS(n) hint description

OPTIMIZER_MODE initialization parameter

Fast response optimization (FIRST_ROWS variants)

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: No Index Available for columns in the predicate

An index is needed to avoid a FTS. The index may never have been created or might have been dropped accidentally.

Cause Justification

1. *For each column in the query's WHERE clause, check that there is an index available. In some cases, multiple columns from a table will be in the WHERE clause - ideally, there is an index defined with these columns as the leading columns of the index.*
2. *Examine the execution plan in the SQLTXPLAIN report and look for predicates using the "FILTER()" function rather than the "ACCESS()" function. Predicates obtained via ACCESS() were obtained using an index (more efficiently and directly), whereas those obtained via FILTER() where obtained by applying a condition to a row source after the data was obtained.*

Solution Identified: Create a new index or re-create an existing index

The performance of the query will greatly improve if few rows are expected and an index may be used to retrieve those rows. The column(s) in the predicate which filter the rows down should be in the leading part of an index. Indexes may need to be created or recreated for the following reasons:

- *A column in the predicate is not indexed; if it were, a full table scan would be avoided*
- *The columns in the predicate are indexed, but the key order (in a composite index) should be rearranged to make the index more selective*
- *For columns that have few distinct values and are not updated frequently, a bitmap (vs. B-tree) index would be better*

M *Effort Details*

Medium; Simply drop and recreate an index or create a new index. However, the application may need to be down to avoid affecting it if an existing index must be dropped and recreated.

M *Risk Details*

Medium; the recreated index may change some execution plans since it will be slightly bigger and its contribution to the cost of a query will be larger. On the other hand, the created index may be more compact than the one it replaces since it will not have many deleted keys in its leaf blocks.

A newly created index may cause other query's plans to change if it is seen as a lower cost alternative (typically this should result in better performance).

The DDL to create or recreate the index may cause some cursors to be invalidated which might lead to a spike in library cache latch contention; ideally, the DDL is issued during a time of low activity.

This change should be thoroughly tested before implementing on a production system.

Solution Implementation

If an index would reduce the time to retrieve rows for the query, its best to review existing indexes and see if any of them can be rebuilt with additional column(s) that would cause the index to be used. Otherwise, a new index may have to be created. Please note that adding indexes will add some overhead during DML operations and should be created judiciously.

See the links below for information on creating indexes.

10g+ : Consult the SQL Access Advisor

Understanding Index Performance

Diagnosing Why a Query is Not Using an Index

Using Indexes and Clusters

SQL Reference: CREATE INDEX

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Missing filter predicate

A missing filter predicate may cause many more rows to be processed or returned than would otherwise. If the large number of rows is unexpected, a filter predicate may have been forgotten when the query was written.

Cause Justification
Examine the predicate (WHERE clause) to see if any tables are missing a filter condition. Discuss or observe how the data from this query is used by end-users. See if end-users need to filter data on their client or only use a few rows out of the entire result set.

Solution Identified: Review the intent of the query and ensure a predicate isn't missing

If the number of rows returned is unexpectedly high, its possible that part of the predicate is missing. With a smaller number of rows returned, the CBO may choose an index that can retrieve the rows quickly.

L ***Effort Details***

Medium effort; usually requires coordination with developers to examine the query

L ***Risk Details***

Low risk; the solution applies to the query and won't affect other queries.

Solution Implementation

Review the predicate and ensure it isn't missing a filter or join criteria.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. Full Table Scan is Used

This table lists common causes for cases where the CBO chose an FTS (and the use of FTS seems to be sub-optimal).
This is related to the cause section above, "Index was NOT used", where you will also find causes when FTS was chosen over an index path. However, the causes in this section are specific to the undesired use of FTS.

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Parameters causing full table scans and merge/hash joins

The following parameters are known to affect the CBO's cost estimates :
- ***optimizer_index_cost_adj set much higher than 100***
- ***db_file_multiblock_read_count set too high (greater than 1MB / db_block_size)***
- ***optimizer_mode=all_rows***

Cause Justification
Full table scans, merge/hash joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L ***Effort Details***

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H ***Risk Details***

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: A large number of rows must be processed for this query

The query must indeed process many rows and must be tuned for processing large amounts of data

Cause Justification
There is a business need for the large volume of data.

Solution Identified: Use parallel execution / parallel DML

If sufficient resources exist, the work can be split using parallel execution (PX) to complete the work in a short time. PX should be considered as a solution after the query has been thoroughly tuned- it shouldn't be the first choice in speeding up a query.

PX works best in cases where a large number of rows must be processed in a timely manner, such as data warehousing or batch operations. OLTP applications with short transactions (a few seconds) are not good candidates for PX.

M ***Effort Details***

Medium effort; it is fairly simple to use parallel execution for a query, but some research and testing may need to be done regarding available resources to ensure PX performs well and doesn't exhaust machine resources.

M ***Risk Details***

Medium risk; the use of PX may affect all users on the machine and other queries (if a table or index's degree was changed).

Solution Implementation

See the documents below.

Using Parallel Execution

Viewing Parallel Execution with EXPLAIN PLAN

Parallel Execution Hints on Views

Troubleshooting Documents:

Checklist for Performance Problems with Parallel Execution

How To Verify Parallel Execution is running

Why doesn't my query run in parallel?

Restrictions on Parallel DML

Find Parallel Statements which are Candidates for tuning

Why didn't my parallel query use the expected number of slaves?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Ensure array processing is used

Array processing allows Oracle to process many rows at the same time. It is most commonly used when fetching so that rather than fetch one row at a time and send each one back to the client, Oracle will fetch a set of them and return the set back to the client in one call (usually 10 or more). Array processing is a more efficient way to manage queries that involve many rows and significant performance improvements occur when using it. Large array sizes mean that Oracle can do more work per call to the database and often greatly reduces time spent waiting for context switching, network latency, block pinning, and logical reads.

L ***Effort Details***

Low effort; set at the session level in the client.

L ***Risk Details***

Low risk; very large array fetch sizes may use a large amount of PGA memory as well as cause a perceived degradation in performance for queries that only need a few rows at a time.

Solution Implementation

Depends on the language used by the client.

SQLPlus Arraysize variable

Pro*C / C++ : Host Arrays

Pro*C / C++ : Using Arrays for Bulk Operations

PL/SQL : Bulk Binds

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use materialized views and query rewrite to use data that has already been summarized

A materialized view is like a query with a result that is materialized and stored in a table. When a user query is found compatible with the query associated with a materialized view, the user query can be rewritten in terms of the materialized view. This technique improves the execution of the user query, because most of the query result has been pre-computed. The query transformer looks for any materialized views that are compatible with the user query and selects one or more materialized views to rewrite the user query. The use of materialized views to rewrite a query is cost-based. That is, the query is not rewritten if the plan generated without the materialized views has a lower cost than the plan generated with the materialized views.

M ***Effort Details***

Medium effort; creating the materialized view is not difficult, but some considerations must be given to whether and how it should be created and maintained (fast refresh vs. complete, refresh interval, storage requirements).

M ***Risk Details***

Medium risk; the CBO will rewrite a query to use the materialized view instead of accessing the base tables in the query. Some queries that are performing well may change and use the materialized view (generally this should be an improvement). The implementation must be thoroughly tested before deploying to production.

Solution Implementation

See the documents below:

Basic Materialized Views

What are Materialized Views?

Using Materialized Views

Advanced Materialized Views

Basic Query Rewrite

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

3. Full Table Scan is Not Used

This table lists common causes for cases where the CBO did NOT choose an FTS (and the use of FTS would have been optimal).

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: INDEX FULL SCAN used to avoid a sort operation

The CBO will cost the effort needed to returns rows in order (due to an ORDER BY). Sometimes the estimated cost of using a FULL INDEX SCAN (rows returned in key order) will be cheaper than doing a sort. This estimation may be incorrect and lead to a bad use of the INDEX FULL SCAN operation.

Cause Justification

1. *The execution plan shows the operation, "INDEX FULL SCAN"*
2. *The predicate corresponding to the "INDEX FULL SCAN" operation shows the columns - those columns are the ones used in the ORDER BY clause*

You might be able to quickly confirm if not using this index helps, by modifying the test query to use the "/+ NO_INDEX(...) */" hint.*

Solution Identified: Use the NO_INDEX or ALL_ROWS hint

If the CBO's choice of using a particular index was incorrect (assuming statistics were properly collected), it may be possible to improve the plan by using the following hints:

- *NO_INDEX : suppress the use of the index; this is usually enough to change the plan to avoid the FULL INDEX SCAN*
- *ALL_ROWS : if FIRST_ROWS_N is being used, the use of the index may be attractive to the CBO for returning rows in order quickly. If this isn't really desired (you want ALL of the rows in the shortest time), the ALL_ROWS hint will help the CBO cost the sort better.*

L *Effort Details*

Low effort; if the query can be modified, adding the hint is trivial.

L *Risk Details*

Low risk; only affects the query being tuned.

Solution Implementation

See the documents below:

When will an ORDER BY statement use an Index

NO_INDEX hint

ALL_ROWS hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use PGA_AGGREGATE_TARGET to optimize session memory

The use of an INDEX FULL SCAN operation may be due to a small SORT_AREA_SIZE. The CBO will consider the cost of satisfying the ORDER BY using the INDEX FULL SCAN if there is insufficient PGA memory for sorting. In Oracle8i administrators sized the PGA by carefully adjusting a number of initialization parameters, such as, SORT_AREA_SIZE, HASH_AREA_SIZE, BITMAP_MERGE_AREA_SIZE, and CREATE_BITMAP_AREA_SIZE, etc.

Beginning with 9i, Oracle provides an option to completely automate the management of PGA memory. Administrators merely need to specify the maximum amount of PGA memory available to an instance using a newly introduced initialization parameter PGA_AGGREGATE_TARGET. The database server automatically distributes this memory among various active queries in an intelligent manner so as to ensure maximum performance benefits and the most efficient utilization of memory. Furthermore, Oracle9i can adapt itself to changing workload thus utilizing resources efficiently regardless of the load on the system. The amount of the PGA memory available to an instance can be changed dynamically by altering the value of the PGA_AGGREGATE_TARGET parameter making it possible to add to and remove PGA memory from an active instance online.

L *Effort Details*

The auto-PGA management feature may be activated easily. Some tuning of this will be needed, but it is not difficult.

M *Risk Details*

Medium risk; the change will affect the entire instance, but in general, many queries should see their performance improve as memory is allocated more intelligently to the PGA (as long as the overall amount isn't set too small).

Solution Implementation

Refer to the following documents:

PGA Memory Management

Automatic PGA Memory Management in 9i

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Parameters causing index scans and nested loop joins

The following parameters are known to bias the CBO towards index scans and nested loop joins :

- ***optimizer_index_cost_adj set much lower than 100***
- ***db_file_multiblock_read_count set too low (smaller than 1MB / db_block_size)***
- ***optimizer_index_caching set too high***
- ***optimizer_mode=first_rows (or first_rows_N)***

Cause Justification
Index scans and nested loop joins occurring and above parameters not set to default values.

Solution Identified: Reset parameters to default settings

Changing certain non-default initialization parameter settings could improve the query. However, this should be done in a session (rather than at the database level in the init.ora or spfile) first and you must consider the impact of this change on other queries. If the parameter cannot be changed due to the effect on other queries, you may need to use outlines or hints to improve the plan.

L ***Effort Details***

Simple change of initialization parameter(s). However, care should be taken to test the effects of this change and these tests may take considerable effort.

H ***Risk Details***

Initialization parameter changes have the potential of affecting many other queries in the database, so the risk may be high. Risk can be mitigated through testing on a test system or in a session, if possible.

Solution Implementation

Various notes describe the important parameters that influence the CBO, see the links below:

TBW: Parameters affecting the optimizer and their default values

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Optimizer mode or hint set to FIRST_ROWS or FIRST_ROWS_K

When optimizer mode is set to FIRST_ROWS or FIRST_ROWS_K, the optimizer will favor the use of indexes to retrieve rows quickly. This mode will result in a very inefficient plan If many rows are actually desired from the query.

Cause Justification

- ***The optimizer mode may be set in a hint, such as "/*+ FIRST_ROWS_1 */ "***
- ***The optimizer mode may be set in an initialization parameter, such as "OPTIMIZER_MODE=FIRST_ROWS_1". Sometimes a session may have its initialization parameters set through a LOGON trigger - the 10053 trace will show whether this parameter was set or not.***
- ***The TKProf will show the optimizer mode used for each statement***

Solution Identified: Try using the ALL_ROWS hint

If most of the rows from the query are desired (not just the first few that are returned), then the ALL_ROWS hint may allow the CBO to find better execution plans than the FIRST_ROWS_N mode which will produce plans that return rows promptly, but will not be as efficient for retrieving all of the rows.

L ***Effort Details***

Simply add the hint to the query.

L ***Risk Details***

The hint will affect only the query where its applied.

Solution Implementation

The hint syntax is:
/*+ ALL_ROWS */

For reference, see:
ALL_ROWS hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has a USE_NL hint that is not appropriate

The query has a USE_NL hint that may have been improperly specified (specifies the wrong inner table) or is now obsolete.

Cause Justification
The query contains a USE_NL hint and performs better without the hint or with a USE_HASH or USE_MERGE hint.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has a USE_NL, FIRST_ROWS, or FIRST_ROWS_K hint that is favoring NL

- ***The query has a USE_NL hint that may have been improperly specified (specifies the wrong inner table) or is now obsolete.***
- ***The query has a FIRST_ROWS or FIRST_ROWS_K hint that is causing the CBO to favor index access and NL join types***

Remove the hints or avoid the use of the index by adding a NOINDEX() hint; NL joins will usually not be "cost competitive" when indexes are not available to the CBO.

Cause Justification
The query contains a USE_NL hint and performs better without the hint or with a USE_HASH or USE_MERGE hint.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: No parallel slaves available for the query

No parallel slaves were available so the query executed in serial mode.

Cause Justification
Event 10392, level 1 shows that the PX coordinator was enable to get enough slaves (at least 2).

Additional Information:
Why didn't my parallel query use the expected number of slaves?

Solution Identified: Additional CPUs are needed

Additional CPUs may be needed to allow enough sessions to use PX. If manual PX tuning is used, you will have to increase the value of PARALLEL_MAX_SERVERS after adding the CPUs.

M ***Effort Details***

Medium effort; adding CPUs may involve downtime depending on the high availability architecture employed.

L ***Risk Details***

Low risk; adding additional CPUs should only improve performance and scalability in this case.

Solution Implementation

Hardware addition, no details provided here.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Incorrect Selectivity or Cardinality

1. ***Filter predicates***

These causes and solutions apply to incorrect selectivity or cardinality estimates for filter predicates.

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Incorrect selectivity estimate

The CBO needs additional information for estimating the selectivity of the query (in maybe just one of the plan steps). Usually this is due to predicate clauses that have some correlation. The CBO assumes that filter predicates are independent of each other; when ANDed, these predicates reduce the number of rows returned (increased selectivity). However, when these predicates are not independent (e.g., a query that filtered on the city name and postal code), more rows are returned than the CBO estimates. This leads to inaccurate cost estimates and inefficient plans.

Cause Justification
The estimated vs. actual cardinality for the query or for individual plan steps differ significantly.

Solution Identified: Use Hints to Get the Desired Plan

Hints will override the CBO's choices (depending on the hint) with a desired change to the execution plan. When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to sub-optimal plans.

M ***Effort Details***

Determining the exact hints to arrive at a certain execution plan may be easy or difficult depending on the degree to which the plan needs to be changed.

L ***Risk Details***

Hints are applied to a single query so their effect is localized to that query and has no chance of widespread changes (except for widely used views with embedded hints).
For volatile tables, there is a risk that the hint will enforce a plan that is no longer optimal.

Solution Implementation

See the following resources for advice on using hints.

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use Plan Stability to Set the Desired Execution Plan

Plan stability preserves execution plans in stored outlines. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If the use of the outline is enabled for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints.

The performance of a statement is improved without modifying the statement (assuming an outline can be created with the hints that generate a better plan).

M ***Effort Details***

Medium effort; Depending on the circumstance, sometimes an outline for a query is easily generated and used. The easiest case is when a better plan is generated simply by changing an initialization parameter and an outline is captured for the query.
In other cases, it is difficult to obtain the plan and capture it for the outline.

L ***Risk Details***

Low risk; the outline will only affect the associated query. The outline should be associated with a category that enables one to easily disable the outline if desired.

Solution Implementation

See the documents below:

Using Plan Stability

Stored Outline Quick Reference

How to Tune a Query that Cannot be Modified

How to Move Stored Outlines for One Application from One Database to Another

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use dynamic sampling to obtain accurate selectivity estimates

The purpose of dynamic sampling is to improve server performance by determining more accurate estimates for predicate selectivity and statistics for tables and indexes. The statistics for tables and indexes include table block counts, applicable index block counts, table cardinalities, and relevant join column statistics. These more accurate estimates allow the optimizer to produce better performing plans.

You can use dynamic sampling to:

- ***Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation.***
- ***Estimate statistics for tables and relevant indexes without statistics.***
- ***Estimate statistics for tables and relevant indexes whose statistics are too out of date to trust.***

L ***Effort Details***

Low effort; Dynamic sampling can be turned on at the instance, session, or query level.

M ***Risk Details***

Medium risk; Depending on the level, dynamic sampling can consume system resources (I/O bandwidth, CPU) and increase query parse time. Its is best used as an intermediate step to find a better execution plan which can then be hinted or captured with an outline.

Solution Implementation

See the documents below:

When to Use Dynamic Sampling

How to Use Dynamic Sampling to Improve Performance

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. Joins

These causes and solutions apply to incorrect selectivity or cardinality estimates for joins.

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Incorrect selectivity / cardinality estimate for the first table in a join

The CBO is not estimating the cardinality of the first table in the join order. This could drastically affect the performance of the query because this error will cascade into subsequent join orders and lead to bad choices for the join type and access paths.
The estimate may be bad due to missing statistics (see the Statistics and Parameters section above) or a bad assumption about the predicates of the query having non-overlapping data. Oracle is unable to use statistics to detect overlapping data values in complex predicates without the use of "dynamic sampling".

Cause Justification
The estimated vs. actual cardinality for the first table in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the first table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan step). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
```

```
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following

document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use hints to choose a desired join order

Hints may be used for guiding the CBO to the correct join order. There are two hints available:

- *ORDERED : The join order will be implemented based on the order of the tables in the FROM clause (from left to right, left being the first table in the join order). This gives complete control over the join order and overrides the LEADING hint below.*
- *LEADING : The join order will start with the specified tables; the rest of the join order will be generated by the CBO. This is useful when you know the plan is improved by just starting with one or two tables and the rest are set properly by the CBO.*

L *Effort Details*

Low effort; the hint is easily applied to the query. The LEADING hint is the easiest to use as it requires specifying just the start of the join.

Sometimes the CBO will not implement a join order even with a hint. This occurs when the requested join order is semantically impossible to satisfy the query.

L *Risk Details*

Low risk; the hint will only affect the specific SQL statement.

Solution Implementation

See the reference documents below:

ORDERED hint

LEADING hint

Using Optimizer Hints

Why is my hint ignored?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Incorrect join selectivity / cardinality estimate

The CBO must estimate the cardinality of each join in the plan. The estimate will be used in each subsequent join for costing the various types of joins (and makes a significant impact to the cost of nested loop joins). When this estimate is wrong, the costing of subsequent joins in the plan may be very inaccurate.

Cause Justification
The estimated vs. actual cardinality for one or more tables in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the each table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan steps). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Solution Identified: Use hints to choose a desired join order

Hints may be used for guiding the CBO to the correct join order. There are two hints available:

- ***ORDERED : The join order will be implemented based on the order of the tables in the FROM clause (from left to right, left being the first table in the join order). This gives complete control over the join order and overrides the LEADING hint below.***
- ***LEADING : The join order will start with the specified tables; the rest of the join order will be generated by the CBO. This is useful when you know the plan is improved by just starting with one or two tables and the rest are set properly by the CBO.***

L ***Effort Details***

Low effort; the hint is easily applied to the query. The LEADING hint is the easiest to use as it requires specifying just the start of the join.

Sometimes the CBO will not implement a join order even with a hint. This occurs when the requested join order is semantically impossible to satisfy the query.

L ***Risk Details***

Low risk; the hint will only affect the specific SQL statement.

Solution Implementation

See the reference documents below:

ORDERED hint

LEADING hint

Using Optimizer Hints

Why is my hint ignored?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***

- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use dynamic sampling to obtain accurate selectivity estimates

The purpose of dynamic sampling is to improve server performance by determining more accurate estimates for predicate selectivity and statistics for tables and indexes. The statistics for tables and indexes include table block counts, applicable index block counts, table cardinalities, and relevant join column statistics. These more accurate estimates allow the optimizer to produce better performing plans.

You can use dynamic sampling to:
- *Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation.*
- *Estimate statistics for tables and relevant indexes without statistics.*
- *Estimate statistics for tables and relevant indexes whose statistics are too out of date to trust.*

L *Effort Details*

Low effort; Dynamic sampling can be turned on at the instance, session, or query level.

M *Risk Details*

Medium risk; Depending on the level, dynamic sampling can consume system resources (I/O bandwidth, CPU) and increase query parse time. Its is best used as an intermediate step to find a better execution plan which can then be hinted or captured with an outline.

Solution Implementation

See the documents below:

When to Use Dynamic Sampling

How to Use Dynamic Sampling to Improve Performance

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

1. Predicates

The causes and solutions to problems with predicates are listed here.

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Cartesian product is occurring due to missing join predicates

Some tables in the query are missing join predicates. When this happens, Oracle will return a cartesian product of the tables resulting in many rows being returned (and generally undesirable results).

Cause Justification

- ***Tables in the FROM clause do not have the proper join clauses. You may need to consult the data model to determine the correct way to join the tables.***
- ***Rows in the result set have many columns***

Solution Identified: Add the appropriate join predicate for the query

Review the join predicates and ensure all required predicates are present

M ***Effort Details***

Medium effort; depending on the complexity of the query and underlying data model, identifying the missing predicate may be easy or difficult.

L ***Risk Details***

Low risk; the additional predicate affects only the query. If not specified properly, the additional predicate may not return the expected values.

Solution Implementation

Requires understanding of the joins and data model to troubleshoot. The solution is simply to add a join predicate.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Missing filter predicate

A missing filter predicate may cause many more rows to be processed or returned than would otherwise. If the large number of rows is unexpected, a filter predicate may have been forgotten when the query was written.

Cause Justification
Examine the predicate (WHERE clause) to see if any tables are missing a filter condition. Discuss or observe how the data from this query is used by end-users. See if end-users need to filter data on their client or only use a few rows out of the entire result set.

Solution Identified: Review the intent of the query and ensure a predicate isn't missing

If the number of rows returned is unexpectedly high, its possible that part of the predicate is missing. With a smaller number of rows returned, the CBO may choose an index that can retrieve the rows quickly.

L ***Effort Details***

Medium effort; usually requires coordination with developers to examine the query

L ***Risk Details***

Low risk; the solution applies to the query and won't affect other queries.

Solution Implementation

Review the predicate and ensure it isn't missing a filter or join criteria.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Index needed for columns used with fine grained access control

The use of FGAC will cause additional predicates to be generated. These predicates may be difficult for the CBO to optimize or they may require the use of new indexes.

Cause Justification

1. ***Query performance improves when FGAC is not used***
2. ***Manually adding the FGAC-generated predicates to the base query will reproduce the problem.***
3. ***Event 10730 shows the predicate added by FGAC and it matches the predicate seen in the execution plan's access and filter predicates***

Solution Identified: Create an index on the columns involved in the FGAC

FGAC introduces additional predicates that may require an index on the relevant columns. In some cases, a function-based index may be needed.

L ***Effort Details***

Low effort; just add an index or recreate an index to include the columns used in the security policy.

L ***Risk Details***

The index should have little negative impact except where a table already has many indexes and this index causes DML to take longer than desired.

Solution Implementation

TBD

TBD

TBD

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Bug 5195882

Queries In FGAC Using Full Table Scan Instead Of Index Access

This bug prevents view merging when PL/SQL functions and views are involved - which is common when FGAC is used. The inability to merge views leads to bad execution plans.

Cause Justification

1. ***Query performance improves when FGAC is not used***
2. ***Manually adding the FGAC-generated predicates to the base query will reproduce the problem.***
3. ***Event 10730 shows the predicate added by FGAC and it matches the predicate seen in the execution plan's access and filter predicates***

Solution Identified: Apply patch for bug 5195882 or use the workaround

Patch and workaround available.

M ***Effort Details***

Requires a patch application. The workaround is lower effort, but side effects are unknown.

M ***Risk Details***

If applying the one-off patch, it carries the risk typically associated with one-off patches.
Patchset 10.2.0.3 has the fix for this bug and is lower risk since patchsets are rigorously tested.

Solution Implementation

Contact Oracle Support Services for the patch.

Workaround:
Set optimizer_secure_view_merging=false

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Join Order and Type

1. ***Join Order***

The causes and solutions to problems with join order are listed here.

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Incorrect selectivity / cardinality estimate for the first table in a join

The CBO is not estimating the cardinality of the first table in the join order. This could drastically affect the performance of the query because this error will cascade into subsequent join orders and lead to bad choices for the join type and access paths.
The estimate may be bad due to missing statistics (see the Statistics and Parameters section above) or a bad assumption about the predicates of the query having non-overlapping data. Oracle is unable to use statistics to detect overlapping data values in complex predicates without the use of "dynamic sampling".

Cause Justification
The estimated vs. actual cardinality for the first table in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the first table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan step). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Solution Identified: Gather statistics properly

The CBO will generate better plans when it has accurate statistics for tables and indexes. In general, the main aspects to focus on are:

- ***ensuring the sample size is large enough***
- ***ensuring all objects (tables and indexes) have stats gathered (CASCADE parameter)***
- ***ensuring that any columns with skewed data have histograms collected, and at sufficient resolution (METHOD_OPT parameter)***
- ***if possible, gather global partition stats***

L ***Effort Details***

Low effort; easily scripted and executed.

M ***Risk Details***

Medium risk; Gathering new stats may change some execution plans for the worse, but its more likely plans will improve. Gathering stats will invalidate cursors in the shared pool - this should be done only during periods of low activity in the database.

Solution Implementation

In general, you can use the following to gather stats for a single table and its indexes as a starting point. You may need to alter this as you find out that histograms are needed or sample sizes need to be explicitly set:

Oracle 9.0.x - 9.2.x

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE ,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1' );
```

Oracle 10g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
estimate_percent =>
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Oracle 11g:

```
exec DBMS_STATS.GATHER_TABLE_STATS(
tabname => '  Table_name  '
ownname => NULL,
cascade => 'TRUE',
method_opt => 'FOR ALL COLUMNS SIZE 1');
```

Note: replace ' Table_name ' with the name of the table to gather statistics for. "" means that you should strive for 100 percent sample sizes if possible, but if time doesn't permit this, then use values that are as high as possible to gather an accurate sample while finishing the stats gathering in a reasonable time (values like 35% can yield good results). Oracle 9.2 used 100% sample sizes by default; 11g uses a new very accurate algorithm that takes about the same time as a 10% sample yet yields results close to a 100% sample. So for 11g, we recommend leaving the estimate_percent to default to AUTO.

Review the following resources for guidance on properly gathering statistics:

- ***Gathering Statistics for the Cost Based Optimizer***
- ***Gathering Schema or Database Statistics Automatically - Examples***
- ***Histograms: An Overview***
- ***Best Practices for automatic statistics collection on 10g***
- ***How to check what automatic statistics collection is scheduled on 10g***
- ***Statistics Gathering: Frequency and Strategy Guidelines***

In Oracle 9.2 and later versions, system statistics may improve the accuracy of the CBO's estimates by providing the CBO with CPU cost estimates in addition to the normal I/O cost estimates.

- ***Collect and Display System Statistics (CPU and IO) for CBO usage***
- ***Scaling the System to Improve CBO optimizer***

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use hints to choose a desired join order

Hints may be used for guiding the CBO to the correct join order. There are two hints available:

- ***ORDERED : The join order will be implemented based on the order of the tables in the FROM clause (from left to right, left being the first table in the join order). This gives complete control over the join order and overrides the LEADING hint below.***
- ***LEADING : The join order will start with the specified tables; the rest of the join order will be generated by the CBO. This is useful when you know the plan is improved by just starting with one or two tables and the rest are set properly by the CBO.***

L ***Effort Details***

Low effort; the hint is easily applied to the query. The LEADING hint is the easiest to use as it requires specifying just the start of the join.

Sometimes the CBO will not implement a join order even with a hint. This occurs when the requested join order is semantically impossible to satisfy the query.

L ***Risk Details***

Low risk; the hint will only affect the specific SQL statement.

Solution Implementation

See the reference documents below:

ORDERED hint

LEADING hint

Using Optimizer Hints

Why is my hint ignored?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Incorrect join selectivity / cardinality estimate

The CBO must estimate the cardinality of each join in the plan. The estimate will be used in each subsequent join for costing the various types of joins (and makes a significant impact to the cost of nested loop joins). When this estimate is wrong, the costing of subsequent joins in the plan may be very inaccurate.

Cause Justification
The estimated vs. actual cardinality for one or more tables in the join order differs significantly. This can be observed by looking at the following:

Estimated cardinality: Look at the execution plan (in SQLTXPLAIN) and find the "Estim Cardinality" column corresponding to the each table in the join order (see the column "Exec Order" to see where to start reading the execution plan)

Actual cardinality: Check the runtime execution plan in the TKProf for the query (for the same plan steps). If you collected the plan from V$SQL using the script in the "Data Collection" section, simply compare the estimated and actual columns.

Solution Identified: Use hints to choose a desired join order

Hints may be used for guiding the CBO to the correct join order. There are two hints available:

- ***ORDERED : The join order will be implemented based on the order of the tables in the FROM clause (from left to right, left being the first table in the join order). This gives complete control over the join order and overrides the LEADING hint below.***
- ***LEADING : The join order will start with the specified tables; the rest of the join order will be generated by the CBO. This is useful when you know the plan is improved by just starting with one or two tables and the rest are set properly by the CBO.***

L ***Effort Details***

Low effort; the hint is easily applied to the query. The LEADING hint is the easiest to use as it requires specifying just the start of the join.

Sometimes the CBO will not implement a join order even with a hint. This occurs when the requested join order is semantically impossible to satisfy the query.

L ***Risk Details***

Low risk; the hint will only affect the specific SQL statement.

Solution Implementation

See the reference documents below:

ORDERED hint

LEADING hint

Using Optimizer Hints

Why is my hint ignored?

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***

- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Use dynamic sampling to obtain accurate selectivity estimates

The purpose of dynamic sampling is to improve server performance by determining more accurate estimates for predicate selectivity and statistics for tables and indexes. The statistics for tables and indexes include table block counts, applicable index block counts, table cardinalities, and relevant join column statistics. These more accurate estimates allow the optimizer to produce better performing plans.

You can use dynamic sampling to:
- *Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation.*
- *Estimate statistics for tables and relevant indexes without statistics.*
- *Estimate statistics for tables and relevant indexes whose statistics are too out of date to trust.*

L *Effort Details*

Low effort; Dynamic sampling can be turned on at the instance, session, or query level.

M *Risk Details*

Medium risk; Depending on the level, dynamic sampling can consume system resources (I/O bandwidth, CPU) and increase query parse time. Its is best used as an intermediate step to find a better execution plan which can then be hinted or captured with an outline.

Solution Implementation

See the documents below:

When to Use Dynamic Sampling

How to Use Dynamic Sampling to Improve Performance

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. Nested Loop Joins

The causes and solutions to problems with the use of nested loop joins are listed here.

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Query has a USE_NL hint that is not appropriate

The query has a USE_NL hint that may have been improperly specified (specifies the wrong inner table) or is now obsolete.

Cause Justification
The query contains a USE_NL hint and performs better without the hint or with a USE_HASH or USE_MERGE hint.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has a USE_NL, FIRST_ROWS, or FIRST_ROWS_K hint that is favoring NL

- ***The query has a USE_NL hint that may have been improperly specified (specifies the wrong inner table) or is now obsolete.***
- ***The query has a FIRST_ROWS or FIRST_ROWS_K hint that is causing the CBO to favor index access and NL join types***

Remove the hints or avoid the use of the index by adding a NOINDEX() hint; NL joins will usually not be "cost competitive" when indexes are not available to the CBO.

Cause Justification
The query contains a USE_NL hint and performs better without the hint or with a USE_HASH or USE_MERGE hint.

Solution Identified: Remove hints that are influencing the choice of index

Remove the hint that is affecting the CBO's choice of an access path. Typically, these hints could be: INDEX_**, NO_INDEX, FULL, AND_EQUAL. By removing the hint, the CBO may choose a better plan (assuming statistics are fresh).

L ***Effort Details***

Low effort; simply remove the suspected hint, assuming you can modify the query.

L ***Risk Details***

Low; this change will only affect the query with the hint.

Solution Implementation

See related documents.

Hints for Access Paths

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

3. Merge Joins

The causes and solutions to problems with the use of merge joins are listed here.

4. ***Hash Joins***

The causes and solutions to problems with the use of hash joins are listed here.

Miscellaneous Causes and Solutions

1. ***Parsing***

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: Dynamic sampling is being used for the query and impacting the parse time

Dynamic sampling is performed by the CBO (naturally at parse time) when it is either requested via hint or parameter, or by default because statistics are missing. Depending on the level of the dynamic sampling, it may take some time to complete - this time is reflected in the parse time for the statement.

Cause Justification

- *The parse time is responsible for most of the query's overall elapsed time*
- *The execution plan output of SQLTXPLAIN, the UTLXPLS script, or a 10053 trace will show if dynamic sampling was used while optimizing the query.*

Solution Identified: Alternatives to Dynamic Sampling

If the parse time is high due to dynamic sampling, alternatives may be needed to obtain the desired plan without using dynamic sampling.

M *Effort Details*

Medium effort; some alternatives are easy to implement (add a hint), whereas others are more difficult (determine the hint required by comparing plans)

L *Risk Details*

Low risk; in general, the solution will affect only the query.

Solution Implementation

Some alternatives to dynamic sampling are:

1. *In 10g or higher, use the SQL Tuning Advisor (STA) to generate a profile for the query (in fact, its unlikely you'll even set dynamic sampling on a query that has been tuned by the STA)*
2. *Find the hints needed to implement the plan normally generated with dynamic sampling and modify the query with the hints*
3. *Use a stored outline to capture the plan generated with dynamic sampling*

For very volatile data (in which dynamic sampling was helping obtain a good plan), an approach can be used where an application will choose one of several hinted queries depending on the state of the data (i.e., if data recently deleted use query #1, else query #2).

Documents for hints:

Using Optimizer Hints

Forcing a Known Plan Using Hints

How to Specify an Index Hint

QREF: SQL Statement HINTS

Documents for stored outlines / plan stability:

Using Plan Stability

Stored Outline Quick Reference

How to Tune a Query that Cannot be Modified

How to Move Stored Outlines for One Application from One Database to Another

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Query has many IN LIST parameters / OR statements

The CBO may take a long time to cost a statement with dozens of IN LIST / OR clauses.

Cause Justification

- *The parse time is responsible for most of the query's overall elapsed time*
- *The query has a large set of IN LIST values or OR clauses.*

Solution Identified: Implement the NO_EXPAND hint to avoid transforming the query block

In versions 8.x and higher, this will avoid the transformation to separate query blocks with UNION ALL (and save parse time) while still allowing indexes to be used with the IN-LIST ITERATOR operation. By avoiding a large number of query blocks, the CBO will save time (and hence the parse time will be shorter) since it doesn't have to optimize each block.

L ***Effort Details***

Low effort; hint applied to a query.

L ***Risk Details***

Low risk; hint applied only to the query and will not affect other queries.

Solution Implementation

See the reference documents.

Optimization of large inlists/multiple OR`s

NO_EXPAND Hint

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Partitioned table with many partitions

The use of partitioned tables with many partitions (more than 1,000) may cause high parse CPU times while the CBO determines an execution plan.

Cause Justification

1. ***The parse time is responsible for most of the query's overall elapsed time***
2. ***Determine total number of partitions for all tables used in the query.***
3. ***If the number is over 1,000, this cause is likely***

Solution Identified: 9.2.0.x, 10.0.0: Bug 2785102 - Query involving many partitions (>1000) has high CPU/ memory use

A query involving a table with a large number of partitions takes a long time to parse, causes rowcache contention, and high CPU consumption. The case of this bug involved a table with greater than 10000 partitions and global statistics ere not gathered.

M ***Effort Details***

Medium effort; application of a patchset.

L ***Risk Details***

Low risk; patchsets generally are low risk because they have been regression tested.

Solution Implementation

Apply patchset 9.2.0.4

Workaround:
Set "_improved_row_length_enabled"=false

Additional bug information:
Bug 2785102

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Waits for large query texts to be sent from the client

A large query (containing lots of text) may take several round trips to be sent from the client to the server; each trip takes time (especially on slow networks).

Cause Justification

1. ***High parse wait times occur any time, not just during peak load or during certain times of the day***
2. ***Most other queries do not have high parse wait times at the same time as the query you are trying to tune***
3. ***TKProf shows "SQL*Net more data from client" wait events.***
4. ***Raw 10046 trace shows "SQL*Net more data from client" waits just before the PARSE call completes***
5. ***Slow network ping times due to high latency networks make these waits worse***

Solution Identified: Use PL/SQL REF CURSORs to avoid sending query text to the server across the network

The performance of parsing a large statement may be improved by encapsulating the SQL in a PL/SQL package and then obtaining a REF CURSOR to the resultset. This will avoid sending the SQL statement across the network and will only require sending bind values and the PL/SQL call.

M ***Effort Details***

Medium effort; a PL/SQL package will need to be created and the client code will need to be changed to call the PL/SQL and obtain a REF CURSOR.

L ***Risk Details***

Low risk; there are changes to the client code as well as the PL/SQL code in the database that must be tested thoroughly, but the changes are not widespread and won't affect other queries.

Solution Implementation

See the documents below.

How to use PL/SQL REF Cursors to Return Result Sets

Using Cursor Variables (REF CURSORs)

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

2. Parallel Execution (PX)

Note: This list shows some common causes and solutions but is not a complete list. If you do not find a possible cause or solution in this list, you can always open a service request with Oracle to investigate other possible causes.

Cause Identified: No parallel slaves available for the query

No parallel slaves were available so the query executed in serial mode.

Cause Justification
Event 10392, level 1 shows that the PX coordinator was enable to get enough slaves (at least 2).

Additional Information:
Why didn't my parallel query use the expected number of slaves?

Solution Identified: Additional CPUs are needed

Additional CPUs may be needed to allow enough sessions to use PX. If manual PX tuning is used, you will have to increase the value of PARALLEL_MAX_SERVERS after adding the CPUs.

M ***Effort Details***

Medium effort; adding CPUs may involve downtime depending on the high availability architecture employed.

L ***Risk Details***

Low risk; adding additional CPUs should only improve performance and scalability in this case.

Solution Implementation

Hardware addition, no details provided here.

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Cause Identified: Hints or configuration settings causing parallel plans

The CBO will attempt to use parallel operations if the following are set or used:

- ***Parallel hint: parallel(t1, 4)***
- ***ALTER SESSION FORCE PARALLEL***
- ***Setting a degree of parallel and/or the number of instances on a table or index in a query***

Cause Justification
Examine the 10053 trace and check the parallel degree for tables and presence of hints in the query. The presence of any of these is justification for this cause.

Additional Information:
Summary of Parallelization Rules

Solution Identified: Remove parallel hints

The statement is executing in parallel due to parallel hints. Removing these hints may allow the statement to run serially.

L ***Effort Details***

Low effort; simply remove the hint from the statement.

L ***Risk Details***

Low risk, only affects the statement.

Solution Implementation

Remove one or more hints of the type:

- ***PARALLEL***
- ***PARALLEL_INDEX***
- ***PQ_DISTRIBUTE***

If one of the tables has a degree greater than 1, the query may still run in parallel.

Hint information:
Hints for Parallel Execution

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- ***Review other possible reasons***
- ***Verify the data collection was done properly***
- ***Verify the problem statement***

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan

Solution Identified: Alter a table or index's degree of parallelism

A table or index in the query has its degree (of parallelism) set higher than 1. This may be one factor causing the query to execute in parallel. If the parallel plan is not performing well, a serial plan may be obtained by changing the degree.

L ***Effort Details***

Low effort; the object may be changed with an ALTER command.

M ***Risk Details***

Medium risk; other queries may be running in parallel due to the degree setting and will revert to a serial plan. An impact analysis should be performed to determine the effect of this change on other queries.

The ALTER command will invalidate cursors that depend on the table or index and may cause a spike in library cache contention - the change should be done during a period of low activity.

Solution Implementation

See the documents below.

Parallel clause for the CREATE and ALTER TABLE / INDEX statements

Implementation Verification

Re-run the query and determine if the performance improves. If performance does not improve, examine the following:

- *Review other possible reasons*
- *Verify the data collection was done properly*
- *Verify the problem statement*

If you would like to log a service request, a test case would be helpful at this stage, see the following document for instructions:

How to Submit a Testcase to Oracle Support for Reproducing an Execution Plan