CHAPTER 20

Tuning an Oracle Database

In This Chapter

Collecting statistics

Tuning the SGA

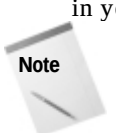
Tuning rollback segments

Tuning the redo log buffer

This chapter discusses some ways that you can monitor and tune the performance of a database instance. In addition to the topics listed above, you'll learn how to interpret the results you obtain from collecting statistics, and you'll learn some things that you can do to improve key performance metrics such as the buffer cache hit ratio.

Collecting Statistics

Oracle makes it easy for you to monitor a large number of database statistics by providing two well-known SQL scripts named UTLBSTAT and UTLESTAT. You can find these scripts in your $ORACLE_HOME/rdbms/admin directory.



The UTLBSTAT and UTLESTAT scripts are both SQL scripts. The full file names have the .sql extension, and are utlbstat.sql and utlestat.sql.

The UTLBSTAT and UTLESTAT SQL scripts allow you to collect statistics showing what is happening with your database over a period of time. To use them, you follow these steps:

1. Run UTLBSTAT. This script creates several work tables, grabs a snapshot of current statistics from several dynamic performance views (the V$ views), and stores that snapshot in the work tables.
2. Wait however long you wish. The reason you need to wait is that these two scripts collect statistics over a period of time. The statistics generated by these scripts will tell you about database performance during this specific period.

3. Run UTLESTAT. This script grabs a snapshot of current statistics from dynamic performance views and compares it to the snapshot recorded earlier by UTLBSTAT. It then generates a report based on the differences between the two scripts.

The value of UTLBSTAT and UTLESTAT lie in their ability to report statistics for a specific period of time. The dynamic performance views and the associated statistics that these scripts reference are available to you in real-time. However, the information returned from those views is cumulative from the point in time that you started the database. If your database has been up continuously for months, you might not notice small day-to-day variations by looking only at the current values in the views.

For example, if the buffer cache hit ratio drops suddenly in one day, that may not affect the overall ratio you get when you query the performance views directly. However, if you've been running UTLBSTAT and UTLESTAT each day, and recording the buffer cache hit ratio, the sudden drop will be painfully obvious.

To get the most value from these statistics, enable Oracle's timed statistics feature. This allows Oracle to track the amount of time spent on various events. Enable timed statistics by placing the following line in your database parameter file:

```
TIMED_STATISTICS = TRUE
```

Collecting timed statistics involves a small amount of overhead, but the value of the information you get makes the overhead well worth it.

No significant overhead is associated with the UTLBSTAT and UTLESTAT scripts themselves. Several short queries are executed when you run UTLBSTAT, and several more are executed when you run UTLESTAT. That's it. The scripts have no impact on database performance during the collection period.

Beginning the collection process

To begin the process of collecting statistics, you need to run UTLBSTAT to establish a baseline. The UTLBSTAT script connects to the database as the internal user, so historically, it has to be run from Server Manager. Beginning with the 8i release, you can also run the script from SQL*Plus.

When you run UTLBSTAT, it first drops and re-creates the working tables that it uses. If those tables don't happen to exist, you'll see several error messages scroll by on the screen. Don't be alarmed. It's okay if the DROP commands fail. The CREATE TABLE commands will succeed, and the work tables will be created.

After creating the work tables, several INSERT statements are issued to populate them. These INSERT statements grab a current snapshot of information from key performance views such as V$SYSSTAT, V$LATCH, and V$LIBRARYCACHE. This

information is stored in the work tables for later comparison with the ending values grabbed by UTLESTAT.

The most recent run of UTLBSTAT is always the starting point for the statistics generated by UTLESTAT. You can run UTLBSTAT many times in succession, but it is the last run that counts.

Listing 20-1 shows UTLBSTAT running against an Oracle8i database from SQL*Plus. The beginning and ending of the output is shown. Most of the middle has been omitted to save space.

Listing 20-1: **Running the UTLBSTAT script**

```
SQL> @utlbstat
SQL>
SQL> Rem ************************************************************************
SQL> Rem                 First create all the tables
SQL> Rem ************************************************************************
SQL>
SQL> drop table stats$begin_stats;
drop table stats$begin_stats
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> create table stats$begin_stats as select * from v$sysstat where 0 = 1;

Table created.

...

SQL> insert into stats$begin_stats select * from v$sysstat;

196 rows created.

SQL>
SQL> insert into stats$begin_lib select * from v$librarycache;

8 rows created.

SQL>
SQL> insert into stats$begin_latch select * from v$latch;

142 rows created.

SQL>
SQL> commit;

Commit complete.
```

Once UTLBSTAT has run, all you need to do is wait until the end of the time period for which you are collecting statistics, and then run UTLESTAT.

If you stop and restart the database after running UTLBSTAT, that will reset all the values reported by the dynamic performance views and will invalidate your statistics. For valid results, the database must be up and running during the entire time, beginning when UTLBSTAT is executed up until the time when UTLESTAT is executed.

Ending the collection process

To end the process of gathering statistics, run the UTLESTAT script. This script gathers another snapshot of statistics from the dynamic performance views, compares that against the original snapshot saved by UTLBSTAT, and generates a text file named report.txt that contains the results. The UTLESTAT script also drops the work tables.

If you run UTLESTAT without first running UTLBSTAT, you will get a lot of errors because the work tables won't exist. Any results that you get—and you probably won't get any—won't be valid.

Listing 20-2 shows UTLESTAT being executed.

Listing 20-2: **Running the UTLESTAT script**

```
SQL> @utlestat
SQL> rem
SQL> rem $Header: utlestat.sql 17-apr-98.15:26:01 kquinn Exp $ estat.sql

...

SQL> connect internal;
Connected.
SQL>
SQL> Rem ****************************************************************
SQL> Rem                Gather Ending Statistics
SQL> Rem ****************************************************************
SQL>
SQL>
SQL> insert into stats$end_latch select * from v$latch;

142 rows created.

SQL> insert into stats$end_stats select * from v$sysstat;

196 rows created.
```

```
...

SQL> drop table stats$end_waitstat;

Table dropped.

SQL> drop table stats$waitstat;

Table dropped.
```

After running UTLESTAT, you can review the results by editing or printing the file named report.txt. The file will be created in whatever directory was your current directory at the time you ran UTLESTAT.

Each subsequent run of UTLESTAT will overwrite the report.txt file. You need to save or rename this file if you want to compare the results of two different runs.

Interpreting the results

The report.txt file will be several pages in size, and it will contain literally hundreds of different statistics. These will be divided into the following sections:

- ✦ Library cache statistics
- ✦ Database statistics
- ✦ The dirty buffer write queue length
- ✦ Systemwide wait events
- ✦ Latch statistics
- ✦ Buffer busy wait statistics
- ✦ Rollback segment statistics
- ✦ Current initialization parameter settings
- ✦ Dictionary cache statistics
- ✦ Tablespace and datafile input/output statistics

Let's examine what you will see in each of the different parts of the report.

Library Cache Statistics

The *library cache* is an area in the system global area (SGA) where Oracle caches execution plans and parse trees for recently issued SQL statements. The library

cache also contains cached copies of PL/SQL code. The statistics indicate how well the cache is performing, and look like this:

```
LIBRARY      GETS GETHITRATI  PINS PINHITRATI RELOADS INVALID
------------ ---- ---------- ----- ---------- ------- -------
...
SQL AREA     9779        .916 29852        .969       4      62
TABLE/PROCED 4220        .892  8035        .882       9       0
TRIGGER        64        .656    74        .554       0       0
```

The LIBRARY column shows the different types of objects that Oracle is caching. The row for SQL AREA, for example, represents information about cached SQL statements. The GETS column refers to the number of times that Oracle looked up an object, while the PINS column refers to the number of times that Oracle executed it. The GETHITRATIO and PINHITRATIO columns (the names may be truncated in the report) indicate how often items were found in memory. In this example, pinned SQL statements were found in memory 96.9 percent of the time. The RELOADS column indicates the number of times an object had to be reloaded after being flushed from the cache.

Key items to look at here are the hit ratios and the reload counts. You want the number of reloads to be zero, if possible. If the reload counts are high, that's a good indication that you should increase the size of your shared pool. Low hit ratios might also be improved by a larger shared pool, or they may reflect an application that isn't issuing the same SQL statements over and over again. In this example, the PINHITRATIO for SQL statements is .969, which is considered good. (Anything .95 or higher is considered good.) For triggers, the PINHITRATIO is rather low, but given the low number of gets and pins, the low ratio could simply be the result of the triggers being loaded into the cache for the first time. As with most of the statistics, you'll get the most useful results after your database has been running for a while.

Database Statistics

The database statistics in the report summarize changes in a long list of system statistics during the period covered by UTLBSTAT and UTLESTAT. The results look like this:

```
Statistic                Total Per Transact Per Logon Per Second
------------------------ ------ ------------ --------- ----------
...
consistent gets         917670      9762.45   3855.76    1696.25
db block gets            20763       220.88     87.24      38.38
...
physical reads            3017         32.1     12.68       5.58
physical writes           2027        21.56      8.52       3.75
...
session logical reads   938433      9983.33      3943    1734.63
...
```

The statistic name is the name of the statistic as reported by the V$SYSSTAT view. The TOTAL column contains the difference in value for the statistic between the beginning and end of the period that was monitored. The other three columns break this difference down on a per-transaction, per-logon, and per-second basis.

One task you can perform with these statistics is to calculate the buffer cache hit ratio for the period being monitored. You can do that by using the following formula:

```
1 - physical reads / (consistent gets + db block gets)
```

Plugging the statistics shown here into this formula will get you a buffer cache hit ratio of .9967. This means that 99.67 percent of the time that Oracle went to read a database block, that block was already in memory. That's good. Buffer cache hit ratios are usually considered good if they are .90 or higher. A lower hit ratio is an indication that you may need to increase the size of the cache (by adjusting the db_block_buffers initialization parameter).

The Dirty Buffer Write Queue Length

The section of the report labeled "The dirty buffer write queue length" consists of only one number and indicates the average number of blocks that are waiting to be written at any given time. Lower values are better. If you find the value of this statistic is increasing, or if it is 10 or higher, you should probably look at reducing it. To reduce this value, you have to increase Oracle's write throughput. One possible way to do this is by increasing the db_block_simultaneous_writes parameter in your database parameter file. Another approach is to configure your database to have more database writer processes. You do this by increasing the db_writer_processes initialization parameter.

Systemwide Wait Events

Wait events are events that cause a process to wait. For example, a process might need to wait for a read to occur. Two sections contain information on wait events, one showing statistics for background processes and one showing statistics for nonbackground processes. Both sections look like this:

```
Event Name                    Count Total Time Avg Time
----------------------------- ----- ---------- --------
SQL*Net message from client    5256     418502    79.62
...
db file scattered read           93         43      .46
log file switch completion        2         41     20.5
single-task message               1         15       15
direct path write               162         14      .09
direct path read                 35         13      .37
control file sequential read     25         12      .48
...
```

The column titles are fairly self-explanatory. For each event, the report shows the total number of waits (the `Count` column), the total time spent waiting (`Total Time`), and the average wait time per event (`Avg Time`). The time is reported in hundredths of a second. For example, this report shows that 12/100ths of a second was spent waiting for a control file read to occur.

Obviously, the fewer the waits and the less time spent waiting, the better. Some values, such as the ones for the event named "SQL*Net message from client," will always be high. These don't represent a problem. However, you should investigate abnormally high values that aren't always high.

Latch Statistics

You'll find two sections containing latch statistics in the report. *Latches* represent locks on critical areas of memory, and *latch contention* can lead to significant performance problems. The first latch statistic section contains statistics for normal latch requests. The second section contains statistics for latch requests made in a `nowait` mode. The two sections are shown in Listing 20-3.

Listing 20-3: Latch statistics appear in two sections

```
LATCH_NAME            GETS MISSES HIT_RATIO SLEEPS SLEEPS/MISS
------------------ ------- ------ --------- ------ -----------
Active checkpoint      224      0         1      0           0
Checkpoint queue l    4907      0         1      0           0
Token Manager          126      0         1      0           0
cache buffer handl     234      0         1      0           0
cache buffers chai 1858071      0         1      0           0
cache buffers lru     8539      0         1      0           0
...

LATCH_NAME         NOWAIT_GETS NOWAIT_MISSES NOWAIT_HIT_RATIO
------------------ ----------- ------------- ----------------
cache buffers chai        4015             0                1
cache buffers lru           63             0                1
channel handle poo          25             0                1
...
```

The first section contains columns with sleep statistics. When a process requests a latch and that latch isn't available, the process will enter a sleep state where it waits for the latch. The second section doesn't contain the two sleep-related columns because it represents statistics for those times when a process requested a latch only if it was available immediately. In other words, the process wasn't willing to wait.

Two key performance indicators here are the number of misses and the number of sleeps. Ideally, you want both to be zero, indicating that processes are always able to get latches when they need them. The hit ratio represents the ratio of successful *gets* to the total number of *gets*. It should be as close to 1 as possible. If your hit ratio drops much below 1 and the number of sleeps increases, you may be experiencing latch contention.

Buffer Busy Wait Statistics

The buffer busy wait statistics section of the Report.txt file is significant only when the "buffer busy waits" statistic in the wait events section is high. Normally, this section of the report will be empty and will look like this:

```
CLASS               COUNT            TIME
------------------ ---------------- ----------------
0 rows selected.
```

If the "buffer busy waits" count is significant, that indicates contention for latches on one or more types of blocks in the buffer cache. The `CLASS` column reports the type of blocks for which waits are occurring. The `COUNT` column reports the number of times a block couldn't be accessed when requested because contention was occurring. The `UTLESTAT` report gives some suggestions for resolving problems with buffer busy waits.

Rollback Segment Statistics

The rollback segment section of the report provides you with information about rollback segment usage during the period covered by the statistics. It looks somewhat like the following example:

```
  UNDO_    TRANS_     TRANS_ UNDO_BYTES_   SEGMENT_
SEGMENT TBL_GETS TBL_WAITS     WRITTEN SIZE_BYTES ... SHRINKS
------- -------- --------- ----------- ---------- ... -------
      0        3         0           0     407552 ...       0
      2       90         0       40008     612352 ...       0
      3       72         0       32984     612352 ...       0
```

This sample output has been edited a bit to make it fit on one page. In the real report, the columns are much wider, and the column names aren't wrapped around onto two lines.

Two key values to look at here are `TRANS_TBL_WAITS` and the number of `SHRINKS`. The `TRANS_TBL_WAITS` column represents the number of times that a process had to wait to get a rollback segment. If this is more than a few percent of the value in the `TRANS_TBL_GETS` column, then consider adding more rollback segments to your database. The `SHRINKS` column represents the number of times that a rollback segment grew beyond its optimal size and was shrunk back again. There's a

performance impact from shrinking a rollback segment, so a large number of shrinks isn't good. To reduce the number of shrinks, increase the optimal size setting for your rollback segments.

Current Initialization Parameter Settings

The current initialization parameter settings section provides you with a list of the parameter settings in effect when you run `UTLESTAT`. Only the nondefault settings are listed:

```
NAME                                     VALUE
---------------------------------------- --------------------
audit_trail                              DB
compatible                               8.1.5
db_block_buffers                         8192
db_block_size                            2048
...
```

There's nothing in this section that you really need to look for. It simply serves as a reference should you ever compare two different reports.

Dictionary Cache Statistics

The dictionary cache section of the report tells you how efficient the data dictionary cache is in terms of how often dictionary information needs to be read from disk vs. how often it is found already in memory. The dictionary cache section of the report looks like this:

```
NAME            GET_REQS GET_MISS SCAN_REQ SCAN_MIS ...
--------------- -------- -------- -------- -------- ...
dc_tablespaces       441       18        0        0 ...
dc_free_extents      411      134      110        0 ...
dc_segments          803      190        0        0 ...
dc_rollback_seg      107        0        0        0 ...
...
```

In the dictionary cache section, you need to check only the dictionary cache hit ratio. You can compute that ratio by using the following formula:

```
1 - SUM(GET_MISS) / SUM(GET_REQS)
```

You have to sum the `GET_REQS` and `GET_MISS` values for all dictionary elements. There's really no sense in looking at the hit ratios for any one element because you can't adjust them on such a fine basis. Using just the data in this example, the dictionary cache hit ratio ends up being 0.806. That's not very good. Hit ratios of 0.95 or more are considered ideal. If your hit ratio is low, you can attempt to improve it by increasing the size of the shared pool. The `SHARED_POOL_SIZE` initialization parameter controls the size of the shared pool.

Tablespace and Datafile Input/Output Statistics

The last two sections in the report show the amount of disk input and output on a per- tablespace basis and then on a per-file basis. Both sections contain the same I/O-related information, as shown in Listing 20-4.

Listing 20-4: The tablespace and datafile input/output section

```
TABLE_SPACE      READS BLKS_READ READ_TIME WRITES BLKS_WRT ...
---------------- ----- --------- --------- ------ -------- ...
CHECKUP_HISTORY      3         3         0      3        3 ...
INDX                 3         3         0      3        3 ...
ITEM_DATA            3         3         0      3        3 ...
ITEM_PHOTOS          3         3         0      3        3 ...


TABLE_SPACE     FILE_NAME       READS BLKS_READ READ_TIME ...
--------------- --------------- ----- --------- --------- ...
CHECKUP_HISTORY E:\ORACLE\ORA...     3         3         0 ...
INDX            E:\ORACLE\ORA...     3         3         0 ...
ITEM_DATA       E:\ORACLE\ORA...     3         3         0 ...
ITEM_PHOTOS     E:\ORACLE\ORA...     3         3         0 ...
```

The READS and WRITES columns show the number of physical reads and writes for each of the tablespaces and datafiles. The BLKS_READ column shows the number of database blocks that were read. The BLKS_WRT column shows the number of database blocks written. The READ_TIME and WRITE_TIME columns (not shown here because of the page width) show the time spent reading and writing, respectively. The times are in hundredths of a second.

The key point that you want to look at here is the distribution of your I/O load. You should avoid concentrating large amounts of I/O on any one disk, because you can end up requesting more throughput than that disk can provide. If you see high concentrations of I/O in any one tablespace or on any one disk, take steps to distribute that load over multiple disks.

In a UNIX environment, you need to be cognizant of your disk configuration and how your mountpoints have been set up when you interpret this section. What looks like one disk may in fact be several. The reverse could also be true.

This data also provides an indication of how much sequential I/O (possibly in the form of tablescans) is occurring. Reads that are the result of index lookups tend to bring back one block of data at a time. Tablescans, on the other hand, cause several

blocks of data to be read in by each physical read. If the BLKS_READ value is high in relation to the READS value, that's an indication that tablescans may be taking place. It's not absolute proof, though, and tablescans aren't always negative. But if you see something that you aren't expecting, investigate it.

Tuning the SGA

The system global area (SGA) is a large block of memory that is shared by all the processes that make up an instance. It contains several key memory structures, including the database buffer cache, the shared pool, and the redo log buffer. These are illustrated in Figure 20-1.

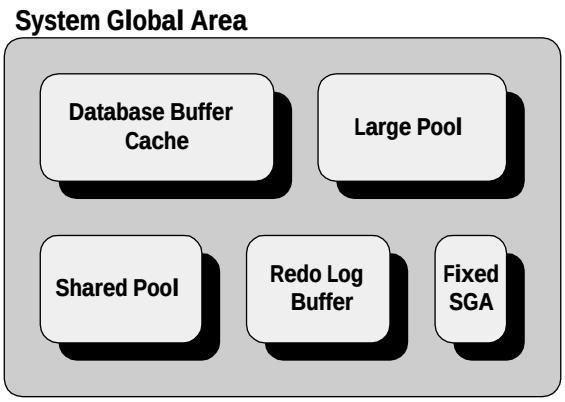
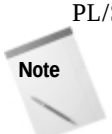


Figure 20-1: Memory structures in the SGA

The *database buffer cache*, often just referred to as the buffer cache, holds the most recently accessed database blocks in memory where Oracle can quickly access them if you need them again. The shared pool caches data dictionary information, PL/SQL code, and execution plans for SQL statements.



If you need a quick recap of Oracle8i's memory architecture, see Chapter 3, "Oracle8i Architecture."

Properly sizing the buffer cache and the shared pool is key to optimizing an Oracle database's performance. Changing the size of these areas is easy: You just change some initialization parameters. Knowing when to change them is the trick. This section will show you two key statistics that you can monitor over time to be sure that you have a reasonable amount of memory allocated to each.

The buffer cache hit ratio

You can use the *buffer cache hit ratio* (often just called the cache hit ratio) to determine how much benefit you are getting from your database buffer cache. I/O activity in a database is often concentrated on a relatively small number of blocks in comparison to the total number on disk. Oracle takes advantage of this by holding the most recently used blocks in memory. If a block happens to be in memory and Oracle needs to access it again, you've just saved an I/O operation. The buffer cache hit ratio tells you how often this is happening.

Computing the Buffer Cache Hit Ratio

To compute the buffer cache hit ratio, use the following formula:

```
1 - (physical reads / (db block gets + consistent gets))
```

You can get the values that you need for this formula by querying the V$SYSSTAT dynamic performance view. For example:

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name IN ('physical reads',
  4                 'db block gets',
  5                 'consistent gets');

NAME                     VALUE
------------------- ---------
db block gets             1806
consistent gets          71765
physical reads            4133
```

Using these figures, the cache hit ratio is computed as follows:

```
0.94 = 1 - (4133 / (1806 + 71765))
```

As a rule of thumb, hit ratios are considered good if they are 0.90 or higher. That's not an ironclad rule, though. If you have an application that simply isn't hitting the same indexes or data from one query to the next, you may never achieve 0.90.

The hit ratio that you get by querying V$SYSSTAT includes all activity since the instance was started. It may be better to run UTLBSTAT and UTLESTAT, and use the information from the resulting report to compute the cache hit ratio for a short period of time. You might, for example, choose a representative time during the day when the system is being used normally.

Improving the Cache Hit Ratio

If your hit ratio is low, you should at least attempt to improve it. The only way to improve it, short of possibly rewriting all your applications, is to increase the amount of memory allocated to the buffer cache. Tuning the cache hit ratio involves an iterative process consisting of these steps:

1. Compute the current hit ratio.
2. Increase the size of the buffer cache.
3. Compute the hit ratio again. If it improved, go back to step 2. If there was no improvement, then undo your change and stop.

The point of all this is that you should increase the size of your buffer cache only if each increase results in a significant improvement in the cache hit ratio. At some point, you will reach a size beyond which you see no improvement. That's the size you want for the buffer cache. Make it larger than that, and you are wasting memory that you could more profitably use for other purposes. You might, for example, increase the size of your shared pool so that it can accommodate more SQL statements.

Changing the Buffer Cache Size

The `DB_BLOCK_BUFFERS` initialization parameter controls the size of the buffer cache. You set this in your database parameter file, and it is read whenever you start the instance. The `DB_BLOCK_BUFFERS` parameter defines the buffer cache size in terms of the number of data blocks that it can hold. Here's a sample setting:

```
DB_BLOCK_BUFFERS = 1000
```

In this example, enough memory will be set aside in the SGA to hold 1,000 blocks. If your block size (`DB_BLOCK_SIZE`) is 8KB, then the total size of the buffer cache will be 1000 * 8KB, or 8,192,000 bytes.

Because the initialization parameter file is read only when you start an instance, changing the `DB_BLOCK_BUFFERS` parameter involves stopping and restarting the database. Follow these steps:

1. Change the `DB_BLOCK_BUFFERS` parameter in the initialization file.
2. Stop the instance.
3. Restart the instance.

By making the change first, you minimize the total downtime to just the few seconds that you need to type the `SHUTDOWN` and `STARTUP` commands.

The library cache hit ratio

Recall that the library cache is the area in memory where Oracle stores SQL statements in case they are needed again. Actually, it's not the SQL statements themselves that are so important, but rather, the parse trees and execution plans associated with those statements.

You may recall from Chapter 3 that most applications execute a relatively small number of distinctly different SQL statements. If you have 50 people banging away at an order-entry screen, the underlying SQL statements executed by the application are going to be the same from one order to the next. Oracle stores the execution plans for these statements in the library cache, checks each incoming SQL statement against the cached statements, and reuses execution plans whenever possible. The library cache hit ratio tells you how frequently reuse is occurring.

Deciding If Two SQL Statements Are the Same?

To gain the maximum benefit from the shared SQL area, you need to understand something about how Oracle decides if two SQL statements are the same. Here are the rules that Oracle follows when comparing two statements:

- The statement text must match exactly. Each character counts, and the comparison is case-sensitive.
- The objects referred to in the statement must be identical. If two statements select from a table named `DATA`, but the statements refer to tables owned by two different users, then they won't be considered the same.

Keep the first rule in mind when developing an application. For example, say that your developers code the following two statements into different screens of an order-entry application:

```
SELECT cust_name
FROM cust
WHERE cust_num = :1


SELECT cust_name FROM cust WHERE cust_num = :1
```

Because the line breaks are in different places, Oracle will see these as two distinctly different SQL statements. Both versions will end up being cached in the library cache. The impact on memory when just two versions of a SQL statement are used is minimal, but multiply this scenario by five or ten statements, with five or ten versions of each statement, and suddenly you are using 5–10 times the amount of memory that you really need. The more consistent your developers are when coding SQL statements, the better off you'll be.

One last issue worth mentioning is the use of bind variables in SQL statements. Both of the statements shown previously use bind variables named `:1`. Whether you name your bind variables with names or numbers doesn't matter, but it does matter that you use them.

Consider what would happen if our hypothetical order-entry system dynamically generated SQL statements for each customer lookup. You might get a series of statements like these:

```
SELECT cust_name FROM cust WHERE cust_num = 10332
SELECT cust_name FROM cust WHERE cust_num = 98733
SELECT cust_name FROM cust WHERE cust_num = 81032
SELECT cust_name FROM cust WHERE cust_num = 00987
```

Oracle will see each of these as a new statement, will parse it, and then will send it to the optimizer to generate an execution plan. Because bind variables aren't being used, you lose any benefit from the library cache. The moral of the story? Use bind variables whenever possible for frequently executed SQL statements.

Computing the Library Cache Hit Ratio

You can compute the library cache hit ratio by using the following formula:

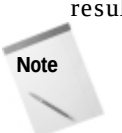
```
SUM(pins - reloads) / SUM(pins)
```

You can get this information directly, by querying the V$LIBRARYCACHE dynamic performance view. For example:

```
SQL> SELECT SUM(pins - reloads) / SUM(pins)
  2  FROM v$librarycache;

SUM(PINS-RELOADS)/SUM(PINS)
---------------------------
                 .99972407
```

A hit ratio close to 1 is good. You certainly wouldn't need to worry about this result. If it's less than .99, you should try to improve it.



The hit ratio that you get by querying V$LIBRARYCACHE includes all activity since the instance was started. If your instance has been running for a long time (days or months), you may want to compute the hit ratio over a shorter period of time. You can use the values from a UTLBSTAT/UTLESTAT report to do that.

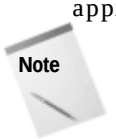
Improving the Library Cache Hit Ratio

If your hit ratio is low, or if it's lower than you would like, you can try to improve it by adding more memory to the shared pool. You do that by following an iterative process just like the one you use when you adjust the database buffer cache. The idea is to keep increasing the size of the shared pool until you reach the point where an increase has no effect on the hit ratio.

Before you increase the size of the shared pool, however, check to see how much of it is currently unused. You can do that by issuing this query:

```
SELECT * FROM v$sgastat WHERE name = 'free memory';
```

If it turns out that you have plenty of free memory in the shared pool already, then an increase in size probably won't improve the library cache hit ratio. If you have a lot of free space, the cause of a poor hit ratio is likely to be in the design of the applications that are running against your database.



If you're running a decision support system where each query is likely to be different no matter what you do, a low library cache hit ratio might just be a fact of life.

Changing the Shared Pool Size

You change the shared pool size by adjusting the value of the `shared_pool_size` initialization parameter. Chapter 3 discusses this parameter, but here are some sample values:

```
shared_pool_size = 50000000        50 million bytes
shared_pool_size = 10m             10 megabytes
shared_pool_size = 10k             10 kilobytes
```

You set this parameter in your database parameter file. When you change it, you need to stop and restart the instance for the change to take effect.

The dictionary cache hit ratio

The dictionary cache is stored in the shared pool and caches recently accessed data- dictionary information. Caching this information in memory greatly reduces the time and I/O necessary to parse SQL statements.

The dictionary cache is often referred to as the row cache.

You can determine the efficiency of the dictionary cache by computing the hit ratio. The following example shows how you can compute this ratio:

```
SQL> SELECT SUM(gets - getmisses - usage - fixed)
  2         / SUM(gets)
  3  FROM v$rowcache;

SUM(GETS-GETMISSES-USAGE-FIXED)/SUM(GETS)
-----------------------------------------
                                .91279688
```

A value of 0.85 or higher is generally considered acceptable. Of course, you can always try to improve on any results that you get! The process for tuning the dictionary cache hit ratio is exactly the same as for tuning the library cache hit ratio. You have to adjust the size of the shared pool. Oracle determines on its own how much of the shared pool memory to use for the dictionary cache.

Tuning Rollback Segments

Two points are important when tuning rollback segments: detecting contention and reducing shrinkage. Contention occurs when there are too few rollback segments in your database for the amount of updates that are occurring. When contention occurs, time is lost while processes wait their turn. Shrinkage occurs when you have defined an *optimal* size for a rollback segment (using the `OPTIMAL` keyword in the storage clause), and then the rollback segment grows beyond that size and is forced to shrink back again.

Detecting rollback segment contention

You can detect rollback segment contention by querying the `V$ROLLSTAT`. You're interested in the number of times that a process was forced to wait in relation to the number of times it attempted to access a rollback segment. The following query will tell you that:

```
SELECT SUM(gets), SUM(waits), SUM(waits)/SUM(gets)
FROM v$rollstat;
```

Contention is indicated by waits. Any nonzero value for waits indicates that contention is occurring. If the sum of the waits ever becomes greater than about 1 percent of the total gets, then the contention is severe enough that you ought to do something about it. You can lower contention by creating more rollback segments.

Because the query shown here reflects all activity that has occurred since the instance has started, you may want to consider marking off a period of time and executing the query twice: once at the beginning of the period and once at the end. That will tell you if contention is occurring *now* (now being defined as during that period). If you have a database that's been up for two weeks and you execute the query once, you have no way of knowing whether the contention problem still exists or whether it was last week's problem.

Detecting shrinkage

When you create a rollback segment, you can define an initial size, a maximum size, and an optimal size. For example:

```
CREATE ROLLBACK SEGMENT rbs21
    TABLESPACE rbs
    STORAGE (INITIAL 10K OPTIMAL 20K
             NEXT 10K MAXEXTENTS 8);
```

This statement creates a rollback segment with an initial size of 20KB (rollback segments always have at least two extents allocated) and an optimal size of 20KB. The rollback segment can grow beyond 20KB if necessary, up to a maximum size of 80KB (8 extents). However, if the rollback segment grows beyond 20KB, Oracle will shrink it back to 20KB in size when that extra space is no longer needed.

Shrinking entails quite a bit of overhead, and you don't want your rollback segments constantly growing and shrinking. You can check the amount of shrinking that is occurring by looking at the V$ROLLSTAT view. Consider this example:

```
SQL> SELECT name, shrinks
  2  FROM v$rollstat, v$rollname
  3  WHERE v$rollstat.usn = v$rollname.usn;

NAME                            SHRINKS
------------------------------ ---------
SYSTEM                                0
RB1                                   0
RB2                                  25
```

If you're getting more than a small handful of shrinks per day, that's too much. You can reduce shrinkage by increasing the optimal size setting or by setting the optimal size to NULL. Setting the optimal size to NULL eliminates the problem of shrinkage entirely, because it tells Oracle never to shrink a rollback segment back to size at all. However, the NULL setting brings with it a potential new problem. If a segment grows to an extremely large size, it will stay that size forever. Ideally, you want the optimal size to be something that is exceeded only occasionally, maybe a few times per day.

Tuning the Redo Log Buffer

The redo log buffer is an area in memory to which processes write a record of the changes that they make to the data in the database. The log writer process eventually writes that record to the database redo log files. Changes must be recorded in the redo log before they can be committed. If user processes can't access the redo log buffer in a timely fashion, performance can suffer.

To tune the redo log buffer, you need to monitor for contention and take steps to reduce it when it is found. Multiple processes can write into the redo log buffer at the same time. When a process needs to write, it must first allocate some space in which to do that. If space is available, that's great. If no space is available because it is all being used by other processes, then you have contention.

The redo buffer allocation retries statistic tells you if contention for the redo log buffer is occurring. You can get the value for this statistic from V$SYSSTAT, as shown in this example:

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name = 'redo buffer allocation retries';

NAME                                                                  VALUE
---------------------------------- ---------
redo buffer allocation retries          39573
```

This statistic tells you the number of times a process had to wait for a slot to open up in the redo log buffer. To properly interpret this statistic, check its value more than once to see how much it is changing over time. Ideally, a process should never need to wait for a slot in the buffer, so if this number is growing consistently, consider increasing the redo log buffer size. You can do that by increasing the value of the LOG_BUFFER initialization parameter. Increasing the buffer size helps ensure that a space will be available when it is needed.

Summary

In this chapter, you learned:

- You can use two scripts, UTLBSTAT.SQL and UTLESTAT.SQL, to collect statistics telling you how efficiently your database is operating. These scripts allow you to measure those statistics over any arbitrary period of time that you desire.
- The GETHITRATIO and the PINHITRATIO are two key statistics that tell you how well the library cache is performing. The library cache contains recently executed SQL statements and PL/SQL program units. The greater the ratios, the more often Oracle is finding items in the cache as opposed to reading from disk.
- To improve the GETHITRATIO and PINHITRATIO, you can try increasing the size of the shared pool. Do that by increasing the value of the shared_pool initialization parameter.
- The buffer cache hit ratio tells you how often Oracle is able to avoid disk I/O because a needed data block was found in the database buffer cache. Buffer cache hit ratios of 90 percent or higher are generally considered to be good.
- If your buffer cache hit ratio is low, you can attempt to improve it by increasing the size of the cache. Increase the value for the db_block_buffers initialization parameter to do this.
- To detect contention for rollback segments, monitor the percentage of waits to gets found in the V$ROLLSTAT dynamic performance view. If the percentage of waits is high, perhaps higher than 1 percent of the gets, then consider adding more rollback segments.
- You should avoid excessive growth and shrinkage of rollback segments. If the number of shrinks reported by V$ROLLSTAT is high, then consider increasing the optimal size setting for the affected rollback segments.
- To check for redo log buffer contention, monitor the value of the redo buffer allocation retries statistic as reported by the V$SYSTAT view. If this value increases significantly over time, you are experiencing contention. You can reduce contention by increasing the size of the redo log buffer.

✦ ✦ ✦