The Oracle Magician

Spring, 2006 *OracleMagician.com* *Volume V, number 1*

From the Editor

Welcome to the new issue of our online magazine, The *Oracle Magician*! This quarterly newsletter focuses on various "tricks of the trade" in the Oracle world--from DBAs, architects, developers, designers, and report writers.

Thank you to the many notes and comments from readers of my book, *The Art & Science of Oracle Performance Tuning*. Your comments and suggestions are appreciated!

In this issue we present a case that really had me bewildered. Fortunately, there was an unusual way to solve this bottleneck that turned out to be really easy to implement. I have seen used this technique many times.

We also present a simple script to find the name of the object that is causing your query to wait.

As always, we invite articles from readers that have interesting performance ideas. Send all ideas to

Edtor@OracleMagician.com

Chris Lawson

Editor

Solve Your Performance Problems by Adding A Join!

By Chris Lawson

Performance specialists are always on the lookup for interesting tuning problems--especially the "impossible" ones. So let's temporarily forget about all those SQL problems solved by adding indexes, gathering statistics, or rewriting an "Exists" clause. It's much more fun to delve into the really tough challenges, preferably those running on large production systems with many angry users.

The Chase Begins!

I recently came across one such performance problem. The beauty of this problem is that it initially appeared to be "impossible" to fix (short of redesigning the application.) When I first saw this problem, I honestly thought I could do nothing to fix it. The solution, however, turned out to be very simple--with impressive performance gains.

In this paper, we'll show you the performance puzzle, and lead you through the solution. First, though, let's set the stage by seeing what was bothering the users.

The Frustrated Engineers

At the headquarters of a large engineering construction firm, project engineers use a custom application they call "Pipe-Plan." This application helps the managers optimize their project planning--personnel placement, scheduling, parts cataloging, etc.

Continued on page 2

Table of Contents

Continued from page 1

One important function lets an engineer check the parts usage at all 1000 job sites. The user simply enters the part number or description, and the program displays the desired information. Unfortunately, the project engineers were seeing a delay of about 30 seconds each time they ran this function. This delay might not seem like a lot, but in contrast, most of the other functions returned in just a few seconds.

The Essence of the Problem

It was easy to isolate the problem SQL. The SQL in this query was well designed, with appropriate indexes on the underlying table. Here's what the SQL looked like:

Select Sum(In-spec), Sum (Out-of_Spec)
from Bill_of_Materials BOM
Where Part_Number = 'PIPE-100-INCH';

Note that the ***Bill_of_Materials*** table is huge--about 300 million rows. This means that very few of the table blocks will be cached. So if we need to retrieve 1,000 rows (one for each job site), there will be nearly 1,000 disk reads. In addition, we expect there will be some disk i/o for the index lookups as well.

A quick check of the V$Sql view yields the actual empirical data for the query in question. These statistics match nicely the above estimate of disk usage. In fact, oracle needs roughly 1,500 disk reads to execute the query for one part.

The above statistics confirmed that we had indeed found the root cause of the performance problem. The question is, how could we either eliminate or expedite these disk reads?

Possible Solutions

There are a few innovative ways we might speed up the above query. For instance, one way is to eliminate the need to query the huge *Bill_of_Materials* table. We could preprocess data, so that the pipe information would be available without reading this gigantic table. We might make a materialized view, refreshed overnight, that contains the desired information for all parts. Then, our problem SQL could query the materialized view instead. The materialized view would be far smaller than the *Bill_of_Materials* table, so many more blocks would likely be cached. So, disk i/o would go down drastically.

This first idea was rejected because the materialized view would frequently be out-of-date. For instance, it would not contain any parts recently added by another analyst. The engineers would thus not see these new parts until the next materialized view refresh.

A second possibility would be to perform the troublesome processing in the *background.* The engineer would enter the Part id, then press 'Go." The screen would return immediately so that the user could perform some other task. The requested information would be available shortly afterwards. This second approach was rejected due to the cumbersome process that the analysts would have to follow. Also, the change would involve considerable design changes to the application.

Fortunately, there was yet another way to solving this tough performance problem. It turned out to be very simple to implement, and transparent to the users. Here's what we did.

Our Approach

Our solution to solving the disk i/o bottleneck was not to avoid the disk i/o, but to simply get it over with sooner. The usual methods for speeding up table access would not work, however. For instance, we certainly couldn't perform a full scan of the *Bill_of_Materials* table. Even with a high degree of parallelism, the delay would be huge. Clearly, we needed to perform lots of single reads, but had to somehow get a bunch of processes working together, sharing the work.

One possible way to get multiple processes going would be to manually launch a bunch of Sql*Plus sessions, then divide up the work somehow. Assuming there is spare cpu and disk

Continued on page 3

Continued from page 2

capacity, this would be like "multithreading" the problem SQL. Although this might involve a major design change to the form, it would likely speed things up a lot.

A Strange (but *Better*) Way

It turns out there's a simple way to initiate multithreading--without resorting to redesign of the application. We will indeed use Oracle parallelism, but in the *opposite* way that it is usually employed. Whereas parallelism usually involves full scans of tables (or indexes), we will coax it to do lots of single block reads--*sequential* reads, in Oracle terms. This seems crazy, but it actually works. Here's how we do it:

Step 1

First of all, if we're going to subdivide the work into multiple threads, the optimizer has to have a driving table that encompasses the total number of units to process--in this case, the large number of job sites. So, we'll simply add the table, ***Job_Sites***, which is a list of all 1000 locations, and join it to our ***Bill_of_Materials*** table. Here's our new SQL;

```
Select Sum(In-spec), Sum (Out-of_Spec)
 from Job_Sites JS,  Bill_of_Materials BOM
 Where Part_Number = 'PIPE-100-INCH'
 And JS.Site# = BOM.Site#;
```

Step 2

The next step is to invoke Oracle's parallelism in our special way. We use the following hint:

```
/*+Leading (JS) Parallel (JS 30) Use_NL (JS BOM) */
```

The SQL hints above instructs the optimizer to start 30 threads beginning with the ***Job_Sites*** table. Furthermore, we want each thread to perform a nested-loop join to the huge *Bill_of_Materials* table. Here's the final SQL:

```
Select
/*+Leading (JS) Parallel (JS 30) Use_NL (JS BOM) */
Sum(In-spec), Sum (Out-of_Spec)
from Job_Sites JS,  Bill_of_Materials BOM
Where Part_Number = 'PIPE-100-INCH'
And JS.Site# = BOM.Site#;
```

With the above SQL construction, Oracle automatically starts up separate sessions--just as if we manually started a bunch of Sql*Plus sessions. The key to this technique is that we specify the parallelism on the *Job_Sites* table--that is the *small* table! That's because we want all the processes to begin on that table. Note that this is the *opposite* of what we normally do with Oracle Parallelism, where the parallel hint goes on the huge table.

Surprisingly, tests show that we can successfully launch a huge number of threads without contention. In fact, the optimum number is far more than the parallelism degree we would normally specify. Sample tests confirm that using 30 threads (or more) on a server with 16 cpus works nicely. Admittedly, we don't quite get 30x improvement, but close. The actual measured gain is about 25x.

Testing

When our problem SQL is running, we should see 30 processes, each performing a join to the ***Bill_Of_Materials*** table. So if we look at the wait events for these children, we should see most children waiting on a *sequential read* of ***Bill_of_Materials***. Here's a script to see what the threads are waiting on:

```
select x.server_name , x.pid as x_pid,
x.sid as x_sid , w2.sid as p_sid ,
v.osuser, v.schemaname, program ,
w1.event child_wait, w2.event parent_wait
from  v$px_process x , v$lock l , v$session v ,
v$session_wait w1, v$session_wait w2
    where x.sid <> l.sid(+)
    and  to_number
(substr(x.server_name,2)) = l.id2(+)
    and  x.sid = w1.sid(+)
    and  l.sid = w2.sid(+)
    and  x.sid = v.sid(+)
    and  nvl(l.type,'PS') = 'PS'
  and x.status not like 'AVAIL%'
  and w2.event not like 'SQL*Net%'
    order by 1,2;
```

Continued on page 4

Continued from page 3

Restrictions and Tips

This multi-threading technique is most useful for joining to huge tables containing hundreds of millions of rows. For smaller tables, the disk i/o will be much lower, so this technique will not be needed. Instead, you can simply use the conventional nested-loop or hash join methods.

Also, this technique will not work with queries containing clauses such as “WHERE IN” or “WHERE EXISTS.” The presence of "OR" conditions may also block proper operation. In these cases, Oracle's optimizer is apparently not able to divide the work up and coordinate the nested-loop slaves. Again, to ensure you’re getting the full benefit, it’s important to confirm that all the children processes are performing sequential reads. You can use the provided script to check this.

Wrap-up

After the job finishes, the V$Sql view will show the elapsed runtime, summed over all child processes. The clock time seen by the user will simply be *V$Sql.Elapsed_Time* divided by the number of executions (i.e., parallelism degree.)

For a fuller explanation of this method, see *Creative Multithreading Using Oracle Parallelism,* available on *www.oraclemagician.com.*

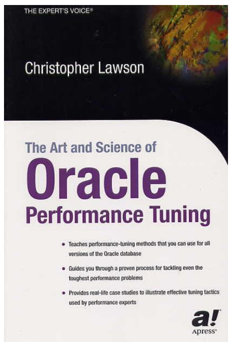
The Oracle Magician

Editor: Chris Lawson
OracleMagician.com

Copyright ©2006 *The Oracle Magician*

Updates and Notes

If you find these tuning issues interesting, I discuss performance tuning in greater detail in my book, *The Art and Science of Oracle Performance Tuning.* It is available at most large bookstores or online at Amazon.com.



Finding the Object Currently Being Read

There's a very easy way to see immediately what object a user is reading. This is especially valuable when trying to diagnose why a SQL is "stuck." Of course, you can get the file/block information from a wait event, but then you have to query the data dictionary to find the respective object. This method avoids that extra work:

```
SELECT DISTINCT
osuser, Sid, username,
Substr(program,1,19) PROG,
object_name, sql_text
From V$Session, V$Sql, dba_objects o
Where v$session.status = 'ACTIVE'
And username is not null
and o.object_id = row_wait_obj#
And v$session.sql_hash_value = hash_value
and v$session.sql_address  = v$sql.address;
```

Along with the object being read at that exact point in time, the above script lists the username, session, the program, and the SQL running.

Thanks to Kenneth Hu of Longs Drugs, Walnut Creek, for pointing out this idea.