Provided by Center of Expertise

Case Study: Resolving High CPU Usage on Oracle Servers

Author: Hector Pujol, Consulting Technical Advisor, Center of Expertise, Oracle USA

Skill Level Rating for this Case Study: Intermediate

About Oracle Case Studies

Oracle Case Studies are intended as learning tools and for sharing information or knowledge related to a complex event, process, procedure, or to a series of related events. Each case study is written based upon the experience that the writer/s encountered.

Each Case Study contains a skill level rating. The rating provides an indication of what skill level the reader should have as it relates to the information in the case study. Ratings are:

- Expert: significant experience with the subject matter
- Intermediate: some experience with the subject matter
- Beginner: little experience with the subject matter

Case Study Abstract

This case study demonstrates a top-down tuning methodology that was applied to a performance problem involving high CPU usage on a database server. The diagnostics that were collected and analyzed for this problem include: operating system (OS) data such as *vmstat* and *top*; statspack, and trace event 10046 with TKProf output. The conclusion from analyzing the collected data was that a specific query was consuming most of the CPU and needed to be tuned. We present various ways of tuning the query as well as verification that the recommended solution solved the problem.

Case History

A large university reported severe performance problems with their email application and other applications that relied upon an Oracle RDBMS server for authentication services. The Oracle database was used solely for user lookups, authentication, and other assorted LDAP queries. It was hosted on a 2-CPU Linux server that also hosted Oracle's Application Server (iAS) and related daemons.

The email application was installed approximately 6 months prior and ran without incident until some application security patches were installed. The day after the patches were installed, the performance problems started occurring.

The DBAs noticed that this authentication server was often seeing load averages over 20 processes (as shown in `uptime` or `top`). Normally, prior to the patch, the load averages were seldom above 1 or 2 processes. The performance seemed to be much worse when "mass mailing" occurred. These mass-mailing jobs sent out email to thousands of recipients and relied upon the authentication/lookup services provided by the database.

The main symptoms reported by users when there was a performance problem were that they were unable to log-in to their email accounts and they experienced timeouts in the application (visible as browser timeouts or client error messages, depending on the mail client used).

The database server version was 9.0.1.5; it had been upgraded from 9.0.1.4 during the same outage period when the application security patches were installed.

Analysis

Summary

Our strategy in diagnosing this problem was to take a "top-down" approach and follow the trail of highest CPU consumption down to the offending processes, and eventually the offending sessions and query. The top-down approach can be summarized as follows[1]:

1. Verify CPU Consumption on the server
   - Verify the observation of high CPU consumption on the affected server and collect *top* and *vmstat* data for a period of time (from 30 minutes up to a few hours)
   - Find which processes are using the most CPU
   - If the top CPU consumers are Oracle processes, then find out which database sessions correspond to these processes[2]

2. Collect Extended SQL Trace Data (Event 10046)
   - Gather SID, SERIAL#, and other information from V$SESSION about these sessions
   - Trace the sessions using an extended SQL trace via event 10046, level 12
   - Produce a TKProf report and look for the queries having the longest elapsed time

[1] The steps have been categorized according to the Oracle Diagnostic Method (ODM), see MetaLink Doc ID 312789.1
[2] If the processes were not Oracle RDBMS processes, then we would have to stop and find out what kind of processes they are and how to diagnose them.

3. Build a Test Case Script for the Problem Query
    - Extract the high-impact queries *with their bind values* and create a test script that will allow us to benchmark changes to the query and verify our progress in tuning it.
    - Run the test script on the production system (with the 10046 event set) to validate that it reflects what was seen in step two.

4. Obtain Statspack Snapshots
    - Collect statspack snapshots during the same time period as the OS were collected and look for correlations with the session-level data collected in step two. This could easily be done before steps 1 through 3 as well.

5. Tune the Query
    - Tune the query and verify performance improvements using the test script
    - Implement the changes in production and verify that the solution worked.
    - Verify the performance has improved and meets the customer's requirements

In steps 1 and 2 above, one can see that the overall effort is one that focuses attention down to a particular session, with the ultimate goal of collecting the extended SQL trace information. We are interested in the extended SQL trace information because it will report CPU times, *wait times*, elapsed times, and bind values[3]. By having the CPU, wait, and elapsed times, we can account for how the time was spent during each query's execution. The bind values allow us to put together a reproducible test case using actual values. The bind values will be extremely useful later when we tune the query.

[3] See Cary Milsap's book *Optimizing Oracle Performance* for an in-depth look at using event 10046.

Detailed Analysis

1. <u>Verify CPU Consumption on the server</u>

Before we can start looking at why this problem is occurring we must first understand the problem and verify what the customer is reporting (its symptoms). This can be done by collecting OS statistics during a recent spike in activity due to a mass mailing. The data was collected using Oracle Support's *OSWatcher* tool[4].

The *vmstat* output during this time looked like this:

```
# of processes in the run queue                                   CPU User mode
                                                                        CPU Sys mode
                                                                             CPU idle

zzz ***Thu Mar 9 18:30:00 PST 2006
  procs                         memory    swap          io     system        cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy  id
 3  0  0    920   8596 342908 4088164  0   0    10   514  501  1790  43   7  50
 5  0  0    920   8844 343024 4089564  0   0     8    79  319  1751  29   4  67
21  0  1    920   5332 343160 4093964  0   0    18   320  390  1690  97   3   0
22  0  0    920  22864 343304 4057500  0   0    11   149  370  2563  97   3   0
25  0  3    920   9728 343404 4059732  0   0    77   412  400  2412  95   5   0
23  0  0    920   8416 343688 4044128  0   0     2     2  254  1308  97   3   0
21  0  1    920   6760 343520 4045256  0   0    56  2428  704  2382  89  10   1
24  0  0    920  14856 343812 4045968  0   0    54    66  278  2785  95   5   0
21  0  2    920  17600 343956 4048296  0   0    17    70  249  1677  96   4   0
20  0  0    920   6028 344080 4050368  0   0     8    52  265  1461  94   6   0
22  0  1    920  27172 344184 4053608  0   0    25    49  263  2513  96   4   0
14  0  2    920  58264 344364 4057036  0   0    73   125  614  3821  79  20   1
 1  1  0    920  65756 344476 4070080  0   0   358  1964  410  2118  12   4  84
 0  0  0    920  48628 344580 4080864  0   0    17   106  229  1190   9   2  88
```

Each line represents a sample collected at 1-minute intervals.

Notice the how the run queues[5] rapidly increase from 3 to 25 as the CPU utilization increases from about 50 percent to 100 percent. This shows the importance of knowing the length of the run queue as well as the CPU utilization. When the CPU is pegged at 100% utilization, the severity of the CPU starvation won't be reflected in the percentage of CPU utilization (its pegged at 100%), but the run queue will clearly show the impact. Similarly, knowing only the run queue will not provide you with knowledge on the exact CPU usage, nor the spread of time usage across system and user modes.

CPU utilization is classified into three types: system (sy), user (us), and idle (id). *System* mode CPU utilization occurs whenever a process requires resources from the system; e.g.,

[4] OSWatcher may be downloaded from MetaLink by viewing Doc ID 301137.1
[5] The run queue is a queue of processes that are ready to run but must wait for their turn on a CPU; a run queue of 20 means that 20 processes are currently waiting to execute.

I/O or memory allocation calls. In typical OLTP systems, the percentage of system mode CPU utilization is often less than 10 percent; in data warehouses more I/O calls are performed and a higher percentage of system mode CPU utilization is common. *User* mode CPU utilization accounts for the rest of the time the CPU is busy and <u>not</u> running in system mode. Idle CPU is essentially the time that the CPU is not busy and waiting for work to do.

In this case, we can quickly see that the percentage of CPU utilization in system mode is usually less than 10 percent. This indicates that we are probably not suffering from memory shortages or doing excessive I/Os[6]. We can further dismiss memory shortages by looking at the paging and swapping statistics in *vmstat* (only swap-ins (si) and swap-outs (so) are shown here). If no pages are being swapped out during peak time, then there can't possibly be a memory shortage.

If we were seeing system-mode CPU utilization higher than 15 percent and we weren't seeing any memory problems, we might begin to suspect excessive I/O calls, very frequent database connections/disconnections, or some anomaly (we would keep this in mind as we collect more data for individual processes; e.g., using the *truss* command or while looking at RDBMS metrics like *logons cumulative* per second).

The fact that most CPU utilization is occurring in *user* mode tells us that whatever is occurring, it's happening within the application that the process is running, not as part of a system call or action. The question is now, which processes are using the CPU?

To answer this question, we can look at the output of the *top* command. This command is useful because it shows at a glance the overall CPU statistics and the processes consuming the most CPU (ranked in descending order of CPU usage). The following shows a typical output of the *top* command during the performance problem:

[6] This is just a casual observation – we would need to look at the process in detail (at least at the statspack level for the database) to conclusively decide that excessive I/O is not being performed

```
6:37pm  up 4 days, 10:09, 11 users,  load average: 22.31, 16.26, 8.79
495 processes: 472 sleeping, 23 running, 0 zombie, 0 stopped
CPU0 states: 79.0% user, 20.0% system,  0.0% nice,  0.0% idle
CPU1 states: 96.0% user,  3.0% system,  0.0% nice,  0.0% idle
CPU2 states: 94.0% user,  5.0% system,  5.0% nice,  0.0% idle
CPU3 states: 94.0% user,  5.0% system,  0.0% nice,  0.0% idle
Mem:  5921032K av, 5904944K used,   16088K free,  500700K shrd,  343812K buff
Swap: 4192912K av,     920K used, 4191992K free                 4045896K cached
```

Process owner

Partial process name; more detail is needed to identify which kind of process (database, iAS, ?)

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
24067 oracle    25   0  409M 378M  377M R    28.0  6.5  37:34 oracle
24082 oracle    25   0  409M 378M  377M R    24.2  6.5  61:19 oracle
24044 oracle    25   0  411M 379M  378M R    23.3  6.5  61:59 oracle
24086 oracle    25   0  411M 380M  379M R    23.3  6.5  60:50 oracle
24091 oracle    25   0  411M 380M  378M R    23.3  6.5  38:20 oracle
24085 oracle    25   0  408M 377M  375M R    21.4  6.5  38:24 oracle
24080 oracle    25   0  410M 379M  377M R    20.5  6.5  61:12 oracle
24081 oracle    25   0  409M 378M  377M R    19.6  6.5  37:50 oracle
24079 oracle    25   0  409M 378M  376M R    17.7  6.5  38:16 oracle
24084 oracle    25   0  411M 380M  378M R    17.7  6.5  61:20 oracle
24089 oracle    25   0  410M 379M  378M R    17.7  6.5  37:44 oracle
24077 oracle    25   0  410M 379M  377M R    16.8  6.5  38:20 oracle
24072 oracle    25   0  409M 377M  376M R    14.9  6.5  62:17 oracle
24087 oracle    25   0  409M 378M  376M R    14.9  6.5  38:14 oracle
24066 oracle    25   0  411M 380M  378M R    13.0  6.5  61:49 oracle
24078 oracle    25   0  410M 379M  377M R    10.2  6.5  62:37 oracle
24090 oracle    25   0  410M 378M  377M R    10.2  6.5  62:02 oracle
24088 oracle    25   0  409M 378M  377M R     8.4  6.5  62:13 oracle
14657 oracle    15   0 96728  63M 62908 S     5.6  1.0   0:05 oracle
```

Top CPU consumers; No process name visible here

Notice the load average for the last one-minute has been around 22 (load averages are for the last 1, 5, and 15 minutes). The load averages are an average[7] of the run queue and basically means that around 22 processes were ready to run; however, the machine has only four virtual CPUs (two real Intel Xeon Hyperthreaded CPUs). The lines beginning with "CPUx states" show that Linux believes there are 4 CPUs. This means that on average, each CPU has one process running and over 5 waiting to run (22 / 4).

It's clear from the output of the *top* command that Oracle database processes are using nearly all of the CPU. We can further clarify the names of these processes by looking at the output of the *ps* command (or use *top*'s "c" command in *top*'s interactive mode):

Accumulated process CPU time

Full process name; these are Oracle database processes

```
  F S UID         PID  PPID    TIME          CMD
000 S oracle    24067 23955    00:35:42 oracleiasdb DESCRIPTION=(LOCAL=YES)…
000 S oracle    24072 23957    00:59:31 oracleiasdb (DESCRIPTION=(LOCAL=YES)…
000 S oracle    24077 23955    00:36:33 oracleiasdb (DESCRIPTION=(LOCAL=YES)…
```

[7] Actually, an exponentially damped moving average; see Gunther under *Additional Resources*

The *ps* command[8] shows the accumulated CPU time for each process as well as the full name for the process associated with the process ID (PID). The process IDs are important for moving along to the next stage of the diagnostic process where we examine and trace the database sessions that are associated with these PIDs. The full name tells us what kind of process it is. In this case they were all Oracle database processes.

At this point we have verified the following:

- The system is under extreme CPU starvation and very high run queues have formed
- Oracle database processes are responsible for the high CPU consumption

If we had found non-Oracle processes using most of the CPU, then we would need to drill-down into those processes and investigate the reason for such high CPU consumption. Since Oracle processes were clearly the culprits here, we drilled down into what Oracle was doing.

Database Drilldown

Once we've verified the problem exists and involves Oracle processes, it was time to determine the cause of the CPU consumption. From the *top* and *ps* output, we had a list of PIDs that were responsible for most of the CPU consumption. The next step was to find out more about the sessions in the database that were associated with these processes. We wanted to find out the following:

- Whether any of these sessions were "runaways" (i.e., stuck in an infinite loop and continuously using CPU)
- Which SQL hash values were most commonly seen when the session was active (same SQL statement)
- SID and serial# for tracing purposes

Here is the query that was used to find out more about these sessions:

```
SELECT /*+ ordered */ p.spid, s.sid, s.serial#, s.username,
TO_CHAR(s.logon_time, 'mm-dd-yyyy hh24:mi') logon_time, s.last_call_et, st.value,
s.sql_hash_value, s.sql_address, sq.sql_text
FROM v$statname sn, v$sesstat st, v$process p, v$session s,  v$sql sq
WHERE s.paddr=p.addr
AND s.sql_hash_value = sq.hash_value and s.sql_Address = sq.address
AND s.sid = st.sid
AND st.STATISTIC# = sn.statistic#
AND sn.NAME = 'CPU used by this session'
AND p.spid = &osPID -- parameter to restrict for a specific PID
AND s.status = 'ACTIVE'
ORDER BY st.value desc
```

The OS PID that was observed from *top* was entered in for the parameter "osPID" (we can also remove this predicate and see the top sessions in the database ordered by highest CPU usage). We executed this query every 1 or 2 seconds and observed the value of *last_call_et*

[8] `ps aux` produces useful output on most systems.

to see if it kept growing or was reset to 0 or some small value. This value will increment every 3 seconds while a session executes a call; the longer the call has been running, the larger the value. If the call finished and an new one started, the value of *last_call_et* would be reset. We care about this because if we had a runaway session, then the *last_call_et* would keep growing as the session was processing the same call for a long time (while consuming CPU).

We noticed the following when we ran this query:

- None of the top CPU consuming sessions had *last_call_et* greater than 3 seconds any time we ran the query. No runaways…whatever is going on is completing within 3 seconds.
- The same 10 or 12 sessions were at the top accruing CPU time.
- We very frequently saw the same *sql_hash_value* and SQL statement while the sessions were active
- The statistic *CPU used by this session* statistic was accruing time steadily

After running this query, we had a list of 12 candidate sessions that we could trace using the extended SQL trace event. We chose the session with the highest accumulated CPU time (*CPU used by this session*), any one of them would have been a good candidate since their total CPU consumption differed only by a few percentage points.

2. Collect Extended SQL Trace Data (Event 10046)

At this point in the process, we were ready to gather the extended SQL traces and find out exactly what the sessions were doing to use all of this CPU.

The extended SQL trace will help us account for the time spent executing SQL for a particular session. We will be able to account for CPU and wait time as well as see the sequence in which SQL statements are emitted by the application and which of them is taking the most CPU and/or elapsed time. In addition, we'll obtain the bind values associated with a query so that we can use them to create our test script. We have to ensure the statement uses binds because the CBO might choose different execution plans if we use literals instead.

With the session ID and serial number of the session obtained from the V$SESSION queries we ran earlier, we started the 10046 trace as follows:

```
SQL> SELECT p.spid, s.sid, s.serial#
FROM v$session s, v$process p
WHERE s.paddr = p.addr AND p.spid = 24078


SPID                SID    SERIAL#
------------ ---------- ----------
24078                18          5

SQL> begin dbms_system.set_ev(18,5, 10046,12,''); end; -- trace on

-- collect trace information for approximately 15 minutes during the problem

SQL> begin dbms_system.set_ev(18, 5, 10046,0,''); end; -- trace off
```

This produced a trace file in the `user_dump_dest` directory. We used the *tkprof* [9]utility to format the trace file and sort the SQL statements in descending order of highest fetch elapsed time as follows:

[9] We used *tkprof* from 9.2 to process this 9.0.1.5 trace file because the 9.0.1.5 *tkprof* does not account for wait times properly.

```
$ tkprof ora_24078.trc 24078.prf sort=fchela

$ cat 24078.prf
                         <some lines removed for brevity>

SELECT /*+ USE_NL(store) USE_NL(dn) INDEX(store EI_ATTRSTORE) INDEX(dn EP_DN)
  ORDERED */ dn.entryid,store.attrname, NVL(store.attrval,' '),
  NVL(store.attrstype, ' ')
FROM
 ct_dn dn, ds_attrstore store WHERE dn.entryid IN (  (SELECT /*+ INDEX( at1
  VA_objectclass ) */ at1.entryid FROM CT_objectclass at1 WHERE at1.attrvalue
   = :0) UNION ((  (SELECT /*+ INDEX( at1 VA_mail ) INDEX( at2 VA_objectclass
  ) */ at1.entryid FROM CT_mail at1, CT_objectclass at2  WHERE at1.attrvalue
  = :1 AND at2.attrvalue  = :2 AND at2.entryid = at1.entryid) UNION (SELECT
  /*+ INDEX( at1 VA_mail ) INDEX( at2 VA_objectclass ) */ at1.entryid FROM
  CT_mail at1, CT_objectclass at2  WHERE at1.attrvalue  = :3 AND
  at2.attrvalue  = :4 AND at2.entryid = at1.entryid) )) ) AND ( (dn.parentdn
  like :bdn  ESCAPE '\' OR (dn.rdn = :rdn AND dn.parentdn = :pdn)) ) AND
  dn.entryid = store.entryid AND dn.entryid >= :entryThreshold AND
  store.attrkind = 'u'  AND store.attrname NOT IN ('member', 'uniquemember')


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse      555      0.09       0.83          0          0          0           0
Execute    555      0.42       0.78          0          0          0           0
Fetch      555    114.04     385.03        513    1448514          0       11724
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1665    114.55     386.65        513    1448514          0       11724


Elapsed times include waiting on following events:

  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message from client                   556        1.94        134.68
  SQL*Net message to client                     555        0.00          0.00
  db file sequential read                       513        0.65          6.68
  SQL*Net more data to client                   169        0.00          0.02
  latch free                                    323        1.03         59.85
```

The top SQL statement found in the TKProf report (sorted by fetch elapsed time) was the same one we saw frequently while querying V$SESSION (hash value of 1656209319).

Let’s analyze the TKProf to see how the time was spent for this SQL statement:

1.) An average of 11,724 rows / 555 fetch calls = 21 rows/call is returned
2.) An average of 1448514 buffer gets /555 executions = 2,609 buffer gets/execution
3.) The elapsed time for this statement was 386.65 seconds
4.) The CPU time for this statement was 114.55 seconds
5.) The wait time should be:
    wait time = elapsed time – CPU time
    wait time = 386.65 – 114.55 = 272.10 seconds

6.) The sum of the wait times shown in the area under “Elapsed times include waiting on the following events” is 66.55 seconds. We exclude the time listed as “SQL*Net message from client” because this wait time is not part of the elapsed time for the query. When we examined the trace file, we saw that this was the database “idle” time in between the completion of this query while waiting for the next query to be processed. In effect, this is the client and network time (a.k.a. “think time” from the database’s point of view)[10].

In this case we ignore the time in between database calls (“SQL*Net message from client”, 134.68 seconds) because we know the system is starving for CPU. If we hadn’t seen any bottlenecks at the OS or database stacks, we would pay attention to this inter-call time because it could strongly indicate a bottleneck in the middle tier (not a database performance problem). The CPU saturation is the main bottleneck – clearly, the machine is not anywhere near idle that we need to look elsewhere for a performance problem!

It’s interesting that the wait time didn’t add up to the 272.10 seconds we anticipated…where did the “missing” time go? Because this system was undergoing severe CPU starvation and the run queues were very high (over 20), processes were spending a considerable amount of time waiting in the run queue. While they were in the run queue, Oracle didn’t accrue any CPU time, nor was it accruing any wait time from the database point of view (i.e., the session wasn’t waiting for “db file sequential read” or anything else because it wasn’t *sleeping*). It’s as if time stood still for the session while it was on the run queue, but, in fact time was passing (“elapsed time” or “wall clock” time) – it just wasn’t *measurable* by Oracle until the call ended[11].

What about the “latch free” wait event – was that important? Normally, one would say ‘yes’ and would focus on why we’re seeing latching problems. In cases where the system is facing CPU starvation, apparent latch contention can flare up as a byproduct of the CPU starvation[12]. It’s also possible that we have real latch waits on the *cache buffers chains* latch due to block contention (“hot blocks”). Considering that this query executed 555 times in 17 minutes by a dozen or so concurrent sessions, it’s quite possible that they were contending for the same index root and branch blocks (and hence the *cache buffers chains* latch that protects the buffer headers for those blocks). However, this points to the same root cause as the high CPU: the query needs to be tuned.

[10] For queries that require multiple fetches, you will see SQL*Net Message from Client waits *between* fetch calls. If a small arraysize is used for fetches, then the query will require multiple fetch calls, each one wasting time waiting for the client/network to issue the next fetch call and causing additional work in the database to reread blocks.
[11] Just like the story of *Rip van Winkle* (http://en.wikipedia.org/wiki/Rip_Van_Winkle)
[12] To add insult to injury, latch contention induces additional CPU usage while a session “spins” on the CPU hoping for the latch to become available. In fact, in this case we found it was actually 85% higher during saturation by comparing the query’s average CPU usage per execution when the system was saturated versus when it wasn’t.

Examining the Query’s Execution Plan

The runtime execution plan is shown just below the statement statistics in TKProf and looks like this:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
     19  TABLE ACCESS BY INDEX ROWID DS_ATTRSTORE
     26   NESTED LOOPS
      1    NESTED LOOPS
      1     VIEW VW_NSO_1
      1      SORT UNIQUE
      1       UNION-ALL
      0        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
      0         INDEX RANGE SCAN (object id 55349)
      1        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
  64439         NESTED LOOPS
      2          TABLE ACCESS BY INDEX ROWID CT_MAIL
      2           INDEX RANGE SCAN (object id 55355)
  64436          INDEX RANGE SCAN (object id 55349)
      0        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
     27         NESTED LOOPS
      2          TABLE ACCESS BY INDEX ROWID CT_MAIL
      2           INDEX RANGE SCAN (object id 55355)
     24          INDEX RANGE SCAN (object id 55349)
      1     TABLE ACCESS BY INDEX ROWID CT_DN
      1      INDEX RANGE SCAN (object id 55345)
     24    INDEX RANGE SCAN (object id 55340)
```

CBO used VA_OBJECTCLASS index due to INDEX hint

The areas of interest in this plan are:

- the top line that shows 19 rows were returned
- the line in the middle: “`64436          INDEX RANGE SCAN (object id 55349)`”

This means that an index range scan occurred and this scan visited 64,436 rows but ultimately only 19 rows were returned by the query. A lot of work was done for very few rows. Typically, OLTP systems use indexes to minimize the amount of I/Os required to fetch the necessary rows. We can be sure that the average of 2,600 or so buffer gets per execution occurred due to this index. *Buffer Gets* is a measure of how much work the database is performing for each SQL statement and could also be referred to as *Logical I/Os* (LIOs). This is a strong indication that we should attempt to tune this query and reduce the number LIOs and thereby reduce the CPU consumption (and, as a bonus relieve the latch contention).

Our next step will be to put together a simple test script so we can begin tuning this query.

3. <u>Build a Test Case Script for the Problem Query</u>

In order to build an accurate test case script for this query, we will need to set the bind values to some typical values that were captured in the extended SQL trace. The bind values are found in the trace file as such:

"dty" means "data type"; 1 = VARCHAR2 (see the Oracle SQL Reference, Datatypes)

```
BINDS #14:
 bind 0: dty=1 mxl=2000(499) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=40683fe4 bln=2000 avl=08 flg=05
   value="referral"
 bind 1: dty=1 mxl=2000(499) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=40683808 bln=2000 avl=20 flg=05
   value="gwiz@u.edu"
 bind 2: dty=1 mxl=2000(499) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=4068302c bln=2000 avl=12 flg=05
   value="orclmailuser"
 bind 3: dty=1 mxl=2000(499) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=40682850 bln=2000 avl=20 flg=05
   value="gwiz@u.edu"
 bind 4: dty=1 mxl=2000(499) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=40682074 bln=2000 avl=13 flg=05
   value="orclmailgroup"
 bind 5: dty=1 mxl=2000(1023) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=40681898 bln=2000 avl=54 flg=05
   value="cn=oraclecontext,cn=products,cn=emailservercontainer,%"
 bind 6: dty=1 mxl=2000(1023) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=406810bc bln=2000 avl=23 flg=05
   value="cn=emailservercontainer"
 bind 7: dty=1 mxl=2000(1023) mal=00 scl=00 pre=00 oacflg=00 oacfl2=10 size=2000
offset=0
   bfp=406808e0 bln=2000 avl=29 flg=05
   value="cn=oraclecontext,cn=products,"
 bind 8: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=00 oacfl2=0 size=24 offset=0
   bfp=4065a708 bln=22 avl=02 flg=05
   value=1000
```

When we extract the bind values and set up the variables, we obtain the following script that can be run in SQL*Plus:

```
var val0 char(8)
var val1 char(19)
var val2 char(12)
var val3 char(19)
var val4 char(13)
var bdn char(54)
var rdn char(23)
var pdn char(29)

exec :val0 := 'referral';
exec :val1 := 'gwiz@u.edu';
exec :val2 := 'orclmailuser';
exec :val3 := 'gwiz@u.edu';
exec :val4 := 'orclmailgroup';
exec :bdn :=  'cn=oraclecontext,cn=products,cn=emailservercontainer,%';
exec :rdn :=  'cn=emailservercontainer';
exec :pdn :=  'cn=oraclecontext,cn=products,';

SELECT   /*+ USE_NL(store) USE_NL(dn) INDEX(store EI_ATTRSTORE) INDEX(dn EP_DN)
  ORDERED */ dn.entryid,store.attrname, NVL(store.attrval,' '),
  NVL(store.attrstype, ' ')
FROM
 ct_dn dn, ds_attrstore store WHERE dn.entryid IN (  (SELECT /*+ INDEX( at1
  VA_objectclass ) */ at1.entryid FROM CT_objectclass at1 WHERE at1.attrvalue
   = :val0) UNION ((  (SELECT /*+ INDEX( at1 VA_mail ) INDEX( at2 VA_objectclass
  ) */ at1.entryid FROM CT_mail at1, CT_objectclass at2  WHERE at1.attrvalue
  = :val1 AND at2.attrvalue  = :val2 AND at2.entryid = at1.entryid) UNION (SELECT
  /*+ INDEX( at1 VA_mail ) INDEX( at2 VA_objectclass ) */ at1.entryid FROM
  CT_mail at1, CT_objectclass at2  WHERE at1.attrvalue  = :val3 AND
  at2.attrvalue  = :val4 AND at2.entryid = at1.entryid) )) ) AND ( (dn.parentdn
  like :bdn  ESCAPE '\' OR (dn.rdn = :rdn AND dn.parentdn = :pdn)) ) AND
  dn.entryid = store.entryid AND dn.entryid >= 1000 AND
  store.attrname NOT IN ('member', 'uniquemember');
```

Bind values set to values found in the trace file

Our test case script produced the following trace:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1    300.00     506.02         10         37          0           0
Execute      1      0.00      14.07          0          0          0           0
Fetch        4   1700.00    1900.18         15       2652          0          42
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        6   2000.00    2420.27         25       2689          0          42

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 141

Rows     Row Source Operation
-------  ---------------------------------------------------
     42  TABLE ACCESS BY INDEX ROWID DS_ATTRSTORE
     44   NESTED LOOPS
      1    NESTED LOOPS
      1     VIEW VW_NSO_1
      1      SORT UNIQUE
      1       UNION-ALL
      0        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
      0         INDEX RANGE SCAN (object id 55349)
      1        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
  64433         NESTED LOOPS
      2          TABLE ACCESS BY INDEX ROWID CT_MAIL
      2           INDEX RANGE SCAN (object id 55355)
  64430          INDEX RANGE SCAN (object id 55349)
      0        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
     27         NESTED LOOPS
      2          TABLE ACCESS BY INDEX ROWID CT_MAIL
      2           INDEX RANGE SCAN (object id 55355)
     24          INDEX RANGE SCAN (object id 55349)
      1     TABLE ACCESS BY INDEX ROWID CT_DN
      1      INDEX RANGE SCAN (object id 55345)
     42    INDEX RANGE SCAN (object id 55340)
```

Index forced CBO to chose index (VA_OBJECTCLASS)

We can now compare the results of the trace generated while running our test case script to the original trace we obtained of an actual running process. We find that they have:

- The exact execution plan
- The VA_OBJECTCLASS index is traversing approximately the same number of rows (64,000 rows)
- The buffer gets per execution is approximately the same at 2,600

This tells us the test script is representative of the actual query and we can confidently use it for tuning purposes[13].

[13] Sometimes we need to be careful with session level parameters that are set by the application and be sure to set them in our test as well.

4. Obtain Statspack Snapshots

Before we begin tuning the query using our test script, it will be useful to briefly consider overall database performance statistics collected using Statspack.

Statspack snapshots were taken of the database to get an overall picture of database performance at time of the problem (while collecting OS statistics and 10046 tracing). Ideally, we would compare the statspack reports when performance was good to the ones during the performance problem. Doing this kind of comparison is extremely valuable because we would be able to see a change in the CPU, wait event profile, and SQL statement performance and then pursue the reasons for the specific changes. Unfortunately, in this case statspack wasn't installed until *after* the performance problem was encountered, so we have no performance data from before the patch was applied.

It could be argued that this step was rather unnecessary since we already knew that CPU was being used by a number of sessions and the next step would have been to simply trace those sessions and see what they were doing. While this is a valid point, obtaining statspack snapshots is easy and will give us some overall context as well as provide a good sanity check that what we see for the overall database is present in an individual session's SQL trace file. Capturing this data also allows us to quantify the effect of this single statement's resource usage against that of the entire instance.

Accounting for Database Time

During the performance problem we obtained the following excerpts from the statspack report's header section:

```
STATSPACK report for

DB Name         DB Id    Instance     Inst Num Release     Cluster Host
------------ ----------- ------------ -------- ----------- ------- ------------
IASDB          197409164 iasdb               1 9.0.1.5.0   NO      prod.u.edu


            Snap Id     Snap Time      Sessions Curs/Sess Comment
            ------- ------------------ -------- --------- -------------------
Begin Snap:     854 09-Mar-06 18:21:08      202      11.4 Mass Mail
  End Snap:     856 09-Mar-06 18:47:25      194      11.6 End Mass Mail
   Elapsed:                26.28 (mins)
```

Focusing our attention on the "Top 5 Wait Events", we see:

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                                 Wait     % Total
Event                                               Waits     Time (s)   Wt Time
-------------------------------------------- ------------ ----------- -------
latch free                                         10,397       1,616   80.99
db file sequential read                            76,548         258   12.94
enqueue                                                27          66    3.33
db file scattered read                             72,761          18     .89
log file sync                                       2,845          14     .72
```

It's clear that the top wait event is related to latch acquisition. This may tempt some tuners to try and resolve a latching problem and find out which latch is the main point of contention. However, wait time is really just part of the total time that must be accounted for in the database. The total *active*[14] time spent in the database (a.k.a *DB Time* in later versions) by foreground sessions[15] is the following:

Total DB FG Time = Total DB FG CPU time + Total DB FG Wait Time

Unfortunately, since this is a 9.0.1.x database, Statspack did not incorporate CPU time into the "Top 5" list. To obtain the CPU for *foreground* Oracle sessions, you will need to look in the "Instance Activity Stats" section of the statspack report for the value of *CPU used by this session*[16] as shown below.

```
Instance Activity Stats for DB: IASDB  Instance: iasdb  Snaps: 742 -752

Statistic                                 Total     per Second    per Trans
--------------------------------- ------------------ -------------- ------------
CPU used by this session                331,934          210.5         85.0
```

The value of *CPU used by this session* is shown in centiseconds and must be converted to seconds. When we do this and combine it with the wait events we get a total database time of 5,291 seconds and this breakdown:

Event	Waits	Time (s)	% Total DB Time
CPU	N/A	3,319	62.72
latch free	10,397	1,616	30.54
db file sequential read	76,548	258	4.87
enqueue	27	66	1.25
db file scattered read	72,761	18	0.34
log file sync	2,845	14	0.27

[14] *Active* as opposed to *idle*
[15] *Foreground session* refers to Oracle sessions that are NOT doing *background* work such as LGWR or SMON. Usually, we only care about foreground session time because users connect to FG sessions and it's where they experience the performance of the database.
[16] Even though the name of the statistic says "…by this session" in V$SYSSTAT (and statspack) it is the accumulated value of CPU used by *all* foreground sessions who have completed their calls between the beginning and ending snapshots.

Now, this clearly reflects the fact that the database is spending most of its time on the CPU (or waiting for CPU). Some questions we might have at this point are:

1.) What is using this CPU?

> The first place to look is the section "SQL ordered by Gets". Looking here we see the statement we identified using the 10046 trace accounts for 90% of the CPU usage on this system in this snapshot period. This confirms this is the statement to tune (more detail on this below).

2.) Why did we have 5,291 seconds of database time when the elapsed time for this report was just 1,576.8 seconds (26.28 min)?

> The total DB time depends on the number of sessions waiting in the database (which is only limited by how many sessions could possibly connect to the database) and the number of CPUs available on the machine. As an extreme example, one could have 1,000 sessions all waiting for an enqueue for an entire hour and the total database time would be:
>
> 1,000 sessions X 3,600 seconds of wait /session = 3,600,000 seconds of DB time
>
> Or, one could have no sessions waiting but four CPUs and four sessions busy using the CPUs for one entire hour:
>
> 4 sessions X 3600 sec of CPU / session = 14,400 seconds of DB time.
>
> In this report we can see that there were approximately 200 sessions[17] during the report period (see the report header section above) and a total DB time possible of:
>
> 200 sessions X 1,576.8 seconds / session = 315,360 seconds of DB possible.
>
> The actual DB time is much lower because the majority of sessions were idle (there were hundreds of idle sessions and just about a dozen busy ones).

3.) How does the 3,319 seconds of CPU time compare to the total amount of CPU time that was available?

> To answer this one has to work out what the total possible CPU time was for the duration of the statspack report and then compare that to the total amount of CPU utilized. For this system we were investigating, there were two hyperthreaded CPUs which gave us a total of 4 CPUs for the statspack report duration of 26.28 minutes, for a total *possible* amount of CPU of:

[17] This is a rough estimate – we might have had fewer or greater numbers of sessions logged in between the snapshots.

4 X 1576.8 seconds = 6307.2 seconds of CPU available

The total CPU utilization by the database was then: 3,319 / 6307.2 = 53%.

Although this is a large value, it appears to contradict our view from *top* and *vmstat* that the database was using close to 100% of CPU. Unfortunately, we didn't find the reason for this discrepancy but it may be related to the CPUs being "Hyperthreaded" rather than actual independent CPUs. We believe the database's CPU metrics may have somehow been underestimated by a factor of two.

4.) Why are latch waits so prominent?

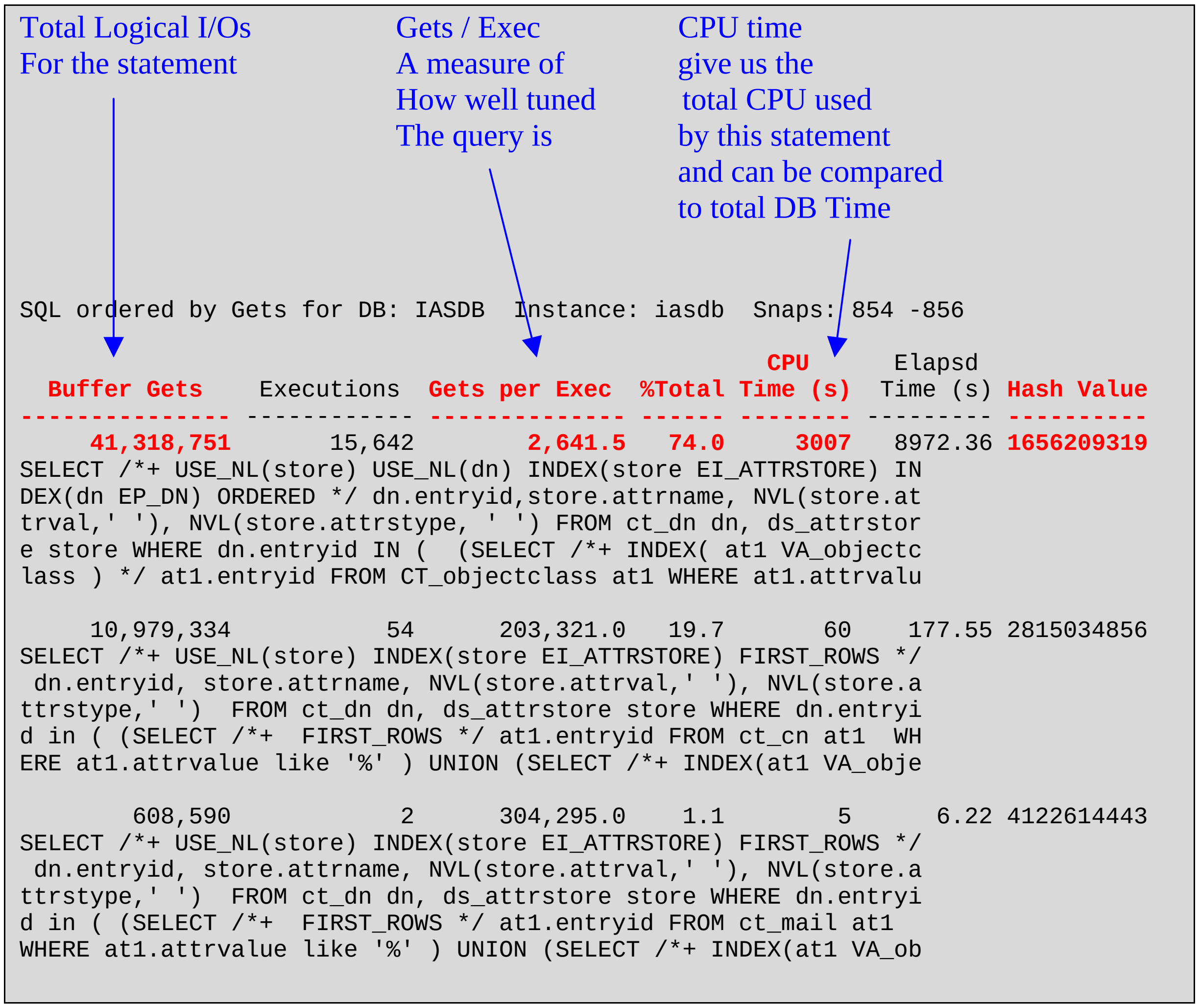
When CPU is scarce, it is quite likely that a session will be holding a resource such as a latch or enqueue when it is taken off the CPU and put on the run queue. This means that another session that needs that resource will have to wait for it much longer than it otherwise would. In fact, it has to wait until the holder is back on the run queue long enough to finish what it was doing and release the latch or enqueue. This point highlights the importance of understanding the overall performance picture from an OS standpoint before jumping to conclusions in statspack or a 10046 trace. When CPU starvation is involved, one should usually avoid putting too much credit in the wait events and instead seek to understand what is consuming the CPU[18]. Of course, as was discussed before, hot blocks due to the large number of concurrent buffer gets from the top query could cause the latch contention, but at this point they shouldn't be the main focus of our effort.

Finding the Top SQL statements in Statspack

Although we have already identified a particular SQL statement as a candidate for tuning, we will briefly look at the top SQL statements reported by Statspack to see if they agree with what we saw for an individual session. We'll be sure that we're on the right track if the top statement in the trace agrees with what we see as the top statement for the overall instance in Statspack

We'll look at the statspack report's section called "SQL ordered by Gets" to get an idea of the most expensive SQL statements from a CPU point of view. *Buffer Gets* is a measure of how much work the database is performing for each SQL statement and could also be referred to as *Logical I/Os* (LIOs). Each buffer get may or may not result in a physical read from disk, but will definitely incur CPU time as well as possibly experiencing latch or buffer pin waits (a.k.a. *buffer busy waits*). Here is an excerpt from the Statspack report:

[18] CPU starvation can lead to other effects since Oracle backgrounds processes like LGWR or DBWR can't get enough time on the CPU. In the case of LGWR one might see high waits for *log file sync* or in the case of DBWR free *buffer waits,* depending on the workload. Trying to "fix" these wait events will lead nowhere since they're a side effect of the actual problem.



Total Logical I/Os
For the statement

Gets / Exec
A measure of
How well tuned
The query is

CPU time
give us the
total CPU used
by this statement
and can be compared
to total DB Time

```
SQL ordered by Gets for DB: IASDB  Instance: iasdb  Snaps: 854 -856

                                                           CPU      Elapsd
  Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- --------- ----------
     41,318,751       15,642        2,641.5   74.0     3007   8972.36 1656209319
SELECT /*+ USE_NL(store) USE_NL(dn) INDEX(store EI_ATTRSTORE) IN
DEX(dn EP_DN) ORDERED */ dn.entryid,store.attrname, NVL(store.at
trval,' '), NVL(store.attrstype, ' ') FROM ct_dn dn, ds_attrstor
e store WHERE dn.entryid IN (  (SELECT /*+ INDEX( at1 VA_objectc
lass ) */ at1.entryid FROM CT_objectclass at1 WHERE at1.attrvalu

     10,979,334           54      203,321.0   19.7       60    177.55 2815034856
SELECT /*+ USE_NL(store) INDEX(store EI_ATTRSTORE) FIRST_ROWS */
 dn.entryid, store.attrname, NVL(store.attrval,' '), NVL(store.a
ttrstype,' ')  FROM ct_dn dn, ds_attrstore store WHERE dn.entryi
d in ( (SELECT /*+  FIRST_ROWS */ at1.entryid FROM ct_cn at1  WH
ERE at1.attrvalue like '%' ) UNION (SELECT /*+ INDEX(at1 VA_obje

        608,590            2      304,295.0    1.1        5      6.22 4122614443
SELECT /*+ USE_NL(store) INDEX(store EI_ATTRSTORE) FIRST_ROWS */
 dn.entryid, store.attrname, NVL(store.attrval,' '), NVL(store.a
ttrstype,' ')  FROM ct_dn dn, ds_attrstore store WHERE dn.entryi
d in ( (SELECT /*+  FIRST_ROWS */ at1.entryid FROM ct_mail at1
WHERE at1.attrvalue like '%' ) UNION (SELECT /*+ INDEX(at1 VA_ob
```

Some key things we notice here:

1.) The statement with hash value 1656209319 was responsible for 74% of the total buffer gets
2.) This statement was responsible for 3,007 / 3,319 = 90% of the CPU consumption in the database.
3.) Each execution used: 3,007 cpu sec/15,642 exec = 192 msec CPU/execution
4.) The statement was doing approximately 2,642 buffer gets for each execution.
5.) The second and third statements were not even close to the first one in terms of total CPU time consumed.
6.) The statement's total elapsed time was 8,972.36. Huh? How could we wait longer than the DB Time of 5,291 seconds? The reason is that while a session is executing this statement, it could be put back and forth on the run queue as the OS schedules other processes to run. Each time the process goes on the run queue, the elapsed time continues accruing, but not CPU time. Oracle has no idea it was on the run queue, but eventually when the call finishes, the elapsed time reflects the total execution time including the time on the run queue.

Unfortunately, the statspack report truncates the SQL text and there were other statements that looked very similar to this. So, we had to verify that this is really the same statement that we saw earlier in the TKProf output[19]. We did this by looking in the raw 10046 trace file we collected and ensured that the hash value in the statspack report was the same one we saw in the 10046 trace:

```
PARSING IN CURSOR #14 len=970 dep=0 uid=140 oct=3 lid=140 tim=1141957838281460
hv=1656209319 ad='5d54b0f0'
SELECT /*+ USE_NL(store) USE_NL(dn) INDEX(store EI_ATTRSTORE) INDEX(dn EP_DN)
ORDERED */ dn.entryid,store.attrname, NVL(store.attrval,' '),
. . . etc . . .
```

Then, we compared the full SQL text in the raw trace with the one in the TKProf. We verified this was a match, so we were sure that what we saw in TKProf is the same high-CPU statement that was seen in the statspack report.

At this point we saw that our V$SESSION queries, 10046 trace / TKProf, and Statspack all showed this query as being the CPU consumer. This agreement between these independent data collections gave us confidence that this single SQL statement was the root cause for the performance problems in the database.

Having a particular SQL statement in mind as being the cause for the CPU consumption, we might ask ourselves, “Why did this query cause problems after the patch was applied?” We can imagine other related questions:

1.) Was this query present before the patch was applied?
We don’t know since there are no statspack snapshots prior to the patch

2.) Was this query much more efficient (much less CPU / execution) prior to the patch; did its execution plan change?
Again, we don’t know if it was even present, but considering the hints on the statement (index, join type, and join order), its extremely unlikely that the CBO would choose a different plan.

3.) The instance needs to be tuned using the init.ora parameters
This very seldom yields significant results in cases like this, but we can quickly review the statspack report’s init.ora section for glaring problems.

4.) The query was present before the patch and some strange Oracle bug is involved that is making this query much more expensive
This is possible, but unlikely given the database version changed very little (from 9.0.1.4 to 9.0.1.5). To determine this, we’ll have to install 9.0.1.4 and 9.0.1.5 databases on a test system and run the same test query to compare.

[19] Unfortunately, *TKProf* doesn’t show the hash value of each statement.

5.) The query is due to the patch and needs to be tuned despite the efforts of the developer that hinted this query.
 Quite possibly true and really the quickest way forward at this point.

Which scenario is right? We won't know unless we talk to the application developers and find out if this query is new or its execution plan is expected. In any case, we will keep our focus on the SQL statement since it clearly is responsible for most of the CPU time in the database.

Reviewing the Init.ora settings using Statspack

Finally, before we leave our discussion about Statspack, it is worth considering the init.ora settings shown in the last section of the statspack report to see if any unusual or undocumented settings are found that could affect query performance or CPU utilization. Here is a list of some noteworthy ones in this report:

```
End value
Parameter Name                Begin value                       (if different)
----------------------------- --------------------------------- --------------
_optim_peek_user_binds        FALSE
compatible                    9.0.0
…
db_block_size                 8192
db_cache_size                 150994944
…
pga_aggregate_target          33554432
…
sga_max_size                  588015404
…
sort_area_size                524288
timed_statistics              TRUE
…
workarea_size_policy          AUTO
```

The parameters that may affect performance are:

`_optim_peek_user_binds = false`

This setting determines whether the optimizer will "peek" at bind values when a statement is parsed. Potentially better plans could result with this set to true. No one could recall why this was set. The application developer did not recommend it. However, it's unlikely this would solve the immediate problem with the identified query since that query is thoroughly hinted and the CBO wouldn't choose a different plan even if it knew the literal values.

`pga_aggregate_target = 33554432`

This parameter could affect the way the optimizer costs certain operations such as hash joins and influence the CBO's execution plans. It could also affect the efficiency of those operations at runtime depending on whether there was sufficient memory available. If insufficient memory is allocated, then Oracle may have to use temporary segments to complete certain operations (inefficiently). To determine this, one should check the following statistics in the report:

```
Instance Activity Stats for DB: IASDB  Instance: iasdb  Snaps: 742 -752

Statistic                                      Total     per Second    per Trans
--------------------------------- ------------------ -------------- ------------
workarea executions - onepass                     14            0.0          0.0
workarea executions - optimal                 59,740           37.9         15.3
```

Ideally, all operations are performed in memory – shown in the statistics for “workarea executions –optimal”. Onepass and multipass operations indicate some use of temporary segments; some small amount of onepass operations is tolerable, but multipass operations are to be avoided if possible.[20]

`timed_statistics = true`

This indicates that operations are being timed in the database; we knew this already but it’s worth mentioning that without this, none of our response time analysis could be accomplished.

`workarea_size_policy = auto`

This is needed to make the pga_aggregate_target setting go into effect.

Some settings are known to affect CPU utilization when they are misused or are defective. We did not find any in this case, but here is what we looked for:

`_spin_count = <some number larger than default of 2000>`

This sets the number of times Oracle will try to acquire a latch before going to sleep for a period of time in the hope that when it wakes up, the latch will be free. If latches are held for a short period of time, this optimizes performance by avoiding a premature sleep. High CPU consumption can result if those seeking silver bullets for a latch contention problem set this value too high.

Our review of init.ora parameters doesn’t reveal anything that would explain the high CPU consumption. It looks like we’ll have to tune this query to use less CPU or find out if the application could execute it less frequently.

[20] These statistics should be taken into account based on 1) evidence of too many operations going to disk and slowing things down (wait events “direct path read” or “direct path write”), or evidence of inefficient query plans that may be caused by insufficient memory at runtime.

5. Tune the Query

To Tune or Not To Tune

At this stage we have identified a specific SQL statement as the cause of the CPU consumption and overall performance problems. Our next task is to find ways to reduce this query's CPU consumption. We have two possible approaches for doing this:

1.) Execute the query less often (maybe the application patch is unintentionally executing the query more often).
2.) Tune the query to make it use less CPU per execution

We analyzed the execution patterns and the value of bind variables in our extended SQL trace and found no evidence of repeating queries for the same values or other behavior that would suggest "over execution" of the query. The application developers looked at application log files and came to the same conclusion: the query is executing the proper number of times for the incoming requests.

This leaves us with the need to tune the query and make each execution more efficient.

Query Tuning Strategy

Query tuning is typically a non-trivial effort, especially when we have a query with various subqueries involved and a schema that is unfamiliar to us. One approach we could take is to recognize that the CBO may actually be able to find a better plan if only it were given a *fair* chance. We say "fair" because in this query the CBO's hands were effectively tied: it had no control over the access paths (INDEX hints were used), join types (USE_NL hints), and join order (ORDERED hint). What if the developer simply made a mistake and chose the wrong sets of hints?

At this point we could apply a tuning technique that might be thought of as the "Judo" [21] query tuning technique: we will use the CBO's effort and strength to tune the query by giving it more information about the query. We will also explore using session parameters to nudge the CBO to alternative paths.

For this technique to work, we want the optimizer to have all the information about the values in the predicate as possible. The more it knows about these values, the more accurate will be its selectivity estimates, and ultimately its estimate of the cost of obtaining rows for each step of the plan. Selectivity calculations are greatly influenced by the statistics collected for the objects in the query, especially the perceived distribution of data. By default, the optimizer assumes a uniform distribution of data within a column. When histograms are present, the optimizer may accurately determine the selectivity of the query despite *uneven* data distribution.

[21] Judo techniques stress the importance of balance to win a fight; in this case we observe the CBO is too off-balance to use its strengths. Others might call this method the "lazy tuner's" technique.

However, even with proper histograms to correctly estimate the selectivity for individual predicates, the optimizer still makes the assumption that predicates are independent of each other when computing the selectivity of the entire query. This is not always true and can lead to gross errors in the selectivity calculations. To mitigate this effect we can use *dynamic sampling* to let the CBO apply the predicates in the query and count how many rows are expected from each rowsource. This permits the CBO to obtain very accurate selectivity values and leads to an execution plan that has realistic cardinality estimates[22] and an efficient plan.

In addition to using dynamic sampling, one can also attempt to get a better plan by using parameters such as FIRST_ROWS_K or OPTIMIZER_INDEX_COST_ADJ to bias the optimizer. These should be used only at the SESSION level to give the CBO different paths to cost that it might have otherwise avoided. Once a better plan is found, one can influence the plan using hints or outlines (The use of parameters such as OPTIMIZER_INDEX_COST_ADJ at the instance level are strongly discouraged because it could have detrimental effects on other plans that are performing well and could make future tuning efforts very difficult).

To implement the "Judo" technique, and give the CBO a good fighting chance at finding a better plan, we ensured statistics were collected (large sample size and histograms) and changed the test script to do the following:

- Removed all of the hints from the SQL
- Replaced the bind values with literals[23]
- Used *dynamic sampling*

Keep in mind that our intentions were to see if we could obtain a better plan, we weren't going to require the use of literals or dynamic sampling in production. We simply wanted to know *if* there was a better plan that was available to us quickly with minimal effort.

[22] This leads to the technique of diagnosing a bad execution plan by looking for deviations between the *estimated cardinality* and the *actual cardinality* found when the query was executed and traced.
[23] We could have also left the bind variables in the query and enabled _optim_peek_user_binds=true (it was disabled in the init.ora) for our session and forced a hard parse by altering the query text slightly (add a comment). We chose to use the literals directly to remove any *potential* shortcomings in with bind peeking.

Here is the modified test case script:

“dynamic sampling” didn’t officially exist in 9.0.1; these are undocumented parameters to make it happen

```
alter session set events '10046 trace name context forever, level 12';

alter session set "_dyn_sel_est_on" = true;
alter session set "_dyn_sel_est_num_blocks" = 1000;

select /*~+ USE_NL(store) USE_NL(dn) INDEX(store EI_ATTRSTORE) INDEX(dn EP_DN)
  ORDERED */ dn.entryid,store.attrname, NVL(store.attrval,' '),
  NVL(store.attrstype, ' ')
-- no hints, use literals 1
FROM
 ct_dn dn, ds_attrstore store WHERE dn.entryid IN (  (SELECT /*~+ INDEX( at1
  VA_objectclass ) */ at1.entryid FROM CT_objectclass at1 WHERE at1.attrvalue
   = 'referral') UNION ((  (SELECT /*~+ INDEX( at1 VA_mail ) INDEX( at2
VA_objectclass
  ) */ at1.entryid FROM CT_mail at1, CT_objectclass at2  WHERE at1.attrvalue
  = 'gwiz@u.edu' AND at2.attrvalue  = 'orclmailuser' AND at2.entryid =
at1.entryid) UNION (SELECT
  /*~+ INDEX( at1 VA_mail ) INDEX( at2 VA_objectclass ) */ at1.entryid FROM
  CT_mail at1, CT_objectclass at2  WHERE at1.attrvalue  = 'gwiz@u.edu' AND
  at2.attrvalue  = 'orclmailgroup' AND at2.entryid = at1.entryid) )) ) AND (
(dn.parentdn
  like 'cn=oraclecontext,cn=products,cn=emailservercontainer,%'  ESCAPE '\' OR
(dn.rdn = 'cn=emailservercontainer' AND dn.parentdn =
'cn=oraclecontext,cn=products,')) ) AND
  dn.entryid = store.entryid AND dn.entryid >= 1000 AND
  store.attrname NOT IN ('member', 'uniquemember');

select 'end' from dual;
```

This causes SQL*Plus to close the previous cursor and make the runtime plan appear in the 10046 trace.

```
alter session set events '10046 trace name context off';
```

You might have noticed some unusual parameters in the script. In order to force the use of dynamic sampling in this version of the database, we had to resort to some undocumented parameters:
`_dyn_sel_est_on = true`: permit dynamic sampling
`_dyn_sel_est_num_blocks = 1000`: the number of blocks to sample

In later versions, we could enable dynamic sampling using:
`optimizer_dynamic_sampling = 10`[24]

[24] Level 10 will sample all of the blocks in the table; lower values will sample a smaller number of blocks. We chose 1000 blocks because this represented about ¼ of the blocks in the table and was enough to find a good plan.

The hints were easily disabled by replacing the "/*+" with "/*~+" and the literals were inserted in place of the bind values.

After executing the test case and producing another TKProf report, we see the following *improved* plan:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.02       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        4      0.00       0.00          0         63          0          43
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        6      0.02       0.01          0         63          0          43

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 141

Rows     Row Source Operation
-------  ---------------------------------------------------
     43  TABLE ACCESS BY INDEX ROWID DS_ATTRSTORE
     45   NESTED LOOPS
      1    NESTED LOOPS
      1     VIEW VW_NSO_1                                   CBO chose a different index
      1      SORT UNIQUE                                    (ST_OBJECTCLASS)
      1       UNION-ALL
      0        TABLE ACCESS BY INDEX ROWID CT_OBJECTCLASS
      0         INDEX RANGE SCAN (object id 55349)
      1        NESTED LOOPS
      2         TABLE ACCESS BY INDEX ROWID CT_MAIL
      2          INDEX RANGE SCAN (object id 55355)
      1         INDEX RANGE SCAN (object id 55348)
      0        NESTED LOOPS
      2         TABLE ACCESS BY INDEX ROWID CT_MAIL
      2          INDEX RANGE SCAN (object id 55355)
      0         INDEX RANGE SCAN (object id 55348)
      1     TABLE ACCESS BY INDEX ROWID CT_DN
      1      INDEX RANGE SCAN (object id 55345)
     43    INDEX RANGE SCAN (object id 55340)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       4        0.00          0.00
  SQL*Net message from client                     4        1.25          1.29
```

This plan is significantly better than the previous one. We observe:

- The CBO chose a different index than the hinted plan
- The total gets for this query is 63 compared to about 2,600 for the previous one
- The amount of CPU used per execution is less than 10 milliseconds (below the 10 millisecond resolution in the trace file) compared to approximately 206 milliseconds of CPU per execution

Why did the use of the ST_OBJECTCLASS index improve the performance of this query?

This index included the two columns that were in the predicates of the various subqueries (ENTRYID and ATTRVALUE) instead of having only one column (ATTRVALUE). The ATTRVALUE column used in the VA_OBJECTCLASS index was actually a less selective column (meaning there were few distinct values for this column, 104 out of 829,475 rows actually) so the range scan on that index had to read many index blocks based solely on ATTRVALUE and look for rows matching the value of ENTRYID.

The ENTRYID column had 196,303 distinct values and therefore the ST_OBJECTCLASS index allowed Oracle to navigate directly to a leaf block containing both desired values and scan from there.

We were fortunate that our technique yielded a better execution plan quickly[25]. Once we had a better plan, this question arose: "what can we do to ensure the application's queries use this plan?" We have the following potential solutions available to us:

Resolution Approach	Ease of Implementation	Risk
Change the application to use a different hint on the query	Change is easy, but may take time to go through 3rd party change control process	Very low; only specific query is affected
Create a stored outline for the query with a better plan chosen[26]	Implementing a stored outline is easy, but capturing a stored outline for the exact query text (with the hints) while it uses the better plan is difficult.	Low; the outline would only affect the query
Change the database's init.ora file to use dynamic sampling and/or other settings that will produce a better plan.	Easy; possibly some downtime involved.	High. Setting init.ora values at the instance level may change some good plans to bad. Dynamic sampling parameter would impact every query when hard parsed (high parse CPU and hot blocks).
Recreate the VA_OBJECTCLASS index to contain all of the columns needed to satisfy the query.	Easy; some downtime to rebuild the index. Index is small (5000 blocks) and won't take long to rebuild.	Low/Moderate; the index change may change some queries plans, but probably will improve rather than regress performance.

[25] If the CBO still hadn't chosen a better plan, we would have carefully considered each column involved in the various predicates and ensured that indexes were available for them. We might also have considered re-creating indexes to add columns in the select lists and further reduce I/Os against the tables.

[26] Oracle10g introduces SQL Profiles that can be generated by the SQL Tuning Advisor. Profiles are more flexible than outlines because they provide additional information to the CBO for getting accurate costs and hence better plans.

One may wonder if rebuilding the existing VA_OBJECTCLASS index might improve performance. After all, isn't it possible that this index is fragmented with many deleted rows and Oracle has to read many sparsely populated index leaf blocks just to satisfy the query? It is possible but unlikely because the patch application did not perform any DML on these tables (we learned this from the developer). But, just to confirm, we investigated the index statistics to be sure if this would help or not:

```
ANALYZE INDEX va_objectclass VALIDATE STRUCTURE;

SELECT name, height, blocks, lf_rows, del_lf_rows, lf_rows_len, del_lf_rows_len
FROM index_stats

NAME                HEIGHT     BLOCKS    LF_ROWS DEL_LF_ROWS LF_ROWS_LEN DEL_LF_ROWS_LEN
--------------- ---------- ---------- ---------- ----------- ----------- ---------------
VA_OBJECTCLASS           3       4352     836941        2428    19200775           58203
```

This query shows us that only 2,428 leaf rows are deleted in this index. The "bad" part of our execution plan is returning 64,430 rows, so in the extreme, worst case that all of the deleted rows happened to be in the range returned by our test query, the deleted rows only account for about 4% of those rows. These deleted rows are more likely spread around the entire index and the actual number of deleted rows that were skipped is much smaller than 4%. Simply rebuilding the index wouldn't have improved the query's performance (and, it might have wasted time and reduced the customer's confidence in our ability to solve this problem!).

Instead of just rebuilding the index, we chose to *drop* and *recreate* the VA_OBJECTCLASS index to include the ENTRYID and ATTRVALUE columns. We chose this solution because it was easy to do, required no application change (the SQL with the same hints will still work), and presented a low risk of negatively impacting the performance of the application. There was a possibility that this new index would be seen by the CBO as bigger and more expensive to use in some queries, thereby changing some plans, but this was a small risk since most queries were hinted. In any case, this confirms the importance of *verifying* the performance improvement and detecting any regressions that might have been caused by the solution.

Solution Verification

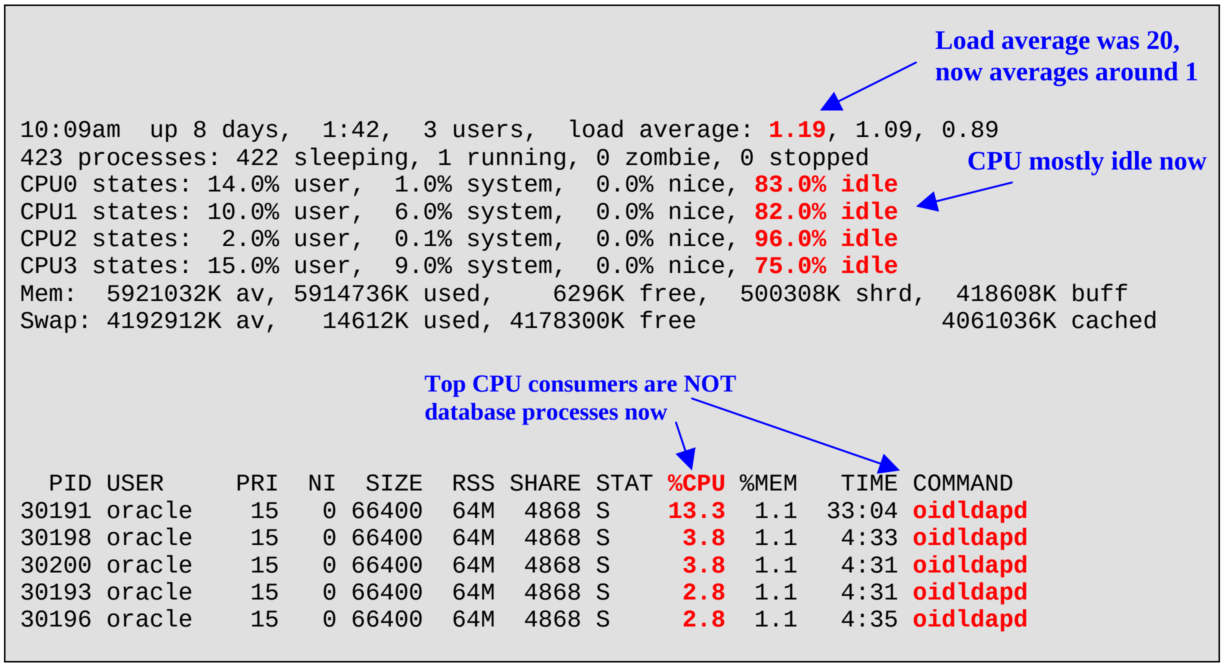
After dropping and recreating the VA_OBJECTCLASS index on the *test* instance to include the columns we needed, we re-ran the tests (on the customer's test system) and verified that the performance was improved as we expected. At the next planned outage, the index was rebuilt in production.

Subsequent analysis from the application development staff found the query had NOT been in use prior to the application patch upgrade, so in effect, this was a new, poorly tuned query.

Conclusion and Learnings

With the index rebuilt, we needed to observe the system to see if this solved the problem. During a time of peak load with mass mailings occurring, the customer said that the run queues on the system became less than 1. The system was now running as good as or better than before the patch was installed.

To verify these results, we reviewed OS statistics and statspack reports. Here is a typical display from *top* after the patch was applied:



```
10:09am  up 8 days,  1:42,  3 users,  load average: 1.19, 1.09, 0.89
423 processes: 422 sleeping, 1 running, 0 zombie, 0 stopped
CPU0 states: 14.0% user,  1.0% system,  0.0% nice, 83.0% idle
CPU1 states: 10.0% user,  6.0% system,  0.0% nice, 82.0% idle
CPU2 states:  2.0% user,  0.1% system,  0.0% nice, 96.0% idle
CPU3 states: 15.0% user,  9.0% system,  0.0% nice, 75.0% idle
Mem:  5921032K av, 5914736K used,    6296K free,  500308K shrd,  418608K buff
Swap: 4192912K av,   14612K used, 4178300K free                 4061036K cached


  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
30191 oracle    15   0 66400  64M  4868 S    13.3  1.1  33:04 oidldapd
30198 oracle    15   0 66400  64M  4868 S     3.8  1.1   4:33 oidldapd
30200 oracle    15   0 66400  64M  4868 S     3.8  1.1   4:31 oidldapd
30193 oracle    15   0 66400  64M  4868 S     2.8  1.1   4:31 oidldapd
30196 oracle    15   0 66400  64M  4868 S     2.8  1.1   4:35 oidldapd
```

This clearly shows that the overall load on the machine was much lower and only about 20% of the CPU was utilized. We know from talking to the administrator that users were getting excellent response times even when the system was sending out mass mailings.

We then looked at a statspack report during the same time as the *top* data above (which was during a mass mailing). We looked at the "Top 5 Wait Events" section first:

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                                Wait     % Total
Event                                                Waits  Time (s)   Wt Time
-------------------------------------------- ------------ ----------- -------
db file sequential read                           329,339          34   25.17
db file scattered read                            377,374          32   24.08
latch free                                          2,245          24   17.67
log file parallel write                            15,532          18   13.50
log file sync                                      10,481          15   11.60
```

We saw that most waits were for I/O and some latching. But, should we have tuned these? As before, these shouldn't be addressed until we account for the CPU usage. In the excerpt below, we have added the CPU component to the wait components and recomputed the "% Total Time".

Event	Waits	Time (s)	% Total DB Time
CPU	N/A	963	88.67
db file sequential read	329,339	34	3.13
db file scattered read	377,374	32	2.95
latch free	2,245	24	2.21
log file parallel write	15,532	18	1.66
log file sync	10,481	15	1.38

Clearly, sessions are spending most of their time on CPU; if performance *were* still a problem, we can easily see that I/O events shouldn't be the main focus of our tuning effort. We know from the *top* command's output that even though database sessions are spending almost 89% of their time on CPU, its still a minimal amount of CPU usage overall.

Let's look now at the SQL statement we tuned (hash value 1656209319):

```
                                                     CPU      Elapsd
  Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- --------- ----------
     24,401,160          120      203,343.0   62.3       80     92.21 2815034856
SELECT /*+ USE_NL(store) INDEX(store EI_ATTRSTORE) FIRST_ROWS */
 dn.entryid, store.attrname, NVL(store.attrval,' '), NVL(store.a
ttrstype,' ')  FROM ct_dn dn, ds_attrstore store WHERE dn.entryi
d in ( (SELECT /*+  FIRST_ROWS */ at1.entryid FROM ct_cn at1  WH
ERE at1.attrvalue like '%' ) UNION (SELECT /*+ INDEX(at1 VA_obje

. . . <cut out many lines> . . .

      1,084,612       27,990           38.7    2.8       21     22.40 1656209319
SELECT /*+ USE_NL(store) USE_NL(dn) INDEX(store EI_ATTRSTORE) IN
DEX(dn EP_DN) ORDERED */ dn.entryid,store.attrname, NVL(store.at
trval,' '), NVL(store.attrstype, ' ') FROM ct_dn dn, ds_attrstor
e store WHERE dn.entryid IN (  (SELECT /*+ INDEX( at1 VA_objectc
lass ) */ at1.entryid FROM CT_objectclass at1 WHERE at1.attrvalu
```

The statement that we tuned was now responsible for only 2.8% of the total buffer gets and only 2% of the CPU used by the database (21 / 963). We confirmed that the "Gets per Exec" were now only 38.7 compared to approximately 2,600 before the index rebuild. That's a 99% reduction in logical reads!

Finally, it would be good to see some throughput metrics that convince us that we're doing at least a comparable amount of work (no tricks up our sleeves). Ideally, the throughput metrics would come from some part of the application itself, but in their absence, we can compare some database metrics:

Metric	Before	After	Percent Improvement
User calls / sec	86.99	155.08	78% more
Transactions /sec	2.48	4.18	69% more
Logical reads / execution	266.62	47.00	82% fewer
bytes sent via SQL*Net to client / sec	143,842,167	194,423.9	35% more
bytes sent via SQL*Net to client / logical read	2.58	17.87	593% more

We focus on metrics such as *user calls*, *transactions*, and *executions* because they imply throughput from the point of view of requests (or actions) per second. *bytes sent via SQL*Net to client / sec* provides a good metric that accounts for the amount of data returned to clients independent of execution plan efficiency.

Using *logical reads / executions* is a <u>bad</u> way to compare throughput because it is dependent on how efficiently data is retrieved (i.e., we're not trying to improve the throughput of our execution plans; we want *better plans*). On the other hand, *bytes sent via SQL*Net to client / logical read* will show us how much more efficient the database is after the tuning because (hopefully) fewer logical reads are required for each byte returned back to the client.

From the above metrics we see that the database performed approximately twice as many *user calls* and *transactions* after the index rebuild. The *logical reads* / *execution* were lower; naturally because now it took fewer logical reads per execution. If we had measured throughput using just *logical reads* we would have been sorely disappointed (and badly mistaken)!

Our next two statistics clearly show the throughput improvement: approximately 30% more bytes per second sent to the client and approximately six times more data sent for each *logical read* required to fetch the data (2.58 bytes / read before, 17.87 after).

Keep in mind that all of the throughput improvements came while *lowering* the overall load on the machine by 95% (load average of 20 down to 1).

In conclusion, we've followed the tuning process from the top level symptoms of high CPU consumption at the OS level, down through the symptoms in a statspack report, through a trace of one of the CPU-consuming sessions, finally reaching the point where we were able to tune the query by rebuilding an index. In the end, we were able to confirm the benefit of the query tuning effort by observing significantly reduced CPU consumption, significantly better throughput in the database, and most importantly, happy users!

Additional Resources

Documents

Gunther, Neil 2003. *Unix Load Average Part 1: How it Works*. http://www.teamquest.com/resources/gunther/display/5/index.htm . TeamQuest Corp.

Metalink Doc ID 164768.1, *Diagnosing High CPU Utilization*

Metalink Doc ID 232443.1, *How to Identify Resource Intensive SQL for Tuning*

Metalink Doc ID 187913.1, *How to Turn on Tracing of Calls to Database*

Metalink Doc ID 39817.1, *Interpreting Raw SQL_TRACE and DBMS_SUPPORT.START_TRACE output*

Metalink Doc ID 190124.1, *The Coe Performance Method*

Metalink Doc ID 312789.1, *What is the Oracle Diagnostic Methodology (ODM)?*

Milsap, Cary 2003. *Optimizing Oracle Performance*. Sebastopol CA: O'Reilly & Associates

Oracle Corp. 2001. *Statistics Package (STATSPACK) README (spdoc.txt)*. Redwood Shores CA: Oracle Corp.

Tools

Metalink DocID 301137.1, *OS Watcher User Guide*

Metalink DocID 352363.1, *LTOM: The On-Board Monitor User's Guide*

Eliminate 99% of the work in this case study by using Enterprise Manager in Oracle 10g; specifically the Automatic Data Diagnostic Monitor and the SQL Tuning Advisor. For more information, see: *Automatic Performance Diagnosis*, *Performance Diagnostics Demystified: Best Practices for Oracle Database 10g* and *Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques* on OTN.

Acknowledgements

The author wishes to thank Richard Sarwal, Connie Green, Michael Matagrano, and Ajay Keni for their collaborative efforts during the actual customer engagement.