The Oracle Magician

Summer, 2007 *OracleMagician.com* *Volume VI, number 1*

From the Editor

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

Thank you for the notes & comments from readers of my book, *The Art & Science of Oracle Performance Tuning*. Your notes are appreciated! Thanks also to the mysterious stranger who nominated me for Oracle "Ace." I was accepted by the nominating committee.

In this issue we present a variety of useful tips presented by my colleagues at Wells Fargo. I also discuss an interesting performance-tuning case related to "hot" tables.

As always, we accept ideas or articles from reads that have interesting performance ideas.

Please send all ideas to Editor@OracleMagician.com

When is a Table Hot?

By Chris Lawson

The Mystery Select

I noticed that the statistics for two simple Sql (one select, one update) averaged about 5,000 logical reads per execution. This was puzzling since the size of the table was 24 blocks! How could it require 5,000 reads to extract a single row from a 24 block table? I confirmed the 24-block segment size by querying *Dba_Segments*, as well as by performing a full table scan.

I confirmed, via the *V$Sql_Plan* view, that Oracle was performing a full table scan each execution. This was indeed the correct execution plan, since there was no suitable index to use.

Each execution returned 1 row, so I knew the extra reads weren't due to some bulk/collect operation.

Checking the Pl/Sql procedure that drove this query, I confirmed that the two Sql were simply part of a big loop, that was executed about 10,000 times. One key point: There were no Commits until the loop was totally finished.

Some Ideas

The excessive logical reads, combined with the infrequent commits suggested that this job was a victim of a "hot table." This is an interesting phenomenon described nicely by Tom Kyte in *Expert Oracle Database Architecture.*

Table of Contents

Continued from page 1

What's Going On?

Here's what happens: When one or more sessions perform a large number of updates (without a commit), then any other session that needs the same block must look "backwards" into the undo to find the current block. As Tom Kyte explains, the block is "too new."

In our case, we were indeed performing thousands of updates of a block, but at first, I couldn't see the "hot table" scenario applied, because in our case the Sql is apparently performed by the *same* session. The hot-table problem should only apply to *other* sessions accessing the busy block, not the *same* session.

I decided to watch the batch job run the next morning. I prepared a simple Sql script to query V$sql and show the up-to-the-minute statistics on number of executions and number of buffer_gets for both Sql in question.

The Surprise

To my surprise, each execution now only required 24 blocks! The statistics now reflected the true cost of a full table scan of this small table. Of course, the Sql ran extremely quickly and rapidly reached a total of 5,000 executions.

While watching a bit longer, I saw something peculiar—the number of logical reads suddenly jumped from 24 per execution to about 10,000! Checking for other sessions that might be accessing the same table, I noticed that one other session was updating and selecting from this same small table. The key point was that this other session used the

Continued on page 3

A USEFUL WAIT EVENT

A colleague recently told me about an interesting wait event called

"SQL*Net break/reset to Client."

What's interesting about this event is that it increments when the session encounters some type of ORA- error. We found this event to be particularly useful when trying to identify application locking when the clause *NoWait* is used.

Recall that with NoWait, your session doesn't hang around, repeatedly trying to get access. Instead, it just continues to the next part of the program. So, unless you have sort of diagnostics built into the application, it can be difficult to detect when this block has occurred.

To see how this event works, I had one session perform a transaction a sample table, ***Chris***, but not commit. In another session, I performed this query: **Select * From Chris for Update Nowait**. Of course, I immediately got the error message,

ORA-00054: resource busy and acquire with NOWAIT specified

Checking the wait events in V$Session_Event confirmed that each time I ran the query (and got the ORA-54 error message), the SQL*Net break event incremented by 2.

Thanks to *Bosco Albuquerque* for passing on this information.

Continued from page 2

exact same procedure (and thus the identical Sql) to access the table. Everything quickly became clear. The small table was indeed a very hot table. The statistics were accurate, but were actually measuring something different than I thought. What I hadn't understood before was that the statistics reflected the results of *two* jobs. On *average*, the Sql took about 5,000 gets per execution.

For the one that ran first, each execution only required 24 gets. The second job, however, had to perform 100x more work *even though it used exactly the same Sql*. To maintain read consistency, the second job needed to find the version of the block that was committed at the time it began querying the active table.

The statistics for the Sql runtime thus reflected the cumulative numbers from both jobs. The first job really only performed 24 reads per execution; it was the second job, performing many thousands of logical reads per execution,

A Virtual Index?

I often face the question of whether the optimizer will make use of a new index. This can be tricky if the optimizer has several close choices and the proposed index would take a long time to create. Of course, if we have a small development database, we can simply create the index and confirm the execution plan. But that doesn't really prove that the index will be used on the "real thing."

There is a way, however, to predict index usage with an undocumented featured called a "virtual" index. First, note that virtual indexes are only enabled be for a session that sets a certain parameter:

```
Alter Session SET "_use_nosegment_indexes" = TRUE;
```

Now, create the virtual index, using the special keyword ***Nosegment***:

Create index Index_V on Table (Columns) **NOSEGMENT** compute statistics. With the index in place, simply use explain plan as usual to see if the new index gets selected. **Note**: Don't forget to drop the index when your testing is complete!

Thanks to Bhanu Gandikota for this.

TEMPORARY DATA FILE

Oracle employs a clever (?) trick when creating a data file for a locally-managed TEMP tablespace. When you add a data file for a TEMP tablespace, Oracle reserves only 128 kb, regardless of the size actually specified. That is, the OS doesn't actually *reserve* the space on disk until information is actually written to the file. This might seem like a petty distinction, but it isn't. You might be misled into thinking you have more disk space available than you really do. You could easily add more data files, not realizing your error.

To see the actual space used, run the ***du -sk*** command***.*** This shows the space actually reserved, in contrast to the larger size displayed via the ***ls*** command. (Normally, the size shown by du and ls would match.) One way to prevent this problem is to force the OS to pre-allocate the entire space that will ultimately be needed. We do this by *pre-creating* the temp file using the UNIX ***mkfile*** command. For example, to pre-create a 1 GB file (and fix the permissions), issue these commands:

```
/usr/sbin/mkfile 1000M /u01/oradata/db1/temp11.dbf
chmod 640 /u01/oradata/db1/temp11.dbf
```

After the file is pre-created, instruct Oracle to add a temp file, but this time, specify the ***REUSE*** option.

Thanks to Walter Schenk for this observation.

Continued from page 3

that distorted the statistics.

CAPTURING BIND VARIABLES

Here's a way in Oracle 10g to extract the bind variables for Sql currently running. We use the view ***V$Sql_Bind_Capture***, joined to V$Session. Note that the parameter **STATISTICS_LEVEL** must be set to **TYPICAL** or **ALL**.

Note that the select will often show list parameters used in the *Where* clause, even if they are technically literals in the original code. For instance, in my query below, the parameters "BACKGROUND" and "ACTIVE" would be included in the output (except that I include a clause to exclude my own queries.)

```
Col Username Format A15
Col Name Format A15
Col Value_String Format A10
Break On Sql_Text
Col Value Format A10

Select Distinct Sq.Sql_Text,
Spc.Name, Substr(Spc.Value_String,1,15)
VALUE
From V$Sql_Bind_Capture Spc,
V$Session S,V$Sql Sq
Where S.Sql_Hash_Value
=Spc.Hash_Value
And S.Sql_Address = Spc.Address
And Sq.Sql_Id=S.Sql_Id
and spc.was_captured='YES'
and s.type<>'BACKGROUND' and
s.status='ACTIVE'
and username not like 'LAWSON%'
order by 1,2;
```

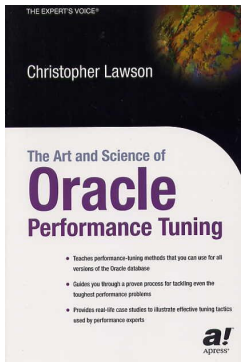
See results in Figure 1, below.

Thanks to Coskan Gundogar, Bosco Albuquerque, and Blair Foley

Figure 1. Bind Variable Results

```
SQL_TEXT                                    NAME       VALUE
------------------------------------------- ---------- ---------
select * Sample_Table Where Id_val = :1     1          6133721
```

You can purchase Chris' book on Amazon, or at your local book store:



Oracle ACE

OTN has a new feature called "Oracle ACE."

www.oracle.com/technology/community/oracle_ace