The Oracle Magician

Fall, 2005 *OracleMagician.com* *Volume IV, number 3*

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 novel way to use Oracle Parallelism. I struggled with the name, since it's very different from how we normally use parallelism. I settled on the phrase "Multithreaded Parallelism." I have found this method quite useful and hope you find it a worthwhile addition to your performance tuning toolkit.

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

Edtor@OracleMagician.com

Chris Lawson

Editor

Creative Multithreading Using Oracle Parallelism

By Chris Lawson

Joins on huge tables

When working with Very Large Databases (VLDB), it is common to have a SQL query that must join to a huge table. In data warehouses, tables containing historical sales or inventory information can easily reach hundreds of millions, or even a few *billion* rows.

Joins to these gigantic tables present special challenges to the performance specialist. Methods that work very well on smaller tables break down, and performance will suffer accordingly.

In this paper, we'll use a simple example to understand the essence of the problem. Then, we'll present a special way to use Oracle parallelism to overcome the obstacles.

A Problem query

Assume we have a need to extract details for all sales for all products starting with the name "BEARING". To accomplish this, we query two tables: PRODUCTS and SALES. Table SALES (1 billion rows) stores all transactions for the last 3 years. Table PRODUCTS (10,000 rows) is the list of all sellable products.

Here's our query:

```
Select S.Sale_Amt,
```

Continued on page 2

Table of Contents

Continued from page 1

```
S.Discount, S.Emp_Discount, S.Clerk, S.Tax,
S.Register, S.Timestamp, S.Store
From PRODUCTS P, SALES S
Where P.part_no = S.Part_No
And P.Part_ Name like 'BEARING%'
```

We estimate that the above query will return roughly 1 million rows (100 products x 10,000 sales.) Note that this result set is only about 0.1% of the SALES table. We can assume that the table SALES has an index that matches the join condition (Part_No).

What can we do?

The issue for the performance specialist is this: How do we best extract 1 million rows from the SALES table? We might be temped to think that partitioning simplifies our task, but that is not the case. The reason is, the partitioning key will probably not match our query conditions. Our query filters rows based on *Part_No,* but most historical tables are partitioned on a *date-based* key.

To extract the million rows, we are faced with two dreary choices for the join method: nested-loop or hash join. Recall that a nested-loop join, with its "row by row" processing, is an excellent choice for a small result set, whereas the hash join is a better alternative for a large result set. Unfortunately, in our case, *both* choices are bad. Let's see why this is the case.

Nested-loop: This method will require numerous range scans of the index on SALES. This step will return the 1 million *Rowids* for the 1 million rows to be retrieved from SALES. Then, Oracle must perform 1 million table accesses. That's where the real delay occurs. We are reading a very small percentage of the SALES table--about 0.1%. Thus, most blocks that we fetch will just contain one of the desired rows. This means one read per row retrieved.

Secondly, due to the size of the SALES table, very few blocks fetched will be cached (although some of the index blocks will be cached). Therefore, we are faced with a cost of over 1 disk read per row retrieved. Based on actual tests conducted, I believe 1.5 reads per row is a good estimate.

A cost of 1.5 disk reads per row might seem okay, but it's really terrible. This equates to 1,500,000 single block reads. At a nominal disk read rate of 100 reads/sec, this causes a delay of 15,000 seconds, or 4.2 hours.

Hash join: This is usually the proper join method when a large result set is expected. In our case, the bottleneck becomes the full table scan of SALES. A reasonable size for this table is 10 million blocks. At a nominal read rate of 1000 blocks/sec, this means 13,000 seconds, or about 3.6 hours. (Note that we're only counting the disk access time in this example; the actual runtime will be somewhat longer.)

Neither option has worked too well so far. The next logical step is to crank up Oracle parallelism.

Parallelism - the 1st step

The traditional way to improve the full scan is to invoke Oracle parallelism. This will undoubtedly perform our full table scan much faster. I have found a degree of 6 works nicely on most servers. The SQL hint would thus be:

```
Select /*+parallel (S 6) */
```

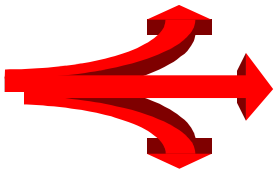
Tests show that a degree of 6 yields a gain of about 5x over the non-parallel version. This equates to about 43 minutes (again, just counting the disk i/o time.)

We've improved our query to 43 minutes, which is far better than our other options. This is quite an improvement--but wait, we can do much better!

Continued on page 3

Continued from page 2

Multithreaded Parallelism



There's another option that seems bizarre at first, but works incredibly well. It requires using Oracle parallelism in a strange way. With this scheme, we can avoid the full scan used by the hash join, while bypassing the inefficiencies intrinsic with the nested-loop method.

Let's review the nested-loop method. The reason it performed so poorly is that it is *single threaded*. In a nested loop, one session works on a single row at a time.

But what if we could expand the nested loop method so that it did lots of nested loop joins all at the same time? One way to invoke this "multi-threading" would be to somehow divide the table by key values, then manually start a bunch of Sql*Plus sessions to share the work. Then, we would combine the result sets.

It turns out that there's a simpler way to initiate multithreading--using Oracle parallelism. In this approach, Oracle automatically starts up separate sessions, just like with the hash join. But now, the sessions are all doing nested loop joins.

Here's the SQL hint to use Oracle parallelism to start multi-threading:

```
Select /*+parallel (P 30) Use_NI (P S) */
```

The key of this technique is that we specify the parallelism on the *Product* table--that is the *small* table! That's because we want all the processes to begin on that table, as we did for our regular nested-loop join. Note that this is the *opposite* of what we normally do with Oracle Parallelism, where the parallel hint goes on the huge table.

We have another change as well: We specify the ***Use_NI*** hint. This ensures that Oracle doesn't perform a hash join. With multithreading, we want the nested-loop method.

A Big Benefit

Surprisingly, tests show that we can successfully launch a huge number of threads without contention. The optimum number is far more than the parallelism degree we normally specify. Also, whereas regular parallelism normally launches processes equal to twice the degree specified (one set for reading; one set for processing results), this new method launches fewer processes--just 1x the degree.

Sample tests confirm that using 30 threads (or more) on a server with 8 cpus works nicely. We don't quite get 30x improvement, but close. The actual measured gain is about 25x. For our sample case here, that means a runtime of **10 minutes**.

Check Children Processes

When testing this method, you should confirm that the child processes are all doing nested loops.This is important, because it's very easy with Oracle parallelism to think that you're dividing the work, but in reality, only the parent process is reading, while the children are all waiting on inter-process communication.

You can confirm that you're multithreading properly by using wait events. Simply check what the children processes are waiting for (see the script on page 4.). When first started, all the children processes should be waiting on sequential reads.

Figure A shows an example of the wait events you should see when running multithreading. (Only two of the children sessions are shown; there really were 30.)

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

Possibilities and Limitations

This multi-threading technique is most useful for

Continued on page 4

Continued from page 3

huge tables containing hundreds of millions of rows. For smaller tables, it's not needed; you can use the conventional nested-loop or hash join with parallelism. Also, our example was based on a join of two tables, but the same principle should work on a simply query (no join), or with joins of more than two tables.

This technique will not work with queries containing clauses such as "WHERE IN" or "WHERE EXISTS." In these cases, Oracle is apparently not able to divide the work up and coordinate the nested-loop slaves. Again, to ensure you're getting real benefit it's important to confirm that all the children processes are performing sequential reads.

Figure A. Child Processes Performing Sequential Reads

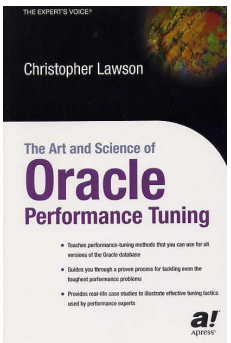
```
Name   Pid   Sid Parent OSUSER    PROGRAM      CHILD_WAIT                    PARENT_WAIT
---- ----- ----- ------ --------- ------------ ----------------------------- -----------
P000    22    23     18 Chris     oracle@axs ( db file sequential read       PX Deq: Execute
Reply

P001    30    48        Chris     oracle@axs ( db file sequential read       PX Deq: Execute
Reply
```

The Oracle Magician

Editor: Chris Lawson
OracleMagician.com

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.



Wait Events for Parallel Processes

```
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 as child_wait
        , w2.event as 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;
```