Moving Baselines

Like the fixed baseline, the moving baseline is used to capture metrics over a period of time. The big difference is the metrics for moving baselines are captured based on the entire AWR retention period. For instance, the default AWR retention is eight days (see Recipe 4-2 on changing the AWR retention period). These metrics, also called adaptive thresholds, are captured based on the entire eight-day window. Furthermore, the baseline changes with each passing day, as the AWR window for a given database moves day by day. Because of this, the metrics over a given period of time can change as a database evolves and performance loads change over time. A default moving baseline is automatically created—the `SYSTEM_MOVING_BASELINE`. It is recommended to increase the default AWR retention period, as this may give a more complete set of metrics on which to accurately analyze performance. The maximum size of the moving window is the AWR retention period. To modify the moving window baseline, use the `MODIFY_BASELINE_WINDOW_SIZE` procedure of the `DBMS_WORKLOAD_REPOSITORY` package, as in the following example:

```
SQL>  exec dbms_workload_repository.modify_baseline_window_size(30);

PL/SQL procedure successfully completed.
```

How It Works

Setting the AWR retention period is probably the most important thing to configure when utilizing the moving baseline, as all adaptive threshold metrics are based on information from the entire retention period. When setting the retention period for the moving baseline, remember again that it cannot exceed the AWR retention period, else you may get the following error:

```
SQL> exec dbms_workload_repository.modify_baseline_window_size(45);
BEGIN dbms_workload_repository.modify_baseline_window_size(45); END;
*
ERROR at line 1:
ORA-13541: system moving window baseline size (3888000) greater than retention
(2592000)
ORA-06512: at "SYS.DBMS_WORKLOAD_REPOSITORY", line 686
ORA-06512: at line 1
```

If you set your AWR retention to an unlimited value, there still is an upper bound to the moving baseline retention period, and you could receive the following error if you set your moving baseline retention period too high, and your AWR retention period is set to unlimited:

```
exec dbms_workload_repository.modify_baseline_window_size(92);
BEGIN dbms_workload_repository.modify_baseline_window_size(92); END;
*
ERROR at line 1:
ORA-13539: invalid input for modify baseline window size (window_size, 92)
ORA-06512: at "SYS.DBMS_WORKLOAD_REPOSITORY", line 686
ORA-06512: at line 1
```

For fixed baselines, the AWR retention isn't a factor, and is a consideration only based on how far back in time you want to compare a snapshot to your baseline. After you have set up any baselines, you can get information on baselines from the data dictionary. To get information on the baselines in your database, you can use a query such as the following one, which would show you any fixed baselines you have configured, as well as the automatically configured moving baseline:

```
  SELECT baseline_name, start_snap_id start_id,
         TO_CHAR(start_snap_time, 'yyyy-mm-dd:hh24:mi') start_time,
         end_snap_id end_id,
         TO_CHAR(end_snap_time, 'yyyy-mm-dd:hh24:mi') end_time,
         expiration
  FROM dba_hist_baseline
  ORDER BY baseline_id;

BASELINE_NAME          START_ID START_TIME           END_ID END_TIME         EXPIRATION
-------------------- ---------- ---------------- ---------- ---------------- ----------
SYSTEM_MOVING_WINDOW        255 2011-05-27:22:00        358 2011-06-08:22:00
Batch Baseline #1           258 2011-05-28:13:39        268 2011-05-29:00:00         30
```

From the foregoing results, the moving baseline includes the entire range of snapshots based on the AWR retention period; therefore the expiration is shown as NULL. You can get similar information by using the SELECT_BASELINE_DETAILS function of the DBMS_WORKLOAD_REPOSITORY package. You do need the baseline_id number to pass into the function to get the desired results:

```
SELECT start_snap_id, start_snap_time, end_snap_id, end_snap_time,
       pct_total_time pct  FROM (SELECT * FROM
       TABLE(DBMS_WORKLOAD_REPOSITORY.select_baseline_details(12)));

START_SNAP_ID START_SNAP_TIME           END_SNAP_ID END_SNAP_TIME                  PCT
------------- ------------------------ ----------- ------------------------ ----------
          258 28-MAY-11 01.39.19.296 PM         268 29-MAY-11 12.00.45.211 AM     100
```

To get more specific information on the moving baseline in the database, you are drilling down into the statistics for the adaptive metrics. For instance, to see an average and maximum for each metric related to *reads* based on the moving window, you could use the following query:

```
column metric_name format a50
SELECT metric_name, average, maximum FROM
(SELECT * FROM TABLE
(DBMS_WORKLOAD_REPOSITORY.select_baseline_metric('SYSTEM_MOVING_WINDOW')))
where lower(metric_name) like '%read%'
order by metric_name;

METRIC_NAME                                          AVERAGE    MAXIMUM
-------------------------------------------------- ---------- ----------
Average Synchronous Single-Block Read Latency      .159658155 53.8876404
Consistent Read Changes Per Sec                    2.99232446 3984.11246
Consistent Read Changes Per Txn                    117.812978     239485
Consistent Read Gets Per Sec                       202.570936  64677.436
Consistent Read Gets Per Txn                       3930.41373 372602.889
Logical Reads Per Sec                              224.984307 64690.6884
Logical Reads Per Txn                              4512.34119     840030
Logical Reads Per User Call                        276.745756     135804
Physical Read Bytes Per Sec                        1249601.48  528672777
Physical Read IO Requests Per Sec                  6.44664078 2040.73828
```

```
Physical Read Total Bytes Per Sec                1272159.18  528699475
Physical Read Total IO Requests Per Sec          7.82238122 2042.31792
Physical Reads Direct Lobs Per Sec               .006030572  4.6953047
Physical Reads Direct Lobs Per Txn               .231642268         141
Physical Reads Direct Per Sec                    59.3280451 64535.1513
Physical Reads Direct Per Txn                    602.336945 371825.222
Physical Reads Per Sec                           152.539244 64535.2511
Physical Reads Per Txn                           2966.04803 371831.889
```

4-7. Managing AWR Baselines via Enterprise Manager

Problem

You want to create and manage AWR baselines using Enterprise Manager.

Solution

Using Enterprise Manager, you can easily configure or modify baselines. In Figure 4-3, you can see the window where you can establish or modify your existing baselines, including any fixed baselines, as well as the system moving baseline. To create a new fixed baseline, you would click the Create button, which would navigate you to the screen shown in Figure 4-4, where you can configure your new fixed baseline. Within this screen, you name your baseline, and choose between a snapshot-based or time-based baseline.

Figure 4-3. *Managing baselines within Enterprise Manager*

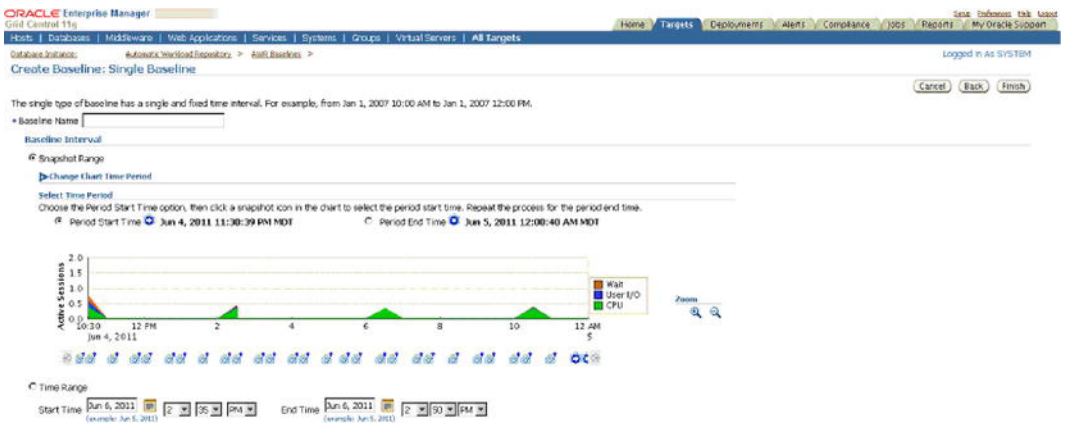

Figure 4-4. *Creating new fixed baseline within Enterprise Manager*

When deciding to modify your existing baselines, the screen options differ between modifying fixed baselines, and modifying the system moving baseline. Figure 4-5 shows the modifiable options for a fixed baseline. As you can see, the only real modification that can be made is the actual baseline name itself. Figure 4-6 shows how to change the moving baseline window within Enterprise Manager. As mentioned before, the actual screens may differ between versions of the Enterprise Manager tool.

Figure 4-5. *Modifying a fixed baseline within Enterprise Manager*

Figure 4-6. *Modifying the moving baseline within Enterprise Manager*

How It Works

For any fixed baseline created or the system moving baseline, you can also simply generate an AWR report based on a particular baseline. Figure 4-1 shows how to generate a snapshot-based AWR report by clicking the By Snapshot button. Using this same screen, you can also generate an AWR report for a baseline simply by clicking the By Baseline button.

If you want to delete a baseline from within Enterprise Manager, simply click the radio button of the baseline you wish to delete, and then click the Delete button, as depicted in Figure 4-7. Figure 4-8 shows how to actually delete the baseline. You can choose to keep or purge the baseline data by clicking the appropriate radio button.

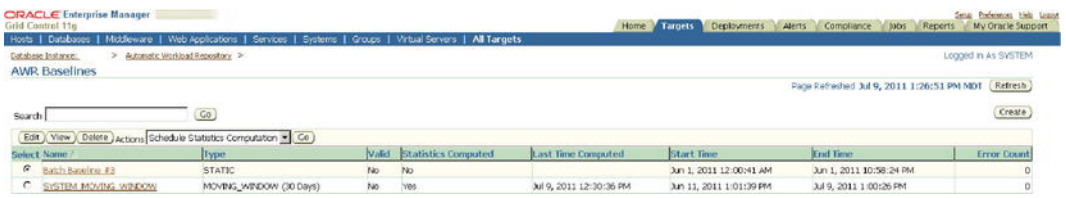

Figure 4-7. *Choosing a baseline to delete*

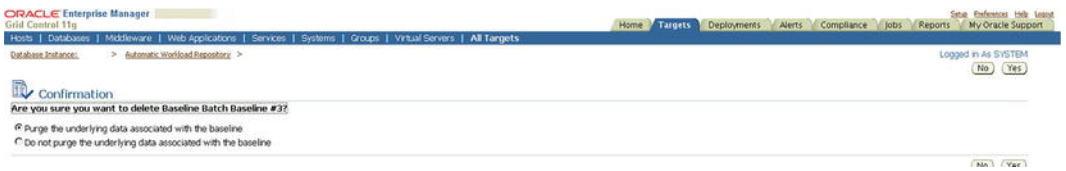

Figure 4-8. *Deleting a baseline*

■ **Note** You cannot delete the system moving baseline.

4-8. Managing AWR Statistics Repository

Problem

You have AWR snapshots and baselines in place for your database, and need to perform regular maintenance activities for your AWR information.

Solution

By using the DBMS_WORKLOAD_REPOSITORY package, you can perform most maintenance on your baselines, including the following:

- Renaming a baseline
- Dropping a baseline
- Dropping a snapshot range

To rename a baseline, use the RENAME_BASELINE procedure of the DBMS_WORKLOAD_REPOSITORY package:

```
SQL>  exec dbms_workload_repository.rename_baseline -
      ('Batch Baseline #9','Batch Baseline #10');

PL/SQL procedure successfully completed.
```

To drop a baseline, simply use the DROP_BASELINE procedure:

```
SQL> exec dbms_workload_repository.drop_baseline('Batch Baseline #1');

PL/SQL procedure successfully completed.
```

If you have decided you have AWR snapshots you no longer need, you can reduce the number of AWR snapshots held within your database by dropping a range of snapshots using the DROP_BASELINE procedure:

```
SQL>  exec dbms_workload_repository.drop_snapshot_range(255,256);

PL/SQL procedure successfully completed.
```

How It Works

In addition to the DBMS_WORKLOAD_REPOSITORY package, there are other things you can do to analyze your AWR information in order to help manage all of the AWR information in your database, including the following:

- Viewing AWR information from the data dictionary
- Moving AWR information to a repository in another database location

If you wanted to store AWR information for an entire grid of databases in a single database, Oracle provides scripts that can be run in order to extract AWR information from a given database, based on a snapshot range, and in turn, load that information into a different database.

To extract a given snapshot range of AWR information from your database, you need to run the `awrextr.sql` script. This script is an interactive script, and asks for the same kind of information as when you generate an AWR report using the `awrrpt.sql` script. You need to answer the following questions when running this script:

1. DBID (defaults to DBID of current database)
2. Number of days of snapshots to display for extraction
3. The beginning snapshot number to extract
4. The ending snapshot number to extract
5. Oracle directory in which to place the resulting output file holding all the AWR information for the specified snapshot range; the directory must be entered in upper case.
6. Output file name (defaults to `awrdat` plus snapshot range numbers)

Keep in mind that the output file generated by this process does take up space, which can vary based on the number of sessions active at snapshot time. Each individual snapshot needed for extraction can take up 1 MB or more of storage, so carefully gauge the amount of snapshots needed. If necessary, you can break the extraction process into pieces if there is inadequate space on a given target directory.

In addition, for each output file generated, a small output log file is also generated, with information about the extraction process, which can be useful in determining if the information extracted matches what you think has been extracted. This is a valuable audit to ensure you have extracted the AWR information you need.

Once you have the extract output file(s), you need to transport them (if necessary) to the target server location for loading into the target database location. The load process is done using the `awrload.sql` script. What is needed for input into the load script includes the following:

1. Oracle directory in which to place the resulting output file holding all the AWR information for the specified snapshot range; the directory must be entered in upper case.
2. File name (would be the same name as entered in number 6 of the extraction process (for the `awrextr.sql` script); when entering the file name, exclude the `.dmp` suffix, as it will be appended automatically.
3. Target schema (default schema name is `AWR_STAGE`)
4. Target tablespace for object that will be created (provides list of choices)
5. Target temporary tablespace for object that will be created (provides list of choices)

After the load of the data is complete, the AWR data is moved to the SYS schema in the data dictionary tables within the target database. The temporary target schema (for example, AWR_STAGE) is then dropped.

In order to generate an AWR report generated from one database that is then loaded into a different database, use the AWR_REPORT_TEXT function of the DBMS_WORKLOAD_REPOSITORY package. For example, let's say we loaded and stored snapshots 300 through 366 into our separate AWR database. If we wanted to generate an AWR report for the information generated between snapshots 365 and 366 for a given database, we would run the following command, with the DBID of the originating, source database, as well as the beginning and ending snapshot numbers as follows:

```
SELECT  dbms_workload_repository.awr_report_text
        (l_dbid=>2334201269,l_inst_num=>1,l_bid=>365,l_eid=>366)
FROM dual;
```

4-9. Creating AWR Baselines Automatically

Problem

You want to periodically create baselines in your database automatically.

Solution

You can create an AWR repeating template, which gives you the ability to have baselines created automatically based on a predefined interval and time frame. By using the CREATE_BASELINE_TEMPLATE procedure within the DBMS_WORKLOAD_REPOSITORY package, you can have a fixed baseline automatically created for this repeating interval and time frame. See the following example to set up an AWR template:

```
SQL> alter session set nls_date_format = 'yyyy-mm-dd:hh24:mi:ss';

SQL> exec DBMS_WORKLOAD_REPOSITORY.create_baseline_template( -
>    day_of_week          => 'WEDNESDAY', -
>    hour_in_day          => 0, -
>    duration             => 6, -
>    start_time           => '2011-06-14:00:00:00', -
>    end_time             => '2011-06-14:06:00:00', -
>    baseline_name_prefix => 'Batch Baseline ', -
>    template_name        => 'Batch Template', -
>    expiration           => 365);

PL/SQL procedure successfully completed.
```

For the foregoing template, a fixed baseline will be created based on the midnight to 6 a.m. window every Wednesday. In this case, this template creates baselines for a normal batch window time frame.

If you are using Enterprise Manager, you can create a template using the same parameters. See Figure 4-9 for an example.

Figure 4-9. *Creating an AWR template*

How It Works

If you need to drop your template, you simply use the `DROP_BASELINE_TEMPLATE` procedure from the `DBMS_WORKLOAD_REPOSITORY` package. See the following example:

```
SQL> exec dbms_workload_repository.drop_baseline_template('Batch Template');

PL/SQL procedure successfully completed.
```

If you wish to view information on any templates you have created, you can query the `DBA_HIST_BASELINE_TEMPLATE` view. See the following sample query:

```
column template_name format a14
column prefix format a14
column hr format 99
column dur format 999
column exp format 999

SELECT template_name, baseline_name_prefix prefix,
to_char(start_time,'mm/dd/yy:hh24') start_time,
to_char(end_time,'mm/dd/yy:hh24') end_time,
substr(day_of_week,1,3) day, hour_in_day hr, duration dur, expiration exp,
to_char(last_generated,'mm/dd/yy:hh24') last
FROM dba_hist_baseline_template;


TEMPLATE_NAME  PREFIX         START_TIME  END_TIME    DAY  HR  DUR  EXP LAST
-------------- -------------- ----------- ----------- --- --- ---- ---- -----------
Batch Template Batch Baseline 06/14/11:00 06/14/11:06 WED   0    6  365 06/14/11:00
```

4-10. Quickly Analyzing AWR Output

Problem

You have generated an AWR report, and want to quickly interpret key portions of the report to determine if there are performance issues for your database.

Solution

The AWR report, like its predecessors Statspack and UTLBSTAT/UTLESTAT for earlier versions of Oracle, has a multitude of statistics to help you determine how your database is functioning and performing. There are many sections of the report. The first three places on the report to gauge how your database is performing are as follows:

1. DB Time
2. Instance Efficiency
3. Top 5 Timed Events

The first section displayed on the report shows a summary of the snapshot window for your report, as well as a brief look at the elapsed time, which represents the snapshot window, and the DB time, which represents activity on your database. If the DB time exceeds the elapsed time, it denotes a busy database. If it is a lot higher than the elapsed time, it may mean that some sessions are waiting for resources. While not specific, it can give you a quick view to see if your overall database is busy and possibly overtaxed. We can see from the following example of this section that this is a very busy database by comparing the elapsed time to the DB time:

```
              Snap Id      Snap Time      Sessions Curs/Sess
            --------- ------------------- -------- ---------
Begin Snap:     18033 11-Jun-11 00:00:43        59       2.3
  End Snap:     18039 11-Jun-11 06:00:22        69       2.4
   Elapsed:               359.66 (mins)
   DB Time:             7,713.90 (mins)
```

The instance efficiency section gives you a very quick view to determine if things are running adequately on your database. Generally, most percentages within this section should be above 90%. The Parse CPU to Parse Elapsd metric shows how much time the CPU is spending parsing SQL statements. The lower this metric is, the better. In the following example, it is about 2%, which is very low. If this metric ever gets to 5%, it may mean investigation is warranted to determine why the CPU is spending this much time simply parsing SQL statements.

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Buffer Nowait %:   99.64       Redo NoWait %:   99.99
            Buffer  Hit   %:   91.88    In-memory Sort %:   99.87
            Library Hit   %:   98.92        Soft Parse %:   94.30
         Execute to Parse %:   93.70         Latch Hit %:   99.89
Parse CPU to Parse Elapsd %:    2.10     % Non-Parse CPU:   99.75
```

The third place to get a quick glance at your database performance is the Top 5 Timed Events section. This section gives you a quick look at exactly where the highest amount of resources are being consumed within your database for the snapshot period. Based on these results, it may show you that there is an inordinate amount of time spent performing full-table scans, or getting data across a network database link. The following example shows that the highest amount of resources is being used performing index scans (noted by "`db file sequential read`"). We can see there is significant time on "`local write wait`", "`enq: CF – contention`", and "`free buffer waits`", which gives us a quick view of what possible contention and wait events are for our database, and gives us immediate direction for investigation and analysis.

```
Top 5 Timed Foreground Events                                  Avg
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                              wait   % DB
Event                                 Waits    Time (s)   (ms)   Time Wait Class
------------------------------ ------------ ----------- ------ ------ ----------
db file sequential read          3,653,606      96,468     26   20.8   User I/O
local write wait                    94,358      67,996    721   14.7   User I/O
enq: CF - contention                18,621      46,944   2521   10.1      Other
free buffer waits                3,627,548      38,249     11    8.3 Configurat
db file scattered read           2,677,267      32,400     12    7.0   User I/O
```

How It Works

After looking at the DB Time, Instance Efficiency, and Top 5 Timed Events sections, if you want to look in more detail at the sections of a given AWR report, refer to Recipe 7-17 in Chapter 7 for more information. Because the sheer volume of information in the AWR report is so daunting, it is strongly recommended to create baselines that represent a normal processing window. Then, AWR snapshots can be compared to the baselines, and metrics that may just look like a number on a given AWR report will stand out when a particular metric is significantly above or below a normal range.

4-11. Manually Getting Active Session Information

Problem

You need to do performance analysis on sessions that run too frequently or are too short to be available on available AWR snapshots. The AWR snapshots are not taken often enough to capture the information that you need.

Solution

You can use the Oracle Active Session History (ASH) information in order to get real-time or near real-time session information. While the AWR information is very useful, it is bound by the reporting periods, which are by default run every hour on your database. The ASH information has active session information, and is sampled every second from `V$SESSION`, and can show more real-time or near real-time session information to assist in doing performance analysis on your database. There are a few ways to get active session information from the database:

- Running the canned ASH report
- Running an ASH report from within Enterprise Manager (see Recipe 4-12)
- Getting ASH information from the data dictionary (see Recipe 4-13)

The easiest method to get information on active sessions is to run the `ashrpt.sql` script, which is similar in nature to the `awrrpt.sql` script that is run when generating an AWR report. When you run the `ashrpt.sql` script, it asks you for the following:

- Report type (text or HTML)
- Begin time for report (defaults to current time minus 15 minutes)
- End time for report (defaults to current time)
- Report name

There are many sections to the ASH report. See Table 4-1 for a brief description of each section. See the following snippet from many of the sections of the ASH report. Some sections have been shortened for brevity.

```
Top User Events            DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

                                                          Avg Active
Event                             Event Class     % Activity   Sessions
---------------------------------- --------------- ---------- ----------
CPU + Wait for CPU                CPU                  35.36       1.66
db file scattered read            User I/O             33.07       1.55
db file sequential read           User I/O             21.33       1.00
read by other session             User I/O              6.20       0.29
direct path read temp             User I/O              2.59       0.12
          -------------------------------------------------------------

Top Background Events      DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

                                                          Avg Active
Event                             Event Class     % Activity   Sessions
---------------------------------- --------------- ---------- ----------
Log archive I/O                   System I/O           12.77       0.68
CPU + Wait for CPU                CPU                   6.38       0.34
log file parallel write           System I/O            5.66       0.30
log file sequential read          System I/O            4.91       0.26
log file sync                     Commit                1.06       0.06
          -------------------------------------------------------------
```

Top Event P1/P2/P3 Values DB/Inst: ORCL/ORCL (Jun 18 12:00 to 12:45)

```
Event                          % Event  P1 Value, P2 Value, P3 Value % Activity
------------------------------ ------- ---------------------------- ----------
Parameter 1                Parameter 2                Parameter 3
-------------------------- -------------------------- --------------------------
db file scattered read           17.30           "775","246084","16"       0.14
file#                      block#                     blocks

Datapump dump file I/O            6.32         "1","32","2147483647"       6.32
count                      intr                       timeout

RMAN backup & recovery I/O        5.83         "1","32","2147483647"       5.80
count                      intr                       timeout
```

Top Service/Module DB/Inst: ORCL/ORCL (Jun 18 12:00 to 12:45)

```
Service        Module                   % Activity Action               % Action
-------------- ------------------------ ---------- ------------------ ----------
SYS$BACKGROUND UNNAMED                       31.00 UNNAMED                 31.00
               DBMS_SCHEDULER                18.87 GATHER_STATS_JOB        18.87
               Data Pump Worker              18.87 APP_IMPORT              18.87
SYS$BACKGROUND MMON_SLAVE                     1.95 Auto-Flush Slave A       1.42
          -------------------------------------------------------------
```

Top SQL Command Types DB/Inst: ORCL/ORCL (Jun 18 12:00 to 12:45)

```
                                        Distinct            Avg Active
SQL Command Type                          SQLIDs % Activity   Sessions
----------------------------------------- ---------- ---------- ----------
INSERT                                             2      18.88       1.00
SELECT                                            27       2.36       0.12
          -------------------------------------------------------------
```

Top SQL Statements DB/Inst: ORCL/ORCL (Jun 18 12:00 to 12:45)

```
       SQL ID    Planhash % Activity Event                          % Event
------------- ----------- ---------- ------------------------------ ----------
av2f2stsjfr5k  3774074286       1.16 CPU + Wait for CPU                   0.80
 select a.tablespace_name, round(sum_free/sum_bytes,2)*100 pct_free from
(select tablespace_name, sum(bytes) sum_bytes from sys.dba_data_files group by t
ablespace_name) a, (select tablespace_name, sum(bytes) sum_free , max(bytes)
 bigchunk from sys.dba_free_space group by tablespace_name) b where a.table
```

```
Top Sessions              DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

   Sid, Serial# % Activity Event                              % Event
--------------- ---------- ------------------------------ ----------
User                 Program                          # Samples Active     XIDs
-------------------- ------------------------------ ------------------ --------
      365, 3613      18.87 CPU + Wait for CPU                    12.29
D_USER               oracle@oraprod (DW01)          1,755/2,700 [ 65%]        8

                           Datapump dump file I/O                 6.32
                                                    903/2,700 [ 33%]        8

      515, 8721      18.87 db file scattered read                17.26
SYS                  oracle@oraprod (J000)          2,465/2,700 [ 91%]        1

Top Blocking Sessions     DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

   Blocking Sid % Activity Event Caused                       % Event
--------------- ---------- ------------------------------ ----------
User                 Program                          # Samples Active     XIDs
-------------------- ------------------------------ ------------------ --------
      549,    1       2.09 enq: CF - contention                   2.03
SYS                  oracle@oraprod (CKPT)            248/2,700 [  9%]        0

Top DB Objects            DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

      Object ID % Activity Event                              % Event
--------------- ---------- ------------------------------ ----------
Object Name (Type)                                     Tablespace
------------------------------------------------------ -------------------------
        1837336       3.25 db file scattered read                 3.25
STG.EMPPART.EMPPART10_11P (TAB EMP_S

        1837324       3.05 db file scattered read                 3.05
STG.EMPPART.EMPPART10_10P (TAB EMP_S

Top DB Files              DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

        File ID % Activity Event                              % Event
--------------- ---------- ------------------------------ ----------
File Name                                              Tablespace
------------------------------------------------------ -------------------------
            200       6.31 Datapump dump file I/O                 6.31
/opt/vol01/ORCL/app_s_016.dbf                           APP_S

          ------------------------------------------------------------
```

```
Activity Over Time       DB/Inst: ORCL/ORCL  (Jun 18 12:00 to 12:45)

                          Slot                                    Event
Slot Time (Duration)     Count Event                             Count % Event
-------------------- -------- ------------------------------ -------- -------
12:00:00   (5.0 min)    2,672 CPU + Wait for CPU                1,789   12.52
                              db file scattered read              290    2.03
                              enq: CF - contention                290    2.03
12:05:00   (5.0 min)    2,586 CPU + Wait for CPU                1,396    9.77
                              RMAN backup & recovery I/O          305    2.14
                              db file scattered read              287    2.01
12:10:00   (5.0 min)    2,392 CPU + Wait for CPU                1,068    7.48
                              Log archive I/O                     423    2.96
                              RMAN backup & recovery I/O          356    2.49
...
          -------------------------------------------------------------
```

Table 4-1. *ASH Report Section Information for the Specified Report Period*

Section Name	Description
General Report Information	Contains database name, reporting period, CPU and memory information
Top User Events	Displays the top run user events for the reporting period
Top Background Events	Shows the top wait events in the database
Top P1/P2/P3 Events	Lists top wait event parameter values based on highest percentages, ordered in descending order
Top Service Module	Displays the top services or module names
Top Client IDs	Shows the top users
Top SQL Command Types	Shows all the SQL commands run
Top SQL Statements	Displays the top consuming SQL statement text
Top SQL Using Literals	Shows SQL statements using literals; this can assist in determining offending SQL for shared pool contention.
Top PL/SQL Procedures	Displays the PL/SQL programs run
Top Sessions	Displays the top sessions within the database

Section Name	Description
Top Blocking Sessions	Sessions that are blocking other sessions
Top Sessions Running PQs	Sessions running parallel query processes
Top DB Objects	Objects referenced
Top DB Files	Files referenced
Top Latches	Latch information for the reporting period
Activity Over Time	Shows top three consuming events for each five-minute reporting period shown on report

How It Works

Retrieving ASH information is necessary if you need to get session information more current than you can retrieve from the AWR report. Again, AWR information is generated only hourly by default. ASH information is gathered every second from `V$SESSION`, and stores the most useful session information to help gauge database performance at any given moment.

The ASH information is stored within a circular buffer in the SGA. Oracle documentation states that the buffer size is calculated as follows:

```
Max [Min [ #CPUs * 2 MB, 5% of Shared Pool Size, 30MB ], 1MB ]
```

The amount of time that the information is stored within the data dictionary depends on the activity within your database. You may need to view the `DBA_HIST_ACTIVE_SESS_HISTORY` historical view in order to get the ASH information you need if your database is very active. For an example of querying the `DBA_HIST_ACTIVE_SESS_HISTORY` view, see Recipe 4-13. To quickly see how much data is held in your historical view, you could simply get the earliest `SAMPLE_TIME` from the `DBA_HIST_ACTIVE_SESS_HISTORY` view:

```
SELECT min(sample_time) FROM dba_hist_active_sess_history;

MIN(SAMPLE_TIME)
---------------------------------------------------------------------------
20-MAR-11 11.00.27.433 PM
```

The MMON background process, which manages the AWR hourly snapshots, also flushes ASH information to the historical view at the same time. If there is heavy activity on the database, and the buffer fills between the hourly AWR snapshots, the MMNL background process will wake up and flush the ASH data to the historical view.

The `V$ACTIVE_SESSION_HISTORY` and `DBA_HIST_ACTIVE_SESS_HISTORY` views contain much more detailed information than just the samples shown within this recipe, and you can drill down and get much more information at the session level, if desired, including information regarding actual SQL statements, the SQL operations, blocking session information, and file I/O information.

4-12. Getting ASH Information from Enterprise Manager

Problem

You want to get to ASH information from within Enterprise Manager because you use Enterprise Manager for performance tuning activities.

Solution

The ASH report generated from within Enterprise Manager has the same sections as specified in Table 4-1 (see Recipe 4-11). To generate an ASH report from within Enterprise Manager, you generally need to be in the Performance tab, depending on your particular version of Enterprise Manager. As with running the `ashrpt.sql` script, you need to specify the beginning and ending time frames for the report period desired. See Figure 4-10 for an example of the screen used to generate an ASH report, and Figure 4-11 for a sample of the ASH report output:

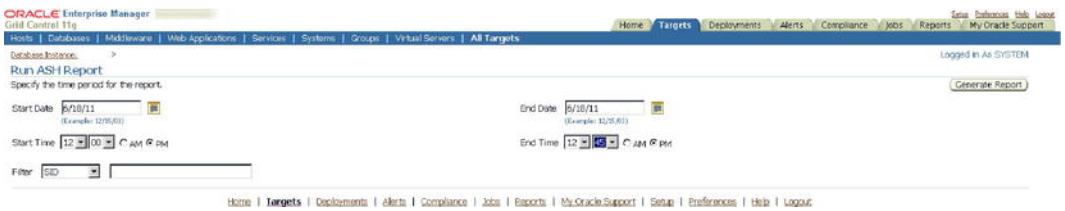

Figure 4-10. *Generating ASH report from Enterprise Manager*

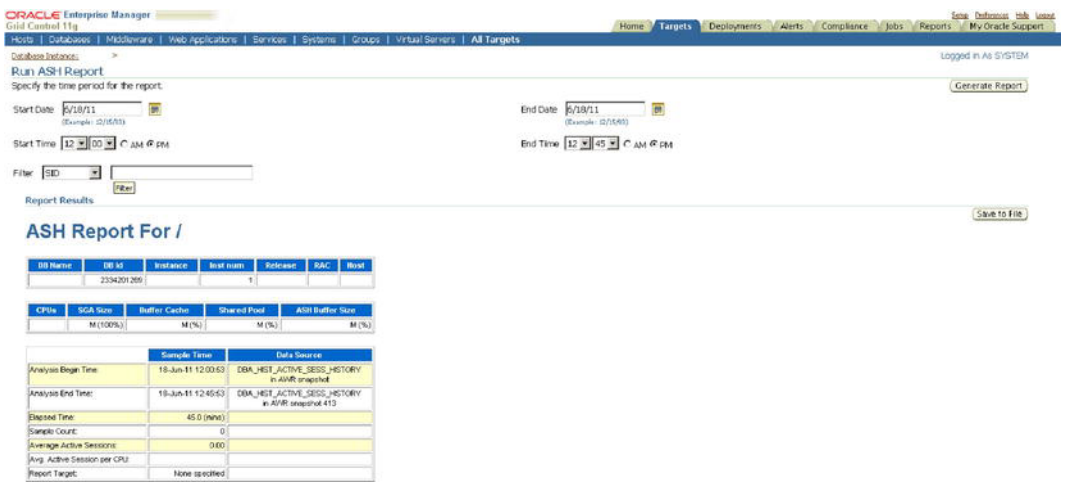

Figure 4-11. *Sample ASH report from Enterprise Manager*

How It Works

When generating an ASH report, you have the option to filter on specific criteria. In Figure 4-12, see the Filter drop-down menu. If you have a very active database, and already want to zero in on a specific SQL_ID, for example, you can choose the SQL_ID option from the Filter drop-down menu, and enter the SQL_ID value. The resulting report will show information based only on the filtered criteria.

The choices to filter on include the following:

- SID
- SQL_ID
- Wait Class
- Service
- Module
- Action
- Client

Many of the foregoing filters can be found in the `V$SESSION` view. For a list of the possible wait classes, you can query the `DBA_HIST_EVENT_NAME` view as shown in the following example:

```
SELECT DISTINCT wait_class FROM dba_hist_event_name;

WAIT_CLASS
----------------------------------------------------------------
Concurrency
User I/O
Administrative
System I/O
Scheduler
Configuration
Other
Application
Cluster
Network
Idle
Commit
```

Figure 4-12. *Customizing ASH report by filter*

4-13. Getting ASH Information from the Data Dictionary

Problem

You want to see what ASH information is kept in Oracle's data dictionary.

Solution

There are a couple of data dictionary views you can use to get ASH information. The first, `V$ACTIVE_SESSION_HISTORY`, can be used to get information on current or recent sessions within your database. The second, `DBA_HIST_ACTIVE_SESS_HISTORY`, is used to store older, historical ASH information.

If you wanted to see all the events and their total wait time for activity within the past 15 minutes in your database, you could issue the following query:

```
SELECT s.event, sum(s.wait_time + s.time_waited) total_wait
FROM v$active_session_history s
WHERE s.sample_time between sysdate-1/24/4 AND sysdate
GROUP BY s.event
ORDER BY 2 desc;

EVENT                                                            TOTAL_WAIT
---------------------------------------------------------------- ----------
                                                                   20002600
db file scattered read                                             15649078
read by other session                                               9859503
db file sequential read                                              443298
direct path read temp                                                156463
direct path write temp                                               139984
log file parallel write                                               49469
db file parallel write                                                21207
log file sync                                                         11793
```

```
SGA: allocation forcing component growth                          11711
control file parallel write                                        4421
control file sequential read                                       2122
SQL*Net more data from client                                       395
SQL*Net more data to client                                          66
```

If you wanted to get more session-specific information, and wanted to see the top 5 sessions that were using the most CPU resources within the last 15 minutes, you could issue the following query:

```
column username format a12
column module format a30

SELECT * FROM
(
SELECT s.username, s.module, s.sid, s.serial#, count(*)
FROM v$active_session_history h, v$session s
WHERE h.session_id = s.sid
AND   h.session_serial# = s.serial#
AND   session_state= 'ON CPU' AND
      sample_time > sysdate - interval '15' minute
GROUP BY s.username, s.module, s.sid, s.serial#
ORDER BY count(*) desc
)
where rownum <= 5;

USERNAME   MODULE                          SID    SERIAL#   COUNT(*)
---------- ------------------------------ ---------- ---------- ----------
SYS        DBMS_SCHEDULER                  536          9         43
APPLOAD    etl1@app1 (TNS V1-V3)          1074       3588         16
APPLOAD    etl1@app1 (TNS V1-V3)          1001       4004         12
APPLOAD    etl1@app1 (TNS V1-V3)           968        108          5
DBSNMP     emagent@ora1 (TNS V1-V3)        524          3          2
```

The SESSION_STATE column has two valid values, ON CPU and WAITING, which denote whether a session is active or is waiting for resources. If you wanted to see the sessions that are waiting for resources, you could issue the same query as previously, with a SESSION_STATE of WAITING.

If you wanted to see the most heavily used database objects for a given sample period, you could join V$ACTIVE_SESSION_HISTORY to the DBA_OBJECTS view to get that information. In the following example, we are getting a list of the top 5 database objects in use, along with the event associated with that database object, over the past 15 minutes:

```
SELECT * FROM
(
SELECT o.object_name, o.object_type, s.event,
       SUM(s.wait_time + s.time_waited) total_waited
FROM v$active_session_history s, dba_objects o
WHERE s.sample_time between sysdate - 1/24/4 and sysdate
AND s.current_obj# = o.object_id
GROUP BY o.object_name, o.object_type, s.event
ORDER BY 4 desc
)
```

```
WHERE rownum <= 5;

OBJECT_NAME                    OBJECT_TYPE     EVENT                      TOTAL_WAITED
------------------------------ --------------- -------------------------- ------------
WRI$_ALERT_OUTSTANDING         TABLE           Streams AQ: enqueue block    110070196
                                               ed on low memory
APP_ETL_IDX1                   INDEX           read by other session         65248777
APP_SOURCE_INFO                TABLE PARTITION db file scattered read        33801035
EMPPART_PK_I                   INDEX PARTITION read by other session         28077262
APP_ORDSTAT                    TABLE PARTITION db file scattered read        15569867
```

How It Works

The DBA_HIST_ACTIVE_SESS_HISTORY view can give you historical information on sessions that have aged out of the V$ACTIVE_SESSION_HISTORY view. Let's say you had a day when performance was particularly bad on your database. You could zero in on historical session information for a given time frame, provided it is still held within the DBA_HIST_ACTIVE_SESS_HISTORY view. For instance, if you wanted to get the users that were consuming the most resources for a given day when performance was poor, you could issue the following query:

```
SELECT * FROM
(
SELECT u.username, h.module, h.session_id sid,
       h.session_serial# serial#, count(*)
FROM dba_hist_active_sess_history h, dba_users u
WHERE h.user_id = u.user_id
AND   session_state= 'ON CPU'
AND  (sample_time between to_date('2011-05-15:00:00:00','yyyy-mm-dd:hh24:mi:ss')
AND   to_date('2011-05-15:23:59:59','yyyy-mm-dd:hh24:mi:ss'))
AND u.username != 'SYS'
GROUP BY u.username, h.module, h.session_id, h.session_serial#
ORDER BY count(*) desc
)
where rownum <= 5;

USERNAME     MODULE                              SID    SERIAL#   COUNT(*)
------------ ------------------------------ ---------- ---------- ----------
APPLOAD1     etl1@app1 (TNS V1-V3)              1047        317       1105
APPLOAD1     etl1@app1 (TNS V1-V3)              1054        468        659
APPLOAD1     etl1@app1 (TNS V1-V3)              1000        909        387
STG          oracle@ora1 (TNS V1-V3)             962       1707        353
APPLOAD1     etl1@app1 (TNS V1-V3)               837      64412        328
```

To then zero in on the database objects, you could issue the following query for the same time frame:

```
SELECT * FROM
(
SELECT o.object_name, o.object_type, s.event,
       SUM(s.wait_time + s.time_waited) total_waited
FROM dba_hist_active_sess_history s, dba_objects o
```

```
WHERE s.sample_time
between to_date('2011-05-15:00:00:00','yyyy-mm-dd:hh24:mi:ss')
AND   to_date('2011-05-15:23:59:59','yyyy-mm-dd:hh24:mi:ss')
AND s.current_obj# = o.object_id
GROUP BY o.object_name, o.object_type, s.event
ORDER BY 4 desc
)
WHERE rownum <= 5;


OBJECT_NAME                 OBJECT_TYPE     EVENT                     TOTAL_WAITED
--------------------------- --------------- ------------------------- ------------
EMPPART                     TABLE PARTITION PX Deq Credit: send blkd    8196703427
APPLOAD_PROCESS_STATUS      TABLE           db file scattered read       628675085
APPLOAD_PROCESS_STATUS      TABLE           read by other session        408577335
APP_SOURCE_INFO             TABLE PARTITION db file scattered read       288479849
APP_QUALITY_INFO            TABLE PARTITION Datapump dump file I/O       192290534
```

CHAPTER 5

Minimizing System Contention

It's not uncommon for Oracle DBAs to field calls about a user being locked or "blocked" in the database. Oracle's locking behavior is extremely sophisticated and supports simultaneous use of the database by multiple users. However, on occasion, it's possible for a user to block another user's work, mostly due to flaws in application design. This chapter explains how Oracle handles locks and how to identify a session that's blocking others.

Oracle database can experience two main types of contention for resources. The first is contention for transaction locks on a table's rows. The second type of contention is that caused by simultaneous requests for areas of the shared memory (SGA), resulting in latch contention. In addition to showing you how to troubleshoot typical locking issues, we will also show how to handle various types of latch contention in your database.

Oracle Wait Interface is a handy name for Oracle's internal mechanism for classifying and measuring the different types of waits for resources in an Oracle instance. Understanding Oracle wait events is *the* key to instance tuning, because high waits slow down response time. We will explain the Oracle Wait Interface in this chapter and show you how to reduce the most common Oracle wait events that beguile Oracle DBAs. We will show you how to use various SQL scripts to unravel the mysteries of the Oracle Wait Interface, and we will also show how to use Oracle Enterprise Manager to quickly track down the SQL statements and sessions that are responsible for contention in the database.

5-1. Understanding Response Time

Problem

You want to understand what database response time is, and its relationship with wait time.

Solution

The most crucial performance indicator in a database is response time. Response time is the time it takes to get a response from the database for a query that a client sends to the database. Response time is simply the sum of two components:

```
response time = processing time + wait time
```

The foregoing relationship is also frequently represented as R=S + W, where R is the response time, S the service time, and W stands for the wait time. The processing time component is the actual time spent by the database processing the request. Wait time, on the other hand, is time actually wasted by the database—it's the time the database spends waiting for resources such as a lock on a table's rows, library cache latch, or any of the numerous resources that a query needs to complete its processing. Oracle has hundreds of official wait events, a dozen or so of which are crucial to troubleshooting slow-running queries.

Do You Have a Wait Problem?

It's easy to find out the percentage of time a database has spent waiting for resources instead of actually executing. Issue the following query to find out the relative percentages of wait times and actual CPU processing in the database:

```
SQL> select metric_name, value
   2 from v$sysmetric
   3 where metric_name in ('Database CPU Time Ratio',
   4 'Database Wait Time Ratio') and
   5 intsize_csec =
   6 (select max(INTSIZE_CSEC) from V$SYSMETRIC);

METRIC_NAME                              VALUE
-----------------------------------  -----------
Database Wait Time Ratio                11.371689
Database CPU Time Ratio                 87.831890
SQL>
```

If the query shows a very high value for the Database Wait Time Ratio, or if the Database Wait Time Ratio is much greater than the Database CPU Time Ratio, the database is spending more time waiting than processing and you must dig deeper into the Oracle wait events to identify the specific wait events causing this.

Find Detailed Information

You can use the following Oracle views to find out detailed information of what a wait event is actually waiting for and how long it has waited for each resource.

V$SESSION: This view shows the event currently being waited for as well as the event last waited for in each session.

V$SESSION_WAIT: This view lists either the event currently being waited for or the event last waited on for each session. It also shows the wait state and the wait time.

V$SESSION_WAIT_HISTORY: This view shows the last ten wait events for each current session.

V$SESSION_EVENT: This view shows the cumulative history of events waited on for each session. The data in this view is available only so long as a session is active.

V$SYSTEM_EVENT: This view shows each wait event and the time the entire instance waited for that event since you started the instance.

V$SYSTEM_WAIT_CLASS: This view shows wait event statistics by wait classes.

How It Works

Your goal in tuning performance is to minimize the total response time. If the Database Wait Time Ratio (in the query shown in the "Solution" section) is high, your response time will also be high due to waits or bottlenecks in your system. On the other hand, high values for the Database CPU Time Ratio indicate a well-running database, with few waits or bottlenecks. The Database CPU Time Ratio is calculated by dividing the total CPU used by the database by the Oracle time model statistic DB time.

Oracle uses *time model statistics* to measure the time spent in the database by the type of operation. Database time, or DB time, is the most important time model statistic—it represents the total time spent in database calls, and serves as a measure of total instance workload. DB time is computed by adding the CPU time and wait time of all sessions (excluding the waits for idle events). An AWR report shows the total DB time for the instance (in the section titled "Time Model System Stats") during the period covered by the AWR snapshots. If the time model statistic DB CPU consumes most of the DB time for the instance, it shows the database was actively processing most of the time. DB time tuning, or understanding how the database is spending its time, is fundamental to understanding performance.

The total time spent by foreground sessions making database calls consists of I/O time, CPU time, and time spent waiting for non-idle events. Your DB time will increase as the system load increases—that is, as more users log on and larger queries are executed, the greater the system load. However, even in the absence of an increase in system load, DB time can increase, due to deterioration either in I/O or application performance. As application performance degrades, wait time will increase and consequently DB time (that is, response time) will increase.

DB time is captured by internal instrumentation, ASH, AWR, and ADDM, and you can find detailed performance information by querying various views or through Enterprise Manager.

■ **Note** If the host system is CPU-bound, you'll see an increase in DB time. You must first tune CPU usage before focusing on wait events in that particular case.

The V$SESSION_WAIT view shows more detailed information than the V$SESSION_EVENT and the V$SYSTEM_EVENT views. While both the V$SESSION_EVENT and the V$SESSION_WAIT views show that there are waits such as the event `db file scattered read`, for example, only the V$SESSION_WAIT view shows the file number (P1), the block number read (P2), and the number of blocks read (P3). The columns P1 and P2 from this view help you identify the segments involved in the wait event that is currently occurring.

■ **Note** The Automatic Workload Repository (AWR) queries the V$SYSTEM_EVENT view for its wait event–related analysis.

You can first query the V$SYSTEM_EVENT view to rank the top wait events by total and average time waited for that event. You can then drill down to the wait event level, by focusing on the events at the top of the event list. Note that you can query the V$WAITSTAT view for the same information as well. In addition to providing information about blocking and blocked users and the current wait events, the V$SESSION view also shows the objects that are causing the problem, by providing the file number and block number for the object.

5-2. Identifying SQL Statements with the Most Waits

Problem

You want to identify the SQL statements responsible for the most waits in your database.

Solution

Execute the following query to identify the SQL statements that are experiencing the most waits in your database:

```
SQL> select ash.user_id,
  2  u.username,
  3  s.sql_text,
  4  sum(ash.wait_time +
  5  ash.time_waited) ttl_wait_time
  6  from v$active_session_history ash,
  7  v$sqlarea s,
  8  dba_users u
  9  where ash.sample_time between sysdate - 60/2880 and sysdate
 10  and ash.sql_id = s.sql_id
 11  and ash.user_id = u.user_id
 12  group by ash.user_id,s.sql_text, u.username
 13* order by ttl_wait_time
SQL>
```

The preceding query ranks queries that ran during the past 30 minutes, according to the total time waited by each query.

How It Works

When you're experiencing a performance problem, it's a good idea to see which SQL statements are waiting the most. These are the statements that are using most of the database's resources. To find the queries that are waiting the most, you must sum the values in the wait_time and the time_waited columns of the V$ACTIVE_SESSION_HISTORY for a specific SQL statement. In order to do this, you must join the V$SQLAREA view with the V$ACTIVE_SESSION_HISTORY view, using SQL_ID as the join column.

Besides the SQL_ID of the SQL statements, the V$ACTIVE_SESSION_HISTORY view also contains information about the execution plans used by the SQL statements. You can use this information to identify why a SQL statement is experiencing a high amount of waits. You can also run an Active Session History (ASH) report, using a SQL script or through Oracle Enterprise Manager, to get details about the

top SQL statements in the sampled session activity. The Top SQL section of an ASH report helps you identify the high-load SQL statements that are responsible for performance problems. Examining the Top SQL report may show you, for example, that one bad query has been responsible for most of the database activity.

5-3. Analyzing Wait Events

Problem

You want to analyze Oracle wait events.

Solution

Several recipes in this chapter show you how to analyze the most important Oracle wait events. An overwhelming amount of wait time in a database is due to I/O–related waits, such as those caused by either full table scans or indexed reads. While indexed reads may seem to be completely normal on the face of it, too many indexed reads can also slow down performance. Therefore, you must investigate why the database is performing a large number of indexed reads. For example, if you see the `db file sequential read` event (indicates indexed reads) at the top of the wait event list, you must look a bit further to see how the database is accumulating these read events. If you find that the database is performing hundreds of thousands of query executions, with each query doing only a few indexed reads, that's fine. However, if you find that just a couple of queries are contributing to a high number of logical reads, then, most likely, those queries are reading more data than necessary. You must tune those queries to reduce the `db file sequential read` events.

How It Works

Wait events are statistics that a server process or thread increments when it waits for an event to complete, in order to continue its processing. For example, a SQL statement may be modifying data, but the server process may have to wait for a data block to be read from disk, because it's not available in the SGA. Although there's a large number of wait events, the most common events are the following:

- buffer busy waits
- free buffer waits
- db file scattered read
- db file sequential read
- enqueue waits
- log buffer space
- log file sync

Analyzing Oracle wait events is the most important performance tuning task you'll perform when troubleshooting a slow-running query. When a query is running slow, it usually means that there are excessive waits of one type or another. Some of the waits may be due to excessive I/O due to missing

indexes. Other waits may be caused by a latch or a locking event. Several recipes in this chapter show you how to identify and fix various types of Oracle wait-related performance problems. In general, wait events that account for the most wait time warrant further investigation. However, it's important to understand that wait events show only the symptoms of underlying problems—thus, you should view a wait event as a window into a particular problem, and not the problem itself. When Oracle encounters a problem such as buffer contention or latch contention, it simply increments a specific type of wait event relating to that latch or buffer. By doing this, the database is showing where it had to wait for a specific resource, and was thus unable to continue processing. The buffer or latch contention can often be traced to faulty application logic, but some wait events could also emanate from system issues such as a misconfigured RAID system. Missing indexes, inappropriate initialization parameters, inadequate values for initialization parameters that relate to memory, and inadequate sizing of redo log files are just some of the things that can lead to excessive waits in a database. The great benefit of analyzing Oracle wait events is that it takes the guesswork out of performance tuning—you can see exactly what is causing a performance slowdown, so you can immediately focus on fixing the problem.

5-4. Understanding Wait Class Events

Problem

You want to understand how Oracle classifies wait events into various classes.

Solution

Every Oracle wait event belongs to a specific wait event class. Oracle groups wait events into classes such as `Administrative`, `Application`, `Cluster`, `Commit`, `Concurrency`, `Configuration`, `Scheduler`, `System I/O`, and `User I/O`, to facilitate the analysis of wait events. Here are some examples of typical waits in some of these classes:

`Application`: Lock-related wait information

`Commit`: Waits for confirmation of a redo log write after committing a transaction

`Network`: Waits caused by delays in sending data over the network

`User I/O`: Waiting to read blocks from disk

Two key wait classes are the `Application` and the `User I/O` wait classes. The `Application` wait class contains waits due to row and table locks caused by an application. The `User I/O` class includes the `db file scattered read`, `db file sequential read`, `direct path read`, and `direct path write` events. The `System I/O` class includes redo log–related wait events among other waits. The `Commit` class contains just the `log file sync` wait information. There's also an "idle" class of wait events such as `SQL*Net message from client` for example, that merely indicate an inactive session. You can ignore the idle waits.

How It Works

Classes of wait events help you quickly find out what type of activity is affecting database performance. For example, the `Administrative` wait class may show a high number of waits because you're rebuilding

an index. Concurrency waits point to waits for internal database resources such as latches. If the Cluster wait class shows the most wait events, then your RAC instances may be experiencing contention for global cache resources (gc cr block busy event). Note that the System I/O wait class includes waits for background process I/O such as the DBWR (database writer) wait event db file parallel write.

The Application wait class contains waits that result from user application code—most of your enqueue waits fall in this wait class. The only wait event in the Commit class is the log file sync event, which we examine in detail later in this chapter. The Configuration class waits include waits such as those caused by log files that are sized too small.

5-5. Examining Session Waits

Problem

You want to find out the wait events in a session.

Solution

You can use the V$SESSION_WAIT view to get a quick idea about what a particular session is waiting for, as shown here:

```
SQL> select event, count(*) from v$session_wait
     group by event;

EVENT                                                            COUNT(*)
---------------------------------------------------------------- ----------
SQL*Net message from client                                      11
Streams AQ: waiting for messages in the queue                    1
enq: TX - row lock contention                                    1
…
15 rows selected.

SQL>
```

The output of the query indicates that one session is waiting for an enqueue lock, possibly because of a blocking lock held by another session. If you see a large number of sessions experiencing row lock contention, you must investigate further and identify the blocking session.

Here's one more way you can query the V$SESSION_WAIT view, to find out what's slowing down a particular session:

```
SQL> select event, state, seconds_in_wait siw
     from   v$session_wait
     where  sid = 81;


EVENT                                    STATE           SIW
---------------------------------------- ----------- ------
enq: TX - row lock contention            WAITING         976
```

The preceding query shows that the session with the SID 81 has been waiting for an enqueue event, because the row (or rows) it wants to update is locked by another transaction.

■ **Note** In Oracle Database 11g, the database counts each resource wait as just one wait, even if the session experiences many internal time-outs caused by the wait. For example, a wait for an enqueue for 15 seconds may include 5 different 3-second-long wait calls—the database considers these as just a single enqueue wait.

How It Works

The first query shown in the "Solution" section offers an easy way to find out which wait events, if any, are slowing down user sessions. When you issue the query without specifying a SID, it displays the current and last waits for all sessions in the database. If you encounter a locking situation in the database, for example, you can issue the query periodically to see whether the total number of enqueue waits is coming down. If the number of enqueue waits across the instance is growing, that means more sessions are encountering slowdowns due to blocked locks.

The `V$SESSION_WAIT` view shows the current or the last wait for each session. The `STATE` column in this view tells you if a session is currently waiting. Here are the possible values for the `STATE` column:

`WAITING`: The session is currently waiting for a resource.

`WAITED UNKNOWN TIME`: The duration of the last wait is unknown (this value is shown only if you set the `TIMED_STATISTICS` parameter to false).

`WAITED SHORT TIME`: The most recent wait was less than a hundredth of a second long.

`WAITED KNOWN TIME`: The `WAIT_TIME` column shows the duration of the last wait.

Note that the query utilizes the `seconds_in_wait` column to find out how long this session has been waiting. Oracle has deprecated this column in favor of the `wait_time_micro` column, which shows the amount of time waited in microseconds. Both columns show the amount of time waited for the current wait, if the session is currently waiting. If the session is not currently waiting, the `wait_time_micro` column shows the amount of time waited during the last wait.

5-6. Examining Wait Events by Class

Problem

You want to examine Oracle wait event classes.

Solution

The following query shows the different types of wait classes and the wait events associated with each wait class.

```
SQL> select  wait_class, name
  2  from v$event_name
  3  where name LIKE 'enq%'
  4  and wait_class <> 'Other'
  5* order by wait_class
SQL> /

WAIT_CLASS                                NAME
--------------------                 --------------------------
Administrative                       enq: TW - contention
Concurrency                          enq: TX - index contention
…
SQL>
```

To view the current waits grouped into various wait classes, issue the following query:

```
SQL> select wait_class, sum(time_waited), sum(time_waited)/sum(total_waits)
  2  sum_waits
  3  from v$system_wait_class
  4  group by wait_class
  5* order by 3 desc;

WAIT_CLASS                            SUM(TIME_WAITED)               SUM_WAITS
--------------- ----------           -----------------        -----------------
Idle                                         249659211               347.489249
Commit                                         1318006               236.795904
Concurrency                                      16126                 4.818046
User I/O                                        135279                 2.228869
Application                                        912                 .0928055
Network                                            139                 .0011209
…
SQL>
```

If you see a very high sum of waits for the `Idle` wait class, not to worry—actually, you should expect to see this in any healthy database. In a typical production environment, however, you'll certainly see more waits under the `User I/O` and `Application` wait classes. If you notice that the database has accumulated a very large wait time for the `Application` wait class, or the `User I/O` wait class, for example, it's time to investigate those two wait classes further. In the following example, we drill down into a couple of wait classes to find out which specific waits are causing the high sum of total wait time under the `Application` and `Concurrency` classes. To do this, we use the `V$SYSTEM_EVENT` and the `$EVENT_NAME` views in addition to the `V$SYSTEM_WAIT_CLASS` view. Focus not just on the total time waited, but also on the average wait, to gauge the effect of the wait event.

```
SQL> select a.event, a.total_waits, a.time_waited, a.average_wait
     from v$system_event a, v$event_name b, v$system_wait_class c
     where a.event_id=b.event_id
     and b.wait_class#=c.wait_class#
     and c.wait_class in ('Application','Concurrency')
     order by average_wait desc;
```

```
EVENT                       TOTAL_WAITS   TIME_WAITED    AVERAGE_WAIT
--------------------------- -----------   -------------  -------------
enq: UL - contention                  1             499         499.19

latch: shared pool                  251           10944           43.6

library cache load lock              24             789          32.88

SQL>
```

■ **Tip** Two of the most common Oracle wait events are the `db file scattered read` and the `db file sequential read` events. The `db file scattered read` wait event is due to full table scans of large tables. If you experience this wait event, investigate the possibility of adding indexes to the table or tables. The `db file sequential read` wait event is due to indexed reads. While an indexed read may seem like it's a good thing, a very high amount of indexed reads could potentially indicate an inefficient query that you must tune. If high values for the `db file sequential read` wait event are due to a very large number of small indexed reads, it's not really a problem—this is natural in a database. You should be concerned if a handful of queries are responsible for most of the waits.

You can see that the enqueue waits caused by the row lock contention are what's causing the most waits under these two classes. Now you know exactly what's slowing down the queries in your database! To get at the session whose performance is being affected by the contention for the row lock, drill down to the session level using the following query:

```
SQL> select a.sid, a.event, a.total_waits, a.time_waited, a.average_wait
     from v$session_event a, v$session b
     where time_waited > 0
     and a.sid=b.sid
     and b.username is not NULL
     and a.event='enq: TX - row lock contention';

  SID              EVENT                       TOTAL_WAITS    time_waited      average_wait
----------     ------------------------------  ------------   -----------      ------------
        68     enq: TX - row lock contention             24           8018               298
SQL>
```

The output shows that the session with the SID 68 is waiting for a row lock that's held by another transaction.

How It Works

Understanding the various Oracle wait event classes enhances your ability to quickly diagnose Oracle wait-related problems. Analyzing wait events by classes lets you know if contention, user I/O, or a configuration issue is responsible for high waits. The examples in the "Solution" section show you how

to start analyzing the waits based on the wait event classes. This helps identify the source of the waits, such as concurrency issues, for example. Once you identify the wait event class responsible for most of the waits, you can drill down into that wait event class to find out the specific wait events that are contributing to high total waits for that wait event class. You can then identify the user sessions waiting for those wait events, using the final query shown in the "Solution" section.

5-7. Resolving Buffer Busy Waits

Problem

Your database is experiencing a high number of buffer busy waits, based on the output from the AWR report. You want to resolve those waits.

Solution

Oracle has several types of buffer classes, such as data block, segment header, undo header, and undo block. How you fix a buffer busy wait situation will depend on the types of buffer classes that are causing the problem. You can find out the type of buffer causing the buffer waits by issuing the following two queries. Note that you first get the value of `row_wait_obj#` from the first query and use it as the value for `data_object_id` in the second query.

```
SQL> select row_wait_obj#
     from v$session
     where event = 'buffer busy waits';

SQL> select owner, object_name, subobject_name, object_type
     from dba_objects
     where data_object_id = &row_wait_obj;
```

The preceding queries will reveal the specific type of buffer causing the high buffer waits. Your fix will depend on which buffer class causes the buffer waits, as summarized in the following subsections.

Segment Header

If your queries show that the buffer waits are being caused by contention on the segment header, there's free list contention in the database, due to several processes attempting to insert into the same data block—each of these processes needs to obtain a free list before it can insert data into that block. If you aren't already using it, you must switch from manual space management to automatic segment space management (ASSM)—under ASSM, the database doesn't use free lists. However, note that moving to ASSM may not be easily feasible in most cases. In cases where you can't implement ASSM, you must increase the free lists for the segment in question. You can also try increasing the free list groups as well.

Data Block

Data block buffer contention could be related to a table or an index. This type of contention is often caused by right-hand indexes, that is, indexes that result in several processes inserting into the same

point, such as when you use sequence number generators to produce the key values. Again, if you're using manual segment management, move to ASSM or increase free lists for the segment.

Undo Header and Undo Block

If you're using automatic undo management, few or none of the buffer waits will be due to contention for an undo segment header or an undo segment block. If you do see one of these buffer classes as the culprit, however, you may increase the size of your undo tablespace to resolve the buffer busy waits.

How It Works

A buffer busy wait indicates that more than one process is simultaneously accessing the same data block. One of the reasons for a high number of buffer busy waits is that an inefficient query is reading too many data blocks into the buffer cache, thus potentially keeping in wait other sessions that want to access one or more of those same blocks. Not only that, a query that reads too much data into the buffer cache may lead to the aging out of necessary blocks from the cache. You must investigate queries that involve the segment causing the buffer busy waits with a view to reducing the number of data blocks they're reading into the buffer cache.

If your investigation of buffer busy waits reveals that the same block or set of blocks is involved most of the time, a good strategy would be to delete some of these rows and insert them back into the table, thus forcing them onto different data blocks.

Check your current memory allocation to the buffer cache, and, if necessary, increase it. A larger buffer cache can reduce the waiting by sessions to read data from disk, since more of the data will already be in the buffer cache. You can also place the offending table in memory by using the `KEEP POOL` in the buffer cache (please see Recipe 3-7). By making the hot block always available in memory, you'll avoid the high buffer busy waits.

Indexes that have a very low number of unique values are called low cardinality indexes. Low cardinality indexes generally result in too many block reads. Thus, if several DML operations are occurring concurrently, some of the index blocks could become "hot" and lead to high buffer busy waits. As a long-term solution, you can try to reduce the number of the low cardinality indexes in your database.

Each Oracle data segment such as a table or an index contains a header block that records information such as free blocks available. When multiple sessions are trying to insert or delete rows from the same segment, you could end up with contention for the data segment's header block.

Buffer busy waits are also caused by a contention for free lists. A session that's inserting data into a segment needs to first examine the free list information for the segment, to find blocks with free space into which the session can insert data. If you use ASSM in your database, you shouldn't see any waits due to contention for a free list.

5-8. Resolving Log File Sync Waits

Problem

You're seeing a high amount of `log file sync` wait events, which are at the top of all wait events in your database. You want to reduce these wait events.

Solution

The following are two strategies for dealing with high `log file sync` waits in your database.

- If you notice a very large number of waits with a short average wait time per wait, that's an indication that too many commit statements are being issued by the database. You must change the commit behavior by batching the commits. Instead of committing after each row, for example, you can specify that the commits occur after every 500 rows.
- If you notice that the large amount of wait time accumulated due to the `redo log file sync` event was caused by long waits for writing to the redo log file (high average time waited for this event), it's more a matter of how fast your I/O subsystem is. You can alternate the redo log files on various disks to reduce contention. You can also see if you can dedicate disks entirely for the redo logs instead of allowing other files on those disks—this will reduce I/O contention when the LGWR is writing the buffers to disk. Finally, as a long-term solution, you can look into placing redo logs on faster devices, say, by moving them from a RAID 5 to a RAID 1 device.

How It Works

Oracle (actually the LGWR background process) automatically flushes a session's redo information to the redo log file whenever a session issues a `COMMIT` statement. The database writes commit records to disk before it returns control to the client. The server process thus waits for the completion of the write to the redo log. This is the default behavior, but you can also control the database commit behavior with the `COMMIT_WRITE` initialization parameter.

■ **Note** The `COMMIT_WRITE` parameter is an advanced parameter that has been deprecated in Oracle Database 11.2. Since it may have an adverse impact on performance, you may want to leave the parameter alone and rely on Oracle's default commit behavior.

The session will tell the LGWR process to write the session's redo information from the redo log buffer to the redo log file on disk. The LGWR process posts the user session after it finishes writing the buffer's contents to disk. The `log file sync` wait event includes the wait during the writing of the log buffer to disk by LGWR and the posting of that information to the session. The server process will have to wait until it gets confirmation that the LGWR process has completed writing the log buffer contents out to the redo log file.

The `log file sync` events are caused by contention during the writing of the log buffer contents to the redo log files. Check the `V$SESSION_WAIT` view to ascertain whether Oracle is incrementing the `SEQ#` column. If Oracle is incrementing this column, it means that the LGWR process is the culprit, as it may be stuck.

As the log file sync wait event is caused by contention caused by the LGWR process, see if you can use the NOLOGGING option to get rid of these waits. Of course, in a production system, you can't use the NOLOGGING option when the database is processing user requests, so this option is of limited use in most cases.

The log file sync wait event can also be caused by too large a setting for the LOG_BUFFER initialization parameter. Too large a value for the LOG_BUFFER parameter will lead the LGWR process to write data less frequently to the redo log files. For example, if you set the LOG BUFFER to something like 12 MB, it sets an internal parameter, log_io_size, to a high value. The log_io_size parameter acts as a threshold for when the LGWR writes to the redo log files. In the absence of a commit request or a checkpoint, LGWR waits until the log_io_size threshold is met. Thus, when the database issues a COMMIT statement, the LGWR process would be forced to write a large amount of data to the redo log files at once, resulting in sessions waiting on the log file sync wait event. This happens because each of the waiting sessions is waiting for LGWR to flush the contents of the redo log buffer to the redo log files. Although the database automatically calculates the value of the log_io_size parameter, you can specify a value for it, by issuing a command such as the following:

```
SQL> alter system set "_log_io_size"=1024000 scope=spfile;

System altered.

SQL>
```

5-9. Minimizing read by other session Wait Events

Problem

Your AWR report shows that the read by other session wait event is responsible for the highest number of waits. You'd like to reduce the high read by other session waits.

Solution

The main reason you'll see the read by other session wait event is that multiple sessions are seeking to read the same data blocks, whether they are table or index blocks, and are forced to wait behind the session that's currently reading those blocks. You can find the data blocks a session is waiting for by executing the following command:

```
SQL> select p1 "file#", p2 "block#", p3 "class#"
     from v$session_wait
     where event = 'read by other session';
```

You can then take the block# and use it in the following query, to identify the exact segments (table or index) that are causing the read by other session waits.

```
SQL> select relative_fno, owner, segment_name, segment_type
     from dba_extents
     where file_id = &file
     and &block between block_id
     and block_id + blocks - 1;
```

Once you identify the hot blocks and the segments they belong to, you need to identify the queries that use these data blocks and segments and tune those queries if possible. You can also try deleting and re-inserting the rows inside the hot blocks.

In order to reduce the amount of data in each of the hot blocks and thus reduce these types of waits, you can also try to create a new tablespace with a smaller block size and move the segment to that tablespace. It's also a good idea to check if any low cardinality indexes are being used, because this type of an index will make the database read a large number of data blocks into the buffer cache, potentially leading to the `read by other session` wait event. If possible, replace any low cardinality indexes with an index on a column with a high cardinality.

How It Works

The `read by other session` wait event indicates that one or more sessions are waiting for another session to read the same data blocks from disk into the SGA. Obviously, a large number of these waits will slow down performance. Your first goal should be to identify the actual data blocks and the objects the blocks belong to. For example, these waits can be caused by multiple sessions trying to read the same index blocks. Multiple sessions can also be trying to execute a full table scan simultaneously on the same table.

5-10. Reducing Direct Path Read Wait Events

Problem

You notice a high amount of the `direct path read` wait events, and also of `direct path read temp` wait events, and you'd like to reduce the occurrence of those events.

Solution

`Direct path read` and `direct path read temp` events are related wait events that occur when sessions are reading data directly into the PGA instead of reading it into the SGA. Reading data into the PGA isn't the problem here—that's normal behavior for certain operations, such as sorting, for example. The `direct path read` and `direct path read temp` events usually indicate that that the sorts being performed are very large and that the PGA is unable to accommodate those sorts.

Issue the following command to get the file ID for the blocks that are being waited for:

```
SQL> select p1 "file#", p2 "block#", p3 "class#"
     from v$session_wait
     where event = 'direct path read temp';
```

The column `P1` shows the file ID for the read call. Column `P2` shows the start `BLOCK_ID`, and column `P3` shows the number of blocks. You can then execute the following statement to check whether this file ID is for a temporary tablespace tempfile:

```
SQL> select relative_fno, owner, segment_name, segment_type
     from dba_extents
     where file_id = &file
     and &block betgween block_id and block_id + &blocks - 1;
```

The direct read–type waits can be caused by excessive sorts to disk or full table scans. In order to find out what the reads are actually for, check the `P1` column (file ID for the read call) of the `V$SESSION_WAIT` view. By doing this, you can find out if the reads are being caused by reading data from the `TEMP` tablespace due to disk sorting, or if they're occurring due to full table scans by parallel slaves.

If you determine that sorts to disk are the main culprit in causing high `direct read` wait events, increase the value of the `PGA_AGGREGATE_TARGET` parameter (or specify a minimum size for it, if you're using automatic memory management). Increasing PGA size is also a good strategy when the queries are doing large hash joins, which could result in excessive I/O on disk if the PGA is inadequate for handling the large hash joins. When you set a high degree of parallelism for a table, Oracle tends to go for full table scans, using parallel slaves. If your I/O system can't handle all the parallel slaves, you'll notice a high amount of direct path reads. The solution for this is to reduce the degree of parallelism for the table or tables in question. Also investigate if you can avoid the full table scan by specifying appropriate indexes.

How It Works

Normally, during both a sequential db read or a scattered db read operation, the database reads data from disk into the SGA. A `direct path read` is one where a single or multiblock read is made from disk directly to the PGA, bypassing the SGA. Ideally, the database should perform the entire sorting of the data in the PGA. When a huge sort doesn't fit into the available PGA, Oracle writes part of the sort data directly to disk. A direct read occurs when the server process reads this data from disk (instead of the PGA).

A `direct path read` event can also occur when the I/O subsystem is overloaded, most likely due to full table scans caused by setting a high degree of parallelism for tables, causing the database to return buffers slower than what the processing speed of the server process requires. A good disk striping strategy would help out here. Oracle's Automatic Storage Management (ASM) automatically stripes data for you. If you aren't already using ASM, consider implementing it in your database.

`Direct path write` and `direct path write temp` wait events are analogous to the `direct path read` and the `direct path read temp` waits. Normally, it's the DBWR that writes data from the buffer cache. Oracle uses a `direct path write` when a process writes data buffers directly from the PGA. If your database is performing heavy sorts that spill onto disk, or parallel DML operations, you can on occasion expect to encounter the `direct path write` events. You may also see this wait event when you execute direct path load events such as a parallel `CTAS` (`create table as select`) or a direct path `INSERT` operation. As with the `direct path read` events, the solution for `direct path write` events depends on what's causing the waits. If the waits are being mainly caused by large sorts, then you may think about increasing the value of the `PGA_AGGREGATE_TARGET` parameter. If operations such as parallel DML are causing the waits, you must look into the proper spreading of I/O across all disks and also ensure that your I/O subsystem can handle the high degree of parallelism during DML operations.

5-11. Minimizing Recovery Writer Waits

Problem

You've turned on the Oracle Flashback Database feature in your database. You're now seeing a large number of wait events due to a slow RVWR (recovery writer) process. You want to reduce the recovery writer waits.

Solution

Oracle writes all changed blocks from memory to the flashback logs on disk. You may encounter the `flashback buf free by RVWR` wait event as a top wait event when thie database is writing to the flashback logs. To reduce these recovery writer waits, you must tune the flash recovery area file system and storage. Specifically, you must do the following:

- Since flashback logs tend to be quite large, your database is going to incur some CPU overhead when writing to these files. One of the things you may consider is moving the flash recovery area to a faster file system. Also, Oracle recommends that you use file systems based on ASM, because they won't be subject to operating system file caching, which tends to slow down I/O.
- Increase the disk throughput for the file system where you store the flash recovery area, by configuring multiple disk spindles for that file system. This will speed up the writing of the flashback logs.
- Stripe the storage volumes, ideally with small stripe sizes (for example, 128 KB).
- Set the `LOG_BUFFER` initialization parameter to a minimum value of 8 MB—the memory allocated for writing to the flashback database logs depends on the setting of the `LOG_BUFFER` parameter.

How It Works

Unlike in the case of the redo log buffer, Oracle writes flashback buffers to the flashback logs at infrequent intervals to keep overhead low for the Oracle Flashback Database. The `flashback buf free by RVWR` wait event occurs when sessions are waiting on the RVWR process. The RVWR process writes the contents of the flashback buffers to the flashback logs on disk. When the RVWR falls behind during this process, the flashback buffer is full and free buffers aren't available to sessions that are making changes to data through DML operations. The sessions will continue to wait until RVWR frees up buffers by writing their contents to the flashback logs. High RVWR waits indicate that your I/O system is unable to support the rate at which the RVWR needs to flush flashback buffers to the flashback logs on disk.

5-12. Finding Out Who's Holding a Blocking Lock

Problem

Your users are complaining that some of their sessions are very slow. You suspect that those sessions may be locked by Oracle for some reason, and would like to find the best way to go about figuring out who is holding up these sessions.

Solution

As we've explained in the introduction to this chapter, Oracle uses several types of locks to control transactions being executed by multiple sessions, to prevent destructive behavior in the database. A blocking lock could "slow" a session down—in fact, the session is merely waiting on another session that

is holding a lock on an object (such as a row or a set of rows, or even an entire table). Or, in a development scenario, a developer might have started multiple sessions, some of which are blocking each other.

When analyzing Oracle locks, some of the key database views you must examine are the V$LOCK and the V$SESSION views. The V$LOCKED_OBJECT and the DBA_OBJECTS views are also very useful in identifying the locked objects. In order to find out whether a session is being blocked by the locks being applied by another session, you can execute the following query:

```
SQL> select s1.username || '@' || s1.machine
  2  || ' ( SID=' || s1.sid || ' )  is blocking '
  3  || s2.username || '@' || s2.machine || ' ( SID=' || s2.sid || ' ) ' AS blocking_status
  4  from v$lock l1, v$session s1, v$lock l2, v$session s2
  5  where s1.sid=l1.sid and s2.sid=l2.sid
  6  and l1.BLOCK=1 and l2.request > 0
  7  and l1.id1 = l2.id1
  8  and l2.id2 = l2.id2 ;


BLOCKING_STATUS
--------------------------------------------------------------------------

HR@MIRO\MIROPC61 ( SID=68 )  is blocking SH@MIRO\MIROPC61 ( SID=81 )

SQL>
```

The output of the query shows the blocking session as well as all the blocked sessions.

A quick way to find out if you have any blocking locks in your instance at all, for any user, is to simply run the following query:

```
SQL> select * from V$lock where block > 0;
```

If you don't get any rows back from this query—good—you don't have any blocking locks in the instance right now! We'll explain this view in more detail in the explanation section.

How It Works

Oracle uses two types of locks to prevent destructive behavior: exclusive and shared locks. Only one transaction can obtain an exclusive lock on a row or a table, while multiple shared locks can be obtained on the same object. Oracle uses locks at two levels—row and table levels. Row locks, indicated by the symbol TX, lock just a single row of a table for each row that'll be modified by a DML statement such as INSERT, UPDATE, and DELETE. This is true also for a MERGE or a SELECT … FOR UPDATE statement. The transaction that includes one of these statements grabs an exclusive row lock as well as a row share table lock. The transaction (and the session) will hold these locks until it commits or rolls back the statement. Until it does one of these two things, all other sessions that intend to modify that particular row are blocked. Note that each time a transaction intends to modify a row or rows of a table, it holds a table lock (TM) as well on that table, to prevent the database from allowing any DDL operations (such as DROP TABLE) on that table while the transaction is trying to modify some of its rows.

In an Oracle database, locking works this way:

- A reader won't block another reader.
- A reader won't block a writer.
- A writer won't block a reader of the same data.
- A writer will block another writer that wants to modify the same data.

It's the last case in the list, where two sessions intend to modify the same data in a table, that Oracle's automatic locking kicks in, to prevent destructive behavior. The first transaction that contains the statement that updates an existing row will get an exclusive lock on that row. While the first session that locks a row continues to hold that lock (until it issues a `COMMIT` or `ROLLBACK` statement), other sessions can modify any other rows in that table other than the locked row. The concomitant table lock held by the first session is merely intended to prevent any other sessions from issuing a DDL statement to alter the table's structure. Oracle uses a sophisticated locking mechanism whereby a row-level lock isn't automatically escalated to the table, or even the block level.

5-13. Identifying Blocked and Blocking Sessions

Problem

You notice enqueue locks in your database and suspect that a blocking lock may be holding up other sessions. You'd like to identify the blocking and the blocked sessions.

Solution

When you see an enqueue wait event in an Oracle database, chances are that it's a locking phenomenon that's holding up some sessions from executing their SQL statements. When a session waits on an "enqueue" wait event, that session is waiting for a lock that's held by a different session. The blocking session is holding the lock in a mode that's incompatible with the lock mode that's being requested by the blocked session. You can issue the following command to view information about the blocked and the blocking sessions:

```
SQL> select decode(request,0,'Holder: ','Waiter: ')||sid sess,
     id1, id2, lmode, request, type
     from v$lock
     where (id1, id2, type) in
     (select id1, id2, type from v$lock where request>0)
     order by id1, request;
```

The `V$LOCK` view shows if there are any blocking locks in the instance. If there are blocking locks, it also shows the blocking session(s) and the blocked session(s). Note that a blocking session can block multiple sessions simultaneously, if all of them need the same object that's being blocked. Here's an example that shows there are locks present:

```
 SQL> select sid,type,lmode,request,ctime,block from v$lock;

       SID           TY           LMODE         REQUEST       CTIME        BLOCK
--------------     --------     -----------    -----------   --------     -------
       127           MR              4              0         102870          0
        81           TX              0              6            778          0
       191           AE              4              0            758          0
       205           AE              4              0            579          0
       140           AE              4              0          11655          0
        68           TM              3              0            826          0
        68           TX              6              0            826          1
…
SQL>
```

The key column to watch is the `BLOCK` column—the blocking session will have the value 1 for this column. In our example, session 68 is the blocking session, because it shows the value 1 under the `BLOCK` column. Thus, the `V$LOCK` view confirms our initial finding in the "Solution" section of this recipe. The blocking session, with a SID of 68, also shows a lock mode 6 under the `LMODE` column, indicating that it's holding this lock in the exclusive mode—this is the reason session 81 is "hanging," unable to perform its update operation. The blocked session, of course, is the victim—so it shows a value of 0 in the `BLOCK` column. It also shows a value of 6 under the `REQUEST` column, because it's requesting a lock in the exclusive mode to perform its update of the column. The blocking session, in turn, will show a value of 0 for the `REQUEST` column, because it isn't requesting any locks—it's already holding it.

If you want to find out the wait class and for how long a blocking session has been blocking others, you can do so by querying the `V$SESSION` view, as shown here:

```
SQL> select  blocking_session, sid,  wait_class,
     seconds_in_wait
     from     v$session
     where blocking_session is not NULL
     order by blocking_session;

BLOCKING_SESSION        SID        WAIT_CLASS        SECONDS_IN_WAIT
-----------------    --------       -------------    ----------------
      68                81          Application                 7069

SQL>
```

The query shows that the session with SID=68 is blocking the session with SID=81, and the block started 7,069 seconds ago.

How It Works

The following are the most common types of enqueue locks you'll see in an Oracle database:

- `TX`: These are due to a transaction lock and usually caused by faulty application logic.
- `TM`: These are table-level DML locks, and the most common cause is that you haven't indexed foreign key constraints in a child table.

In addition, you are also likely to notice ST enqueue locks on occasion. These indicate sessions that are waiting while Oracle is performing space management operations, such as the allocation of temporary segments for performing a sort.

5-14. Dealing with a Blocking Lock

Problem

You've identified blocking locks in your database. You want to know how to deal with those locks.

Solution

There are two basic strategies when dealing with a blocking lock—a short-term and a long-term strategy. The first thing you need to do is get rid of the blocking lock, so the sessions don't keep queuing up—it's not at all uncommon for a single blocking lock to result in dozens and even hundreds of sessions, all waiting for the blocked object. Since you already know the `SID` of the blocking session (session 68 in our example), just kill the session in this way, after first querying the `V$SESSION` view for the corresponding `serial#` for the session:

```
SQL> alter system kill session '68, 1234';
```

The short-term solution is to quickly get rid of the blocking locks so they don't hurt the performance of your database. You get rid of them by simply killing the blocking session. If you see a long queue of blocked sessions waiting behind a blocking session, kill the blocking session so that the other sessions can get going.

For the long run, though, you must investigate why the blocking session is behaving the way that it is. Usually, you'll find a flaw in the application logic. You may, though, need to dig deep into the SQL code that the blocking session is executing.

How It Works

In this example, obviously, the blocking lock is a DML lock. However, even if you didn't know this ahead of time, you can figure out the type of lock by examining the `TYPE (TY)` column of the `V$LOCK` view. Oracle uses several types of internal "system" locks to maintain the library cache and other instance-related components, but those locks are normal and you won't find anything related to those locks in the `V$LOCK` view.

For DML operations, Oracle uses two basic types of locks—transaction locks (TX) and DML locks (TM). There is also a third type of lock, a user lock (UL), but it doesn't play a role in troubleshooting general locking issues. Transaction locks are the most frequent type of locks you'll encounter when troubleshooting Oracle locking issues. Each time a transaction modifies data, it invokes a TX lock, which is a row transaction lock. The DML lock, TM, on the other hand, is acquired once for each object that's being changed by a DML statement.

The LMODE column shows the lock mode, with a value of 6 indicating an exclusive lock. The REQUEST column shows the requested lock mode. The session that first modifies a row will hold an exclusive lock with LMODE=6. This session's REQUEST column will show a value of 0, since it's not requesting a lock—it already has one! The blocked session needs but can't obtain an exclusive lock on the same rows, so it requests a TX in the exclusive mode (MODE=6) as well. So, the blocked session's REQUEST column will show a value of 6, and its LMODE column a value of 0 (a blocked session has no lock at all in any mode).

The preceding discussion applies to row locks, which are always taken in the exclusive mode. A TM lock is normally acquired in mode 3, which is a Shared Row Exclusive mode, whereas a DDL statement will need a TM exclusive lock.

5-15. Identifying a Locked Object

Problem

You are aware of a locking situation, and you'd like to find out the object that's being locked.

Solution

You can find the locked object's identity by looking at the value of the ID1 (LockIdentifier) column in the V$LOCK view (see Recipe 5-13). The value of the ID1 column where the TYPE column is TM (DML enqueue) identifies the locked object. Let's say you've ascertained that the value of the ID1 column is 99999. You can then issue the following query to identify the locked table:

```
SQL> select object_name from dba_objects where object_id=99999;

OBJECT_NAME
------------
TEST
SQL>
```

An even easier way is to use the V$LOCKED_OBJECT view to find out the locked object, the object type, and the owner of the object.

```
SQL> select lpad(' ',decode(l.xidusn,0,3,0)) || l.oracle_username "User",
     o.owner, o.object_name, o.object_type
     from v$locked_object l, dba_objects o
     where l.object_id = o.object_id
     order by o.object_id, 1 desc;
```

```
User       OWNER      OBJECT_NAME      OBJECT_TYPE
------     ------     ------------     ------------
HR         HR         TEST             TABLE
SH         HR         TEST             TABLE

SQL>
```

Note that the query shows both the blocking and the blocked users.

How It Works

As the "Solution" section shows, it's rather easy to identify a locked object. You can certainly use Oracle Enterprise Manager to quickly identify a locked object, the ROWID of the object involved in the lock, and the SQL statement that's responsible for the locks. However, it's always important to understand the underlying Oracle views that contain the locking information, and that's what this recipe demonstrates. Using the queries shown in this recipe, you can easily identify a locked object without recourse to a monitoring tool such as Oracle Enterprise Manager, for example.

In the example shown in the solution, the locked object was a table, but it could be any other type of object, including a PL/SQL package. Often, it turns out that the reason a query is just hanging is that one of the objects the query needs is locked. You may have to kill the session holding the lock on the object before other users can access the object.

5-16. Resolving enq: TM Lock Contention

Problem

Several sessions in your database are taking a very long time to process some insert statements. As a result, the "active" sessions count is very high and the database is unable to accept new session connections. Upon checking, you find that the database is experiencing a lot of `enq: TM – contention` wait events.

Solution

The `enq: TM – contention` event is usually due to missing foreign key constraints on a table that's part of an Oracle DML operation. Once you fix the problem by adding the foreign key constraint to the relevant table, the `enq: TM – contention` event will go away.

The waits on the `enq: TM – contention` event for the sessions that are waiting to perform insert operations are almost always due to an unindexed foreign key constraint.. This happens when a dependent or child table's foreign key constraint that references a parent table is missing an index on the associated key. Oracle acquires a table lock on a child table if it's performing modifications on the primary key column in the parent table that's referenced by the foreign key of the child table. Note that these are full table locks (`TM`), and not row-level locks (`TX`)—thus, these locks aren't restricted to a row but to the entire table. Naturally, once this table lock is acquired, Oracle will block all other sessions that seek to modify the child table's data. Once you create an index in the child table performing on the column that references the parent table, the waits due to the `TM` contention will go away.

How It Works

Oracle takes out an exclusive lock on a child table if you don't index the foreign key constraints in that table. To illustrate how an unindexed foreign key will result in contention due to locking, we use the following example. Create two tables, STORES and PRODUCTS, as shown here:

```
SQL> create table stores
     (store_id     number(10)     not null,
     supplier_name     varchar2(40)     not null,
     constraint stores_pk PRIMARY KEY (store_id));
SQL>create table products
    (product_id     number(10)     not null,
    product_name    varchar2(30)    not null,
    supplier_id     number(10)     not null,
    store_id     number(10)     not null,
    constraint fk_stores
    foreign key (store_id)
    references stores(store_id)
    on delete cascade);
```

If you now delete any rows in the STORES table, you'll notice waits due to locking. You can get rid of these waits by simply creating an index on the column you've specified as the foreign key in the PRODUCTS table:

```
create index fk_stores on products(store_id);
```

You can find all unindexed foreign key constraints in your database by issuing the following query:

```
SQL> select * from (
     select c.table_name, co.column_name, co.position column_position
     from   user_constraints c, user_cons_columns co
     where  c.constraint_name = co.constraint_name
     and   c.constraint_type = 'R'
     minus
     select ui.table_name, uic.column_name, uic.column_position
     from   user_indexes ui, user_ind_columns uic
     where  ui.index_name = uic.index_name
     )
     order by table_name, column_position;
```

If you don't index a foreign key column, you'll notice the child table is often locked, thus leading to contention-related waits. Oracle recommends that you always index your foreign keys.

■ **Tip** If the matching unique or primary key for a child table's foreign key never gets updated or deleted, you don't have to index the foreign key column in the child table.

Oracle will tend to acquire a table lock on the child table if you don't index the foreign key column. If you insert a row into the parent table, the parent table doesn't acquire a lock on the child table; however, if you update or delete a row in the parent table, the database will acquire a full table lock on

the child table. That is, any modifications to the primary key in the parent table will result in a full table lock (TM) on the child table. In our example, the STORES table is a parent of the PRODUCTS table, which contains the foreign key STORE_ID. The table PRODUCTS being a dependent table, the values of the STORE_ID column in that table must match the values of the unique or primary key of the parent table, STORES. In this case, the STORE_ID column in the STORES table is the primary key of that table.

Whenever you modify the parent table's (STORES) primary key, the database acquires a full table lock on the PRODUCTS table. Other sessions can't change any values in the PRODUCTS table, including the columns other than the foreign key column. The sessions can only query but not modify the PRODUCTS table. During this time, any sessions attempting to modify any column in the PRODUCTS table will have to wait (TM: enq contention wait). Oracle will release this lock on the child table PRODUCTS only after it finishes modifying the primary key in the parent table, STORES. If you have a bunch of sessions waiting to modify data in the PRODUCTS table, they'll all have to wait, and the active session count naturally will go up very fast, if you've an online transaction processing–type database that has many users that perform short DML operations. Note that any DML operations you perform on the child table don't require a table lock on the parent table.

5-17. Identifying Recently Locked Sessions

Problem

A session is experiencing severe waits in the database, most likely due to a blocking lock placed by another session. You've tried to use the V$LOCK and other views to drill deeper into the locking issue, but are unable to "capture" the lock while it's in place. You'd like to use a different view to "see" the older locking data that you might have missed while the locking was going on.

Solution

You can execute the following statement based on ASH, to find out information about all locks held in the database during the previous five minutes. Of course, you can vary the time interval to a smaller or larger period, so long as there's ASH data covering that time period.

```
SQL> select to_char(h.sample_time, 'HH24:MI:SS') TIME,h.session_id,
     decode(h.session_state, 'WAITING' ,h.event, h.session_state) STATE,
     h.sql_id,
     h.blocking_session BLOCKER
     from v$active_session_history h, dba_users u
     where u.user_id = h.user_id
     and h.sample_time > SYSTIMESTAMP-(2/1440);

TIME            SID                       STATE                        SQL_ID          BLOCKER
------------   -----      ----------------------------------     -------------    ---------
17:00:52        197         116 enq: TX - row lock contention     094w6n53tnywr     191
17:00:51        197         116 enq: TX - row lock contention     094w6n53tnywr     191
17:00:50        197         116 enq: TX - row lock contention     094w6n53tnywr     191

…

SQL>
```

You can see that ASH has recorded all the blocks placed by session 1, the blocking session (SID=191) that led to a "hanging" situation for session 2, the blocked session (SID=197).

How It works

Often, when your database users complain about a performance problem, you may query the V$SESSION or V$LOCK views, but you may not find anything useful there, because the wait issue may have been already resolved by then. In these circumstances you can query the V$ACTIVE_SESSION_HISTORY view to find out what transpired in the database during the previous 60 minutes. This view offers a window into the Active Session History (ASH), which is a memory buffer that collects information about all active sessions, every second. The V$ACTIVE_SESSION_HISTORY contains one row for each active session, and newer information continuously overwrites older data, since ASH is a rolling buffer.

We can best demonstrate the solution by creating the scenario that we're discussing, and then working through that scenario. Begin by creating a test table with a couple of columns:

```
SQL>  create table test (name varchar(20), id number (4));
Table created.
SQL>
```

Insert some data into the test table.

```
SQL> insert into test values ('alapati','9999');
1 row created.
SQL> insert into test values ('sam', '1111');
1 row created.
SQL> commit;
Commit complete.
SQL>
```

In session 1 (the current session), execute a SELECT * FOR UPDATE statement on the table TEST—this will place a lock on that table.

```
SQL> select * from test for update;


SQL>
```

In a different session, session 2, execute the following UPDATE statement:

```
SQL> update test set name='Jackson' where id = '9999';
```

Session 2 will hang now, because it's being blocked by the SELECT FOR UPDATE statement issued by session 1. Go ahead now and issue either a ROLLBACK or a COMMIT from session 1:

```
SQL> rollback;
Rollback complete.
SQL>
```

When you issue the ROLLBACK statement, session 1 releases all locks it's currently holding on table TEST. You'll notice that session 2, which has been blocked thus far, immediately processes the UPDATE statement, which was previously "hanging," waiting for the lock held by session 2.

Therefore, we know for sure that there was a blocking lock in your database for a brief period, with session 1 the blocking session, and session 2 the blocked session. You can't find any evidence of this in the V$LOCK view, though, because that and all other lock-related views show you details only about currently held locks. Here's where the Active Session History views shine—they can provide you

information about locks that have been held recently but are gone already before you can view them with a query on the V$LOCK or V$SESSION views.

■ **Caution** Be careful when executing the Active Session History (ASH) query shown in the "Solution" section of this recipe. As the first column (SAMPLE_TIME) shows, ASH will record session information every second. If you execute this query over a long time frame, you may get a very large amount of output just repeating the same locking information. To deal with that output, you may specify the SET PAUSE ON option in SQL*Plus. That will pause the output every page, enabling you to scroll through a few rows of the output to identify the problem.

Use the following query to find out the wait events for which this session has waited during the past hour.

```
SQL> select sample_time, event, wait_time
     from v$active_session_history
     where session_id = 81
     and session_serial# = 422;
```

The column SAMPLE_TIME lets you know precisely when this session suffered a performance hit due to a specific wait event. You can identify the actual SQL statement that was being executed by this session during that period, by using the V$SQL view along with the V$ACTIVE_SESSION_HISTORY view, as shown here:

```
SQL> select sql_text, application_wait_time
     from v$sql
     where sql_id in ( select sql_id from v$active_session_history
     where sample_time =  '08-MAR-11 05.00.52.00 PM'
     and session_id = 68 and session_serial# = 422);
```

Alternatively, if you have the SQL_ID already from the V$ACTIVE_SESSION_HISTORY view, you can get the value for the SQL_TEXT column from the V$SQLAREA view, as shown here:

```
SQL> select sql_text FROM v$sqlarea WHERE sql_id = '7zfmhtu327zm0';
```

Once you have the SQL_ID, it's also easy to extract the SQL Plan for this SQL statement, by executing the following query based on the DBMS_XPLAN package:

```
SQL> select * FROM table(dbms_xplan.display_awr('7zfmhtu327zm0'));
```

The background process MMON flushes ASH data to disk every hour, when the AWR snapshot is created. What happens when MMON flushes ASH data to disk? Well, you won't be able to query older data any longer with the V$ACTIVE_SESSION_HISTORY view. Not to worry, because you can still use the DBA_HIST_ACTIVE_SESS_HISTORY view to query the older data. The structure of this view is similar to that of the V$ACTIVE_SESSION_HISTORY view. The DBA_HIST_ACTIVE_SESS_HISTORY view shows the history of the contents of the in-memory active session history of recent system activity. You can also query the V$SESSION_WAIT_HISTORY view to examine the last ten wait events for a session, while it's still active. This view offers more reliable information for very recent wait events than the V$SESSION and V$SESSION_WAIT views, both of which show wait information for only the most recent wait. Here's a typical query using the V$SESSION_WAIT_HISTORY view.

```
SQL> select sid from v$session_wait_history
     where wait_time = (select max(wait_time) from v$session_wait_history);
```

Any non-zero values under the WAIT_TIME column represent the time waited by this session for the last wait event. A zero value for this column means that the session is waiting currently for a wait event.

5-18. Analyzing Recent Wait Events in a Database

Problem

You'd like to find out the most important waits in your database in the recent past, as well as the users, SQL statements, and objects that are responsible for most of those waits.

Solution

Query the V$ACTIVE_SESSION_HISTORY view to get information about the most common wait events, and the SQL statements, database objects, and users responsible for those waits. The following are some useful queries you can use.

To find the most important wait events in the last 15 minutes, issue the following query:

```
SQL> select event,
     sum(wait_time +
     time_waited) total_wait_time
     from v$active_session_history
     where sample_time between
     sysdate - 30/2880 and sysdate
     group by event
     order by total_wait_time desc
```

To find out which of your users experienced the most waits in the past 15 minutes, issue the following query:

```
SQL> select s.sid, s.username,
     sum(a.wait_time +
     a.time_waited) total_wait_time
     from v$active_session_history a,
     v$session s
     where a.sample_time between sysdate - 30/2880 and sysdate
     and a.session_id=s.sid
     group by s.sid, s.username
     order by total_wait_time desc;
```

Execute the following query to find out the objects with the highest waits.

```
SQL>select a.current_obj#, o.object_name, o.object_type, a.event,
    sum(a.wait_time +
    a.time_waited) total_wait_time
    from v$active_session_history a,
    dba_objects d
    where a.sample_time between sysdate - 30/2880 and sysdate
```

```
    and a.current_obj# = d.object_id
    group by a.current_obj#, d.object_name, d.object_type, a.event
    order by total_wait_time;
```

You can identify the SQL statements that have been waiting the most during the last 15 minutes with this query.

```
SQL> select a.user_id,u.username,s.sql_text,
     sum(a.wait_time + a.time_waited) total_wait_time
     from v$active_session_history a,
     v$sqlarea s,
     dba_users u
     where a.sample_time between sysdate - 30/2880 and sysdate
     and a.sql_id = s.sql_id
     and a.user_id = u.user_id
     group by a.user_id,s.sql_text, u.username;
```

How It Works

The "Solution" section shows how to join the `V$ACTIVE_SESSION_HISTORY` view with other views, such as the `V$SESSION`, `V$SQLAREA`, `DBA_USERS`, and `DBA_OBJECTS` views, to find out exactly what's causing the highest number of wait events or who's waiting the most, in the past few minutes. This information is extremely valuable when troubleshooting "live" database performance issues.

5-19. Identifying Time Spent Waiting Due to Locking

Problem

You want to identify the total time spent waiting by sessions due to locking issues.

Solution

You can use the following query to identify (and quantify) waits caused by locking of a table's rows. Since the query orders the wait events by time waited, you can quickly see which type of wait events accounts for most of the waits in your instance.

```
SQL> select wait_class, event, time_waited / 100 time_secs
  2  from v$system_event e
  3  where e.wait_class <> 'Idle' AND time_waited > 0
  4  union
  5  select 'Time Model', stat_name NAME,
  6  round ((value / 1000000), 2) time_secs
  7  from v$sys_time_model
  8  where stat_name NOT IN ('background elapsed time', 'background cpu time')
  9*  order by 3 desc;
```

```
WAIT_CLASS                      EVENT                        TIME_SECS
------------------------  ----------------------------   --------------
System I/O                log file parallel write              45066.32
System I/O                control file sequential read         23254.41
Time Model                DB time                              11083.91
Time Model                sql execute elapsed time              7660.04
Concurrency               latch: shared pool                    5928.73
Application               enq: TX - row lock contention         3182.06
…
SQL>
```

In this example, the wait event `enq: TX - row lock contention` reveals the total time due to row lock enqueue wait events. Note that the shared pool latch events are classified under the Concurrency wait class, while the enqueue `TX - row lock contention` event is classified as an Application class wait event.

How It Works

The query in the "Solution" section joins the `V$SYSTEM_EVENT` and the `V$SYS_TIME_MODEL` views to show you the total time waited due to various wait events. In our case, we're interested in the total time waited due to enqueue locking. If you're interested in the total time waited by a specific session, you can use a couple of different `V$` views to find out how long sessions have been in a wait state, but we recommend using the `V$SESSION` view, because it shows you various useful attributes of the blocking and blocked sessions. Here's an example showing how to find out how long a session has been blocked by another session.

```
SQL>select sid, username, event, blocking_session,
    seconds_in_wait, wait_time
    from v$session where state in ('WAITING');
```

The query reveals the following about the session with SID 81, which is in a `WAITING` state:

```
SID  : 81 (this is the blocked session)
username: SH (user who's being blocked right now)
event: TX - row lock contention (shows the exact type of lock contention)
blocking session: 68 (this is the "blocker")
seconds_in_wait: 3692 (how long the blocked session is in this state)
```

The query reveals that the user SH, with a SID of 81, has been blocked for almost an hour (3,692 seconds). User SH is shown as waiting for a lock on a table that is currently locked by session 68. While the `V$SESSION` view is highly useful for identifying the blocking and blocked sessions, it can't tell you the SQL statement that's involved in the blocking of the table. Often, identifying the SQL statement that's involved in a blocking situation helps in finding out exactly why the statement is leading to the locking behavior. To find out the actual SQL statement that's involved, you must join the `V$SESSION` and the `V$SQL` views, as shown here.

```
SQL> select sid, sql_text
     from v$session s, v$sql q
     where sid in (68,81)
     and (
     q.sql_id = s.sql_id or  q.sql_id = s.prev_sql_id)
SQL> /
```

```
     SID                      SQL_TEXT
-------------     ----------------------------------------------------
     68           select * from test for update
     81           update hr.test set name='nalapati' where user_id=1111

SQL>
```

The output of the query shows that session 81 is being blocked because it's trying to update a row in a table that has been locked by session 68, using the `SELECT … FOR UPDATE` statement. In cases such as this, if you find a long queue of user sessions being blocked by another session, you must kill the blocking session so the other sessions can process their work. You'll also see a high active user count in the database during these situations—killing the blocking session offers you an immediate solution to resolving contention caused by enqueue locks. Later on, you can investigate why the blocks are occurring, so as to prevent these situations.

For any session, you can identify the total time waited by a session for each wait class, by issuing the following query:

```
SQL> select wait_class_id, wait_class,
     total_waits, time_waited
     from v$session_wait_class
     where sid = <SID>;
```

If you find, for example, that this session endured a very high number of waits in the application wait class (wait class ID for this class is 4217450380), you can issue the following query using the `V$SYSTEM_EVENT` view, to find out exactly which waits are responsible:

```
SQL> select event, total_waits, time_waited
     from v$system_event e, v$event_name n
     where n.event_id = e.event_id
     and e.wait_class_id = 4217450380;

EVENT                                TOTAL_WAITS             TIME_WAITED
----------------------        ----------------------     --------------------

enq: TM - contention                              82                     475
…
SQL>
```

In our example, the waits in the application class (ID 4217450380) are due to locking contention as revealed by the wait event `enq:TM - contention`. You can further use the `V$EVENT_HISTOGRAM` view, to find out how many times and for how long sessions have waited for a specific wait event since you started the instance. Here's the query you need to execute to find out the wait time pattern for enqueue lock waits:

```
SQL> select wait_time_milli bucket, wait_count
     from v$event_histogram
     where event = 'enq: TX - row lock contention';
```

A high amount of enqueue waits due to locking behavior is usually due to faulty application design. You'll sometimes encounter this when an application executes many updates against the same row or a set of rows. Since this type of high waits due to locking is due to inappropriately designed applications, there's not much you can do by yourself to reduce these waits. Let your application team know why these waits are occurring, and ask them to consider modifying the application logic to avoid the waits.

Any of the following four DML statements can cause locking contention: `INSERT`, `UPDATE`, `DELETE`, and `SELECT FOR UPDATE`. `INSERT` statements wait for a lock because another session is attempting to insert a row with an identical value. This usually happens when you have a table that has a primary key or unique constraint, with the application generating the keys. Use an Oracle sequence instead to generate the key values, to avoid these types of locking situations. You can specify the `NOWAIT` option with a `SELECT FOR UPDATE` statement to eliminate session blocking due to locks. You can also use the `SELECT FOR UPDATE NOWAIT` statement to avoid waiting by sessions for locks when they issue an `UPDATE` or `DELETE` statement. The `SELECT FOR UPDATE NOWAIT` statement locks the row without waiting.

5-20. Minimizing Latch Contention

Problem

You're seeing a high number of latch waits, and you'd like to reduce the latch contention.

Solution

Severe latch contention can slow your database down noticeably. When you're dealing with a latch contention issue, start by executing the following query to find out the specific types of latches and the total wait time caused by each wait.

```
SQL> select event, sum(P3), sum(seconds_in_wait) seconds_in_wait
     from v$session_wait
     where event like 'latch%'
     group by event;
```

The previous query shows the latches that are currently being waited for by this session. To find out the amount of time the entire instance has waited for various latches, execute the following SQL statement.

```
SQL> select wait_class, event, time_waited / 100 time_secs
     from v$system_event e
     where e.wait_class <> 'Idle' AND time_waited > 0
     union
     select 'Time Model', stat_name NAME,
     round ((value / 1000000), 2) time_secs
     from v$sys_time_model
     where stat_name not in ('background elapsed time', 'background cpu time')
     order by 3 desc;

WAIT_CLASS                          EVENT                     TIME_SECS
-------------------         ---------------------------      -----------------
Concurrency                 library cache pin                        622.24
Concurrency                 latch: library cache                     428.23
Concurrency                 latch: library cache lock                 93.24
Concurrency                 library cache lock                        24.20
Concurrency                 latch: library cache pin                  60.28

…
```

The partial output from the query shows the latch-related wait events, which are part of the Concurrency wait class.

You can also view the top five wait events in the AWR report to see if lache contention is an issue, as shown here:

```
Event                          Waits          Time (s)     (ms)      Time      Wait Class
-----------------------   ------------    ----------    -------   --------   ------------
db file sequential read      42,005,780       232,838          6       73.8      User I/O
CPU time                                      124,672                  39.5      Other
latch free                   11,592,952        76,746          7       24.3      Other
wait list latch free            107,553         2,088         19        0.7      Other
latch: library cache          1,135,976         1,862          2        0.6      Concurrency
```

Here are the most common Oracle latch wait types and how you can reduce them.

> Shared pool and library latches: These are caused mostly by the database repeatedly executing the same SQL statement that varies slightly each time. For example, a database may execute a SQL statement 10,000 times, each time with a different value for a variable. The solution in all such cases is to use bind variables. An application that explicitly closes all cursors after each execution may also contribute to this type of wait. The solution for this is to specify the CURSOR_SPACE_FOR_TIME initialization parameter. Too small a shared pool may also contribute to the latch problem, so check your SGA size.
>
> Cache buffers LRU chain: These latch events are usually due to excessive buffer cache usage and may be caused both by excessive physical reads as well as logical reads. Either the database is performing large full table scans, or it's performing large index range scans. The usual cause for these types of latch waits is either the lack of an index or the presence of an unselective index. Also check to see if you need to increase the size of the buffer cache.
>
> Cache buffer chains: These waits are due to one or more hot blocks that are being repeatedly accessed. Application code that updates a table's rows to generate sequence numbers, rather than using an Oracle sequence, can result in such hot blocks. You might also see the cache buffer chains wait event when too many processes are scanning an unselective index with similar predicates.

Also, if you're using Oracle sequences, re-create them with a larger cache size setting and try to avoid using the ORDER clause. The CACHE clause for a sequence determines the number of sequence values the database must cache in the SGA. If your database is processing a large number of inserts and updates, consider increasing the cache size to avoid contention for sequence values. By default, the cache is set to 20 values. Contention can result if values are being requested fast enough to frequently deplete the cache. If you're dealing with a RAC environment, using the NOORDER clause will prevent enqueue contention due to the forced ordering of queued sequence values.

How It Works

Oracle uses internal locks called latches to protect various memory structures. When a server process attempts to get a latch but fails to do so, that attempt is counted as a latch free wait event. Oracle doesn't group all latch waits into a single latch free wait event. Oracle does use a generic latch free wait event, but this is only for the minor latch-related wait events. For the latches that are most common, Oracle uses various subgroups of latch wait events, with the name of the wait event type. You can identify the

exact type of latch by looking at the latch event name. For example, the latch event `latch: library cache` indicates contention for library cache latches. Similarly, the `latch: cache buffer chains` event indicates contention for the buffer cache.

Oracle uses various types of latches to prevent multiple sessions from updating the same area of the SGA. Various database operations require sessions to read or update the SGA. For example, when a session reads a data block into the SGA from disk, it must modify the buffer cache least recently used chain. Similarly, when the database parses a SQL statement, that statement has to be added to the library cache component of the SGA. Oracle uses latches to prevent database operations from stepping on each other and corrupting the SGA.

A database operation needs to acquire and hold a latch for very brief periods, typically lasting a few nanoseconds. If a session fails to acquire a latch at first because the latch is already in use, the session will try a few times before going to "sleep." The session will re-awaken and try a few more times, before going into the sleep mode again if it still can't acquire the latch it needs. Each time the session goes into the sleep mode, it stays longer there, thus increasing the time interval between subsequent attempts to acquire a latch. Thus, if there's a severe contention for latches in your database, it results in a severe degradation of response times and throughput.

Don't be surprised to see latch contention even in a well-designed database running on very fast hardware. Some amount of latch contention, especially the cache buffers chain latch events, is pretty much unavoidable. You should be concerned only if the latch waits are extremely high and are slowing down database performance.

Contention due to the library cache latches as well as shared pool latches is usually due to applications not using bind variables. If your application can't be recoded to incorporate bind variables, all's not lost. You can set the `CURSOR_SHARING` parameter to force Oracle to use bind variables, even if your application hasn't specified them in the code. You can choose between a setting of `FORCE` or `SIMILAR` for this parameter to force the substituting of bind variables for hard-coded values of variables. The default setting for this parameter is `EXACT`, which means that the database won't substitute bind variables for literal values. When you set the `CURSOR_SHARING` parameter to `FORCE`, Oracle converts all literals to bind variables. The `SIMILAR` setting causes a statement to use bind variables only if doing so doesn't change a statement's execution plan. Thus, the `SIMILAR` setting seems a safer way to go about forcing the database to use bind variables instead of literals. Although there are some concerns about the safety of setting the `CURSOR_SHARING` parameter to `FORCE`, we haven't seen any real issues with using this setting. The library cache contention usually disappears once you set the `CURSOR_SHARING` parameter to `FORCE` or to `SIMILAR`. The `CURSOR_SHARING` parameter is one of the few Oracle silver bullets that'll improve database performance immediately by eliminating latch contention. Use it with confidence when dealing with library cache latch contention.

The cache buffer chains latch contention is usually due to a session repeatedly reading the same data blocks. First identify the SQL statement that's responsible for the highest amount of the cache buffers chain latches and see if you can tune it. If this doesn't reduce the latch contention, you must identify the actual hot blocks that are being repeatedly read.

If a hot block belongs to an index segment, you may consider partitioning the table and using local indexes. For example, a hash partitioning scheme lets you spread the load among multiple partitioned indexes. You can also consider converting the table to a hash cluster based on the indexed columns. This way, you can avoid the index altogether. If the hot blocks belong to a table segment instead, you can still consider partitioning the table to spread the load across the partitions. You may also want to reconsider the application design to see why the same blocks are being repeatedly accessed, thus rendering them "hot."

5-21. Managing Locks from Oracle Enterprise Manager

Problem

You'd like to find out how to handle locking issues through the Oracle Enterprise Manager Database Control GUI interface.

Solution

Instead of issuing multiple SQL queries to identify quickly disappearing locking events, you can use Oracle Enterprise Manager (OEM) DB Control to identify and resolve locking situations. You can find all current locks in the instance, including the blocking and the blocked sessions—you can can also kill the blocking session from OEM.

Here are the ways you can manage locking issues through OEM:

- In the Home page of DB Control, you'll see locking information in the Alerts table. Look for the User Block category to identify blocking sessions. The alert name you must look for is Blocking Session Count. Clicking the message link shown for this alert, such as "Session 68 is blocking 12 other sessions," for example, will take you to the Blocking Session Count page. In the Alert History table on this page, you can view details about the blocking and blocked sessions.

  Also in the Home page, under Related Alerts, you'll find the ADDM Performance table. Locking issues are revealed by the presence of the Row Lock Waits link. Click the Row Lock Waits link to go to the Row Lock Waits page. This page, shown in Figure 5-1, lets you view you all the SQL statements that were found waiting for row locks.

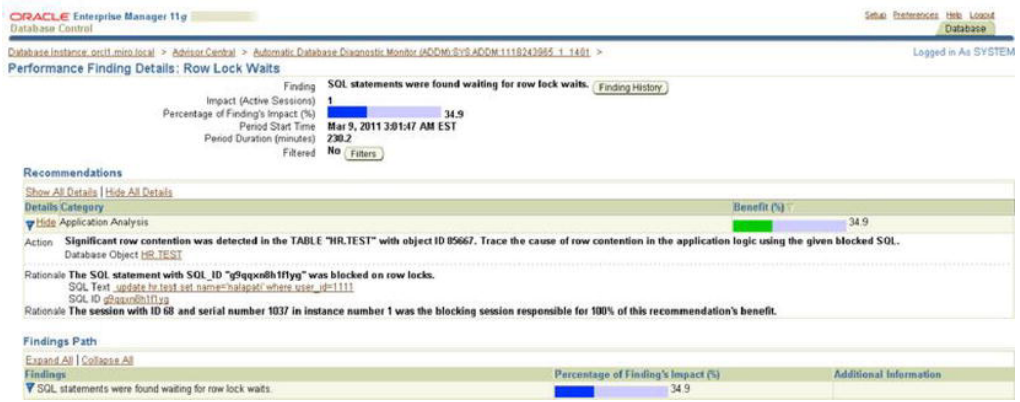

Figure 5-1. *The Row Lock Waits page in OEM*

- You can also view blocking session details by clicking the Performance tab in the Home page. Click Blocking Sessions under the Additional Monitoring Links section to go to the Blocking Sessions page. The Blocking Sessions page contains details for both the blocking as well as the blocked sessions. You can see the exact wait event, which will be `enq: TX row lock contention` when one session blocks another. You can find out the exact SQL statement that's involved in blocking sessions, by clicking the SQL ID link on this page. You can kill the blocking session from this page by clicking the Kill Session button at the top left side of the page.
- Also in the Additional Monitoring Links section is another link named Instance Locks, which takes you to the Instance Locks page. The Instance Locks page shows the session details for both the blocking and blocked sessions. You can click the SQL ID link to view the current SQL that's being executed by the blocker and the blocked sessions. You can also find out the name of the object that's locked. You can kill the blocking session by clicking the Kill Session button.

How It Works

You don't necessarily have to execute multiple SQL scripts to analyze locking behavior in your database. The SQL code we showed you earlier in various recipes was meant to explain how Oracle locking works. On a day-to-day basis, it's much more practical and efficient to just use OEM to quickly find out who's blocking a session and why.

5-22. Analyzing Waits from Oracle Enterprise Manager

Problem

You'd like to use Oracle Enterprise Manager to manage waits in your database instances.

Solution

The OEM interface lets you quickly analyze current waits in your database, instead of running SQL scripts to do so. In the Home page, the Active Sessions graph shows the relative amounts of waits, I/O, and CPU. Click the Waits link in this graph to view the Active Sessions graph. To the right of the graph, you'll see various links such as Concurrency, Application, Cluster, Administrative, User I/O, etc. Clicking each of these links will take you to a page that shows you all active sessions that are waiting for waits under that wait class. We summarize the wait events under the most important of these wait classes here.

`User I/O`: This shows wait events such as `db file scattered read`, `db file sequential read`, `direct path read`, `direct path write`, and `read by other session`. You can click any of the links for the various waits to get a graph of the wait events. For example, clicking the "db file scattered read" link will take you to the histogram for the "Wait Event: db file scattered read" page.

`System I/O`: This shows waits due to the `db file parallel write`, `log file parallel write`, `control file parallel write`, and the `control file sequential read wait` events.

`Application`: This shows active sessions waiting for events such as enqueue locks.

How It Works

Once you understand the theory behind the Oracle Wait Interface, you can use OEM to quickly analyze current wait events in your database. You can find out not only which wait events are adversely affecting performance, but also which SQL statement and which users are involved. All the details pages you can drill down to from the Active Session page show a graph of the particular wait event class from the time the instance started. The pages also contain tables named Top SQL and Top Users, which show exactly which SQL and users are affected by the wait event.

CHAPTER 6

Analyzing Operating System Performance

Solving database performance issues sometimes requires the use of operating system (OS) utilities. These tools often provide information that can help isolate database performance problems. Consider the following situations:

- You're running multiple databases and multiple applications on one server and want to use OS utilities to identify which database (and corresponding process) is consuming the most operating system resources. This approach is invaluable when one database application is consuming resources to the point of causing other databases on the box to perform poorly.
- You need to verify if the database server is adequately sized for current application workload in terms of CPU, memory, disk I/O, and network bandwidth.
- An analysis is needed to determine at what point the server will not be able to handle larger (future) workloads.
- You've used database tools to identify system bottlenecks and want to double-check the analysis via operating system tools.

In these scenarios, to effectively analyze, tune, and troubleshoot, you'll need to employ OS tools to identify resource-intensive processes. Furthermore, if you have multiple databases and applications running on one server, when troubleshooting performance issues, it's often more efficient to first determine which database and process is consuming the most resources. Operating system utilities help pinpoint whether the bottleneck is CPU, memory, disk I/O, or a network issue. In Linux/Unix environments, once you have the operating system identifier, you can then query the database to show any corresponding database processes and SQL statements.

Take a look at Figure 6-1. This flowchart details the decision-making process and the relevant Linux/Unix operating system tools that a DBA steps through when diagnosing sluggish server performance. For example, when you're dealing with performance problems, one common first task is to log on to the box and quickly check for disk space issues using OS utilities like `df` and `du`. A full mount point is a common cause of database unavailability.

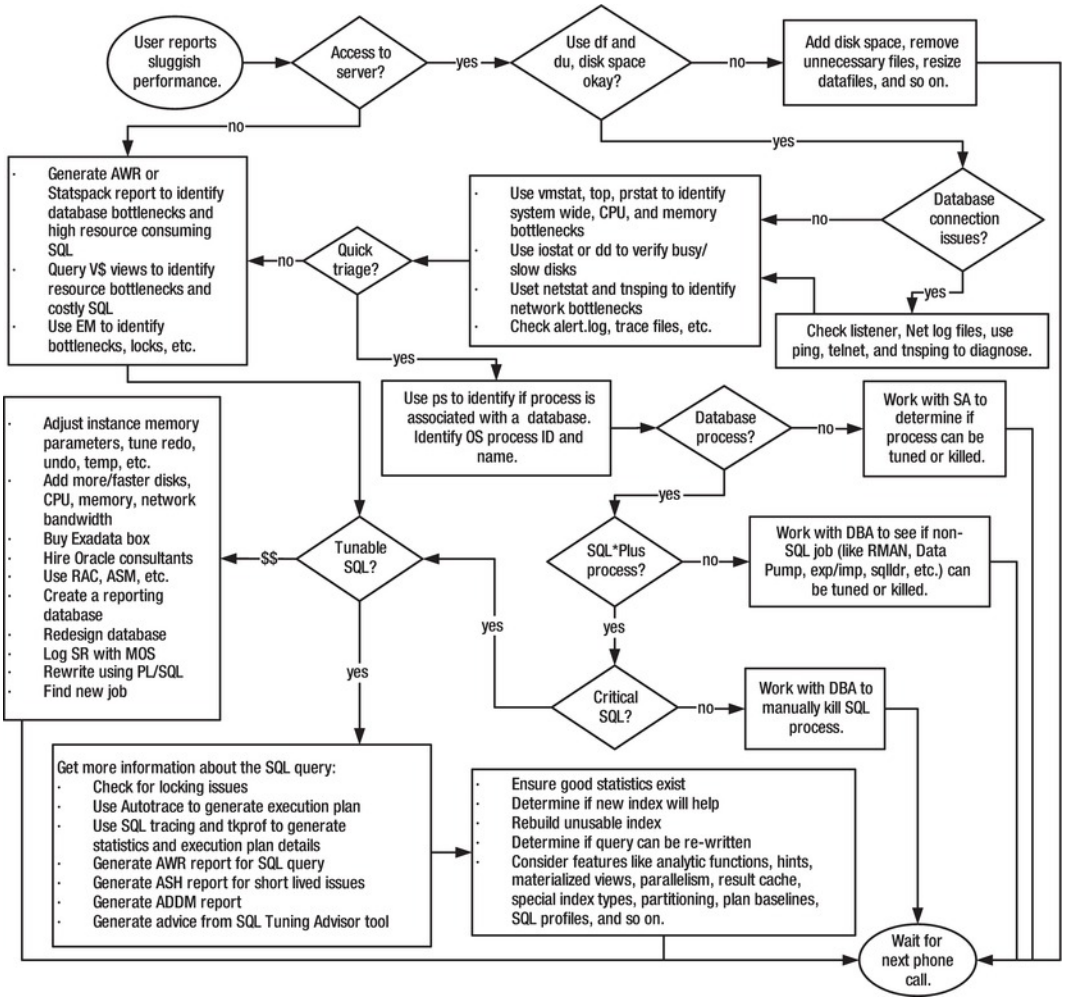

Figure 6-1. *Troubleshooting poor performance*

After inspecting disk space issues, the next task is to use an OS utility such as `vmstat`, `top`, or `ps` to determine what type of bottleneck you have. For example, is sluggish performance related to a disk I/O issue, CPU, memory, or the network? After determining the type of bottleneck, the next step is to determine if a database process is causing the bottleneck.

The `ps` command is useful for displaying the process name and ID of the resource-consuming session. When you have multiple databases running on one box, you can determine which database is associated with the process from the process name. Once you have the process ID and associated database, you can then log on to the database and run SQL queries to determine if the process is associated with a SQL query. If the problem is SQL-related, then you can identify further details regarding the SQL query and where it might be tuned.

Figure 6-1 encapsulates the difficulty of troubleshooting performance problems. Correctly pinpointing the cause of performance issues and recommending an efficient solution is often easier said than done. When trying to resolve issues, some paths result in relatively efficient and inexpensive solutions, such as terminating a runaway operating system process or regenerating fresh statistics. Other decisions may lead you to conclude that you need to add expensive hardware or redesign the system. Your performance tuning conclusions can have long-lasting financial impact on your company and thus influence your ability to retain a job. Obviously you want to focus on the cause of a performance problem and not just address the symptom. If you can consistently identify the root cause of the performance issue and recommend an effective and inexpensive solution, this will greatly improve your employment opportunities.

The focus of this chapter is to provide detailed examples that show how to use Linux/Unix operating system utilities to identify server performance issues. These utilities are invaluable for providing extra information used to diagnose performance issues outside of tools available within the database. Operating system utilities act as an extra set of eyes to help zero in on the cause of poor database performance.

6-1. Detecting Disk Space Issues

Problem

Users are reporting that they can't connect to a database. You log on to the database server, attempt to connect to SQL*Plus, and receive this error:

```
ORA-09817: Write to audit file failed.
Linux Error: 28: No space left on device
Additional information: 12
```

You want to quickly determine if a mount point is full and where the largest files are within this mount point.

Solution

In a Linux/Unix environment, use the `df` command to identify disk space issues. This example uses the `-h` to format the output so that space is reported in megabytes or gigabytes:

```
$ df –h
```

Here is some sample output:

```
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                       29G   28G     0 100% /
/dev/sda1              99M   19M   75M  20% /boot
```

The prior output indicates that the root (`/`) file system is full on this server. In this situation, once a full mount point is identified, then use the `find` command to locate the largest files contained in a directory structure. This example navigates to the `ORACLE_HOME` directory and then connects the `find`, `ls`, `sort`, and `head` commands to identify the largest files beneath that directory:

```
$ cd $ORACLE_HOME
$ find . -ls | sort -nrk7 | head -10
```

If you have a full mount point, also consider looking for the following types of files that can be moved or removed:

- Deleting database trace files
- Removing large Oracle Net log files
- Moving, compressing, or deleting old archive redo log files
- Removing old installation files or binaries
- If you have datafiles with ample free space, consider resizing them to smaller sizes

Another way to identify where the disk space is being used is to find the largest space-consuming directories beneath a given directory. This example combines the `du`, `sort`, and `head` commands to show the ten largest directories beneath the current working directory:

```
$ du -S . | sort -nr | head -10
```

The prior command is particularly useful for identifying a directory that might not necessarily have large files in it, but lots of small files consuming space (like trace files).

■ **Note** On Solaris Unix systems, the prior command will need to use `du` with the `-o` option.

How It Works

When you have a database that is hung because there is little or no free disk space, you should quickly find files that can be safely removed without compromising database availability. On Linux/Unix servers, the `df`, `find`, and `du` commands are particularly useful.

When working with production database servers, it's highly desirable to proactively monitor disk space so that you're warned about a mount point becoming full. Listed next is a simple shell script that monitors disk space for a given set of mount points:

```
#!/bin/bash
mntlist="/orahome /oraredo1 /oraarch1 /ora01 /oradump01 /"
for ml in $mntlist
do
echo $ml
usedSpc=$(df -h $ml | awk '{print $5}' | grep -v capacity | cut -d "%" -f1 -)
BOX=$(uname -a | awk '{print $2}')
#
case $usedSpc in
[0-9])
arcStat="relax, lots of disk space: $usedSpc"
;;
[1-7][0-9])
arcStat="disk space okay: $usedSpc"
```

```
;;
[8][0-9])
arcStat="space getting low: $usedSpc"
;;
[9][0-9])
arcStat="warning, running out of space: $usedSpc"
echo $arcStat $ml | mailx -s "space on: $BOX" dkuhn@oracle.com
;;
[1][0][0])
arcStat="update resume, no space left: $usedSpc"
echo $arcStat $ml | mailx -s "space on: $BOX" dkuhn@oracle.com
;;
*)
arcStat="huh?: $usedSpc"
esac
#
BOX=$(uname -a | awk '{print $2}')
echo $arcStat
#
done
#
exit 0
```

You'll have to modify the script to match your environment. For example, the second line of the script specifies the mount points on the box being monitored:

```
mntlist="/orahome /oraredo1 /oraarch1 /ora01 /oradump01 /"
```

These mount points should match the mount points listed in the output of the `df -h` command. For a Solaris box that this script runs on, here's the output of `df`:

```
Filesystem             size   used  avail capacity  Mounted on
/                       35G   5.9G    30G    17%    /
/ora01                 230G   185G    45G    81%    /ora01
/oraarch1              100G    12G    88G    13%    /oraarch1
/oradump01             300G    56G   244G    19%    /oradump01
/orahome                20G    15G   5.4G    73%    /orahome
/oraredo1               30G   4.9G    25G    17%    /oraredo1
```

Also, depending on what version of Linux/Unix you're using, you'll have to modify this line as well:

```
usedSpc=$(df -h $ml | awk '{print $5}' | grep -v capacity | cut -d "%" -f1 -)
```

The prior line of code depends on the output of the `df` command, which can vary somewhat depending on the operating system vendor and version. For example, on one Linux system, the output of `df` might span two lines and reports on `Use%` instead of `capacity`, so in this scenario, the `usedSpc` variable is populated as shown:

```
usedSpc=$(for x in `df -h $ml | grep -v "Use%"` ; do echo $x ; done | \
grep "%" |  cut -d "%" -f1 -)
```

The prior code (broken into two lines to fit on the page) runs several Linux/Unix commands and places the output in the `usedSpc` variable. The command first runs `df -h`, which is piped to the `awk` command. The `awk` command takes the output and prints out the fifth column. This is piped to the `grep`

command, which uses -v to eliminate the word Use% from the output. This is finally piped to the cut command, which cuts out the "%" character from the output.

On a Linux/Unix system, a shell script such as the prior one can easily be run from a scheduling utility such as cron. For example, if the shell script is named filesp.bsh, here is a sample cron entry:

```
#-----------------------------------------------------------------
# Filesystem check
7 * * * * /orahome/oracle/bin/filesp.bsh 1>/orahome/oracle/bin/log/filesp.log 2>&1
#-----------------------------------------------------------------
```

The prior entry instructs the system to run the filesp.bsh shell script at seven minutes after the hour for every hour of the day.

6-2. Identifying System Bottlenecks (vmstat)

Problem

You want to determine if a server performance issue is specifically related to disk I/O, CPU, memory, or network.

■ **Note** If you are running under Solaris, see Recipe 6-3 for a specific solution applying to that operating system.

Solution

Use vmstat to determine where the system is resource-constrained. For example, the following command reports on system resource usage every five seconds on a Linux system:

```
$ vmstat 5
```

Here is some sample output:

```
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
2  0 228816 2036164  78604 3163452   0    0     1   16    0    0 29  0 70  0  0
 2  0 228816 2035792  78612 3163456   0    0     0   59  398  528 50  1 49  0  0
 2  0 228816 2035172  78620 3163448   0    0     0   39  437  561 50  1 49  0  0
```

To exit out of vmstat in this mode, press Ctrl+C. You can also have vmstat report for a specific number of runs. For example, this instructs vmstat to run every six seconds for a total of ten reports:

```
$ vmstat 6 10
```

Here are some general heuristics you can use when interpreting the output of `vmstat`:

- If the `wa` (time waiting for I/O) column is high, this is usually an indication that the storage subsystem is overloaded. See Recipe 6-6 for identifying the sources of I/O contention.
- If `b` (processes sleeping) is consistently greater than 0, then you may not have enough CPU processing power. See Recipes 6-5 and 6-9 for identifying Oracle processes and SQL statements consuming the most CPU.
- If `so` (memory swapped out to disk) and `si` (memory swapped in from disk) are consistently greater than 0, you may have a memory bottleneck. See Recipe 6-5 for details on identifying Oracle processes and SQL statements consuming the most memory.

How It Works

The `vmstat` (virtual memory statistics) tool helps quickly identify bottlenecks on your server. Use the output of `vmstat` to help determine if the performance bottleneck is related to CPU, memory, or disk I/O. Table 6-1 describes the columns available in the output of `vmstat`. These columns may vary somewhat depending on your operating system and version.

Table 6-1. *Descriptions of* `vmstat` *Output Columns*

Column	Description
`r`	Number of processes waiting for run time
`b`	Number of processes in uninterruptible sleep
`swpd`	Amount of virtual memory
`free`	Amount of idle memory
`buff`	Amount of buffer memory
`cache`	Amount of cache memory
`inact`	Amount of inactive memory (-a option)
`active`	Amount of active memory (-a option)
`si`	Amount of memory swapped from disk/second
`so`	Amount of memory swapped to disk/second
`bi`	Blocks read/second from disk

Continued

Column	Description
bo	Blocks written/second to disk
in	Number of interrupts/seconds
cs	Number of context switches/second
us	CPU time running non-kernel code
sy	CPU time running kernel code
Id	CPU time idle
wa	CPU time waiting for I/O
st	CPU time taken from virtual machine

OS WATCHER

Oracle provides a collection of operating system scripts that gather and store metrics for CPU, memory, disk I/O, and network usage. The OS Watcher tool suite automates the gathering of statistics using tools such as `top`, `vmstat`, `iostat`, `mpstat`, `netstat`, and so on.

You can obtain OS Watcher from the My Oracle Support web site (support.oracle.com). Navigate to the support web site and search for OS Watcher. The OS Watcher User Guide can be found under document ID **301137.1.** This tool is supported on most Linux/Unix systems, and there is also a version for the Windows platform.

6-3. Identifying System Bottlenecks (Solaris)

Problem

You're working on a Solaris system, and irate users are reporting the database application is slow. You have multiple databases running on this box and want to identify which processes are consuming the most CPU resources. Once the resource-consuming processes are identified at the OS, then you want to map them (if possible) to a database process.

■ **Note** If you are not running Solaris, then see the solution in Recipe 6-2.

Solution

On most Solaris systems, the `prstat` utility is used to identify which processes are consuming the most CPU resources. For example, you can instruct the `prstat` to report system statistics every five seconds:

```
$ prstat 5
```

Here is some sample output:

```
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 16609 oracle   2364M 1443M cpu2    60    0   3:14:45  20% oracle/11
 27565 oracle   2367M 1590M cpu3    21    0   0:11:28  16% oracle/14
 23632 oracle   2284M 1506M run     46    2   0:16:18 6.1% oracle/11
  4066 oracle   2270M 1492M sleep   59    0   0:02:52 1.7% oracle/35
 15630 oracle   2274M 1482M sleep   48    0  19:40:41 1.2% oracle/11
```

Type **q** or press Ctrl+C to exit `prstat`. In the prior output, process 16609 is consistently showing up as a top CPU-consuming process.

After identifying a top resource-consuming process, you can determine which database the process is associated with by using the `ps` command. This example reports on process information associated with the PID of 16609:

```
$ ps -ef | grep 16609
  oracle 16609  3021  18   Mar 09 ?         196:29 ora_dw00_ENGDEV
```

In this example, the name of the process is `ora_dw00_ENGDEV` and the associated database is `ENGDEV`.

How It Works

If you're working on a Solaris server, the `top` utility is oftentimes not installed. In these environments, the `prstat` command can be used to determine top resource-consuming processes on the system. Table 6-2 describes several of the columns displayed in the default output of `prstat`.

Table 6-2. *Column Descriptions of the* `top` *Output*

Column	Description
`PID`	Unique process identifier
`USERNAME`	OS username running the process
`SIZE`	Virtual memory size of the process
`RSS`	Resident set size of process
`STATE`	State of process (running, stopped, and so on)
`PRI`	Priority of process
`NICE`	Nice value used to compute priority
`TIME`	Cumulative execution time
`CPU`	Percent of CPU consumption
PROCESS	Name of the executed file
NLWP	Number of LWPs in the process

6-4. Identifying Top Server-Consuming Resources (top)

Problem

You have a Linux server that hosts multiple databases. Users are reporting sluggishness with an application that uses one of the databases. You want to identify which processes are consuming the most resources on the server and then determine if the top consuming process is associated with a database.

Solution

The `top` command shows a real-time display of the highest resource-consuming processes on a server. Here's the simplest way to run `top`:

```
$ top
```

Listed next is a fragment of the output:

```
top - 04:40:05 up 353 days, 15:16,  3 users,  load average: 2.84, 2.34, 2.45
Tasks: 454 total,   4 running, 450 sleeping,   0 stopped,   0 zombie
Cpu(s): 64.3%us,  3.4%sy,  0.0%ni, 20.6%id, 11.8%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   7645184k total,  6382956k used,  1262228k free,   176480k buffers
Swap:  4128760k total,      184k used,  4128576k free,  3953512k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19888 oracle    25   0  148m  13m  11m R 100.1  0.2 313371:45 oracle
19853 oracle    25   0  148m  13m  11m R 99.8  0.2 313375:41 oracle
 9722 oracle    18   0 1095m 287m 150m R 58.6  3.8   0:41.89 oracle
  445 root      11  -5     0    0    0 S  0.3  0.0   8:32.67 kjournald
 9667 oracle    15   0  954m  55m  50m S  0.3  0.7   0:01.03 oracle
    2 root      RT  -5     0    0    0 S  0.0  0.0   2:17.99 migration/0
```

Type **q** or press Ctrl+C to exit `top`. In the prior output, the first section of the output displays general system information such as how long the server has been running, number of users, CPU information, and so on. The second section shows which processes are consuming the most CPU resources (listed top to bottom). In the prior output, the process ID of 19888 is consuming a large amount of CPU. To determine which database this process is associated with, use the `ps` command:

```
$ ps 19888
```

Here is the associated output:

```
PID TTY      STAT   TIME COMMAND
19888 ?        Rs   313393:32 oracle011R2 (DESCRIPTION=(LOCAL=YES)
```

In the prior output, the fourth column displays the value of `oracle011R2`. This indicates that this is an Oracle process associated with the `011R2` database. If the process continues to consume resources, you can next determine if there is a SQL statement associated with the process (see Recipe 6-9) or terminate the process (see Recipe 6-10).

■ **Tip** If you work in a Solaris operating system environment, use the `prstat` command to view the top CPU-consuming processes (see Recipe 6-3 for details).

How It Works

If installed, the `top` utility is often the first investigative tool employed by DBAs and system administrators to identify resource-intensive processes on a server. If a process is continuously consuming excessive system resources, then you should further determine if the process is associated with a database and a specific SQL statement.

By default, `top` will repetitively refresh (every few seconds) information regarding the most CPU-intensive processes. While `top` is running, you can interactively change its output. For example, if you type **>**, this will move the column that `top` is sorting one position to the right. Table 6-3 lists the most useful hot key features to alter the `top` display to the desired format.

Table 6-3. *Commands to Interactively Change the* `top` *Output*

Command	Function
Spacebar	Immediately refreshes the output
< or >	Moves the sort column one to the left or to the right; by default, `top` sorts on the CPU column.
`d`	Changes the refresh time
`R`	Reverses the sort order
`z`	Toggles the color output
h	Displays help menu
`F` or `O`	Chooses a sort column

Table 6-4 describes several of the columns displayed by `top`. Use these descriptions to help interpret the output.

Table 6-4. *Column Descriptions of the* `top` *Output*

Column	Description
`PID`	Unique process identifier
`USER`	OS username running the process
`PR`	Priority of the process
`NI`	Nice value or process; negative value means high priority; positive value means low priority.
`VIRT`	Total virtual memory used by process
`RES`	Non-swapped physical memory used
`SHR`	Shared memory used by process
`S`	Process status
`%CPU`	Processes percent of CPU consumption since last screen refresh
`%MEM`	Percent of physical memory the process is consuming

Column	Description
TIME	Total CPU time used by process
TIME+	Total CPU time, showing hundredths of seconds
COMMAND	Command line used to start a process

6-5. Identifying CPU and Memory Bottlenecks (ps)

Problem

You want to quickly isolate which processes on the server are consuming the most CPU and memory resources.

Solution

The ps (process status) command is handy for quickly identifying top resource-consuming processes. For example, this command displays the top ten CPU-consuming resources on the box:

```
$ ps -e -o pcpu,pid,user,tty,args | sort -n -k 1 -r | head
```

Here is a partial listing of the output:

```
97.8 26902 oracle   ?        oracleO11R2 (DESCRIPTION=(LOCAL=YES)(ADDRESS=(PROTOCOL=beq)))
 0.5 27166 oracle   ?        ora_diag_O11R2
0.0     9 root     ?        [ksoftirqd/2]
```

In the prior output, the process named oracleO11R2 is consuming an inordinate amount of CPU resources on the server. The process name identifies this as an Oracle process associated with the O11R2 database.

Similarly, you can also display the top memory-consuming processes:

```
$ ps -e -o pmem,pid,user,tty,args | sort -n -k 1 -r | head
```

How It Works

The Linux/Unix ps command displays information about currently active processes on the server. The pcpu switch instructs the process status to report the CPU usage of each process. Similarly the pmem switch instructs ps to report on process memory usage. This gives you a quick and easy way to determine which processes are consuming the most resources.

When using multiple commands on one line (such as ps, sort, and head), it's often desirable to associate the combination of commands with a shortcut (alias). Here's an example of creating aliases:

```
$ alias topc='ps -e -o pcpu,pid,user,tty,args | sort -n -k 1 -r | head'
$ alias topm='ps -e -o pmem,pid,user,tty,args | sort -n -k 1 -r | head'
```

Now instead of typing in the long line of commands, you can use the alias—for example:

```
$ topc
```

Also consider establishing the aliases in a startup file (like `.bashrc` or `.profile`) so that the commands are automatically defined when you log on to the database server.

6-6. Identifying I/O Bottlenecks

Problem

You are experiencing performance problems and want to determine if the issues are related to slow disk I/O.

Solution

Use the `iostat` command with the `-x` (extended) option combined with the `-d` (device) option to generate I/O statistics. This next example displays extended device statistics every ten seconds:

```
$ iostat -xd 10
```

You need a fairly wide screen to view this output; here's a partial listing:

```
Device:    rrqm/s wrqm/s   r/s   w/s  rsec/s  wsec/s    rkB/s    wkB/s avgrq-sz
avgqu-sz   await  svctm  %util
sda          0.01   3.31  0.11  0.31    5.32   28.97     2.66    14.49    83.13
0.06  138.44   1.89   0.08
```

This periodic extended output allows you to view in real time which devices are experiencing spikes in read and write activity. To exit from the previous `iostat` command, press `Ctrl+C`. The options and output may vary depending on your operating system. For example, on some Linux/Unix distributions, the `iostat` output may report the disk utilization as `%b` (percent busy).

When trying to determine whether device I/O is the bottleneck, here are some general guidelines when examining the `iostat` output:

- Look for devices with abnormally high blocks read or written per second.
- If any device is near 100% utilization, that's a strong indicator I/O is a bottleneck.

How It Works

The `iostat` command can help you determine whether disk I/O is potentially a source of performance problems. Table 6-5 describes the columns displayed in the `iostat` output.

Table 6-5. *Column Descriptions of* `iostat` *Disk I/O Output*

Column	Description
Device	Device or partition name
`tps`	I/O transfers per second to the device
`Blk_read/s`	Blocks per second read from the device
`Blk_wrtn/s`	Blocks written per second to the device
`Blk_read`	Number of blocks read
`Blk_wrtn`	Number of blocks written
`rrqm/s`	Number of read requests merged per second that were queued to device
`wrqm/s`	Number of write requests merged per second that were queued to device
`r/s`	Read requests per second
`w/s`	Write requests per second
`rsec/s`	Sectors read per second
`wsec/s`	Sectors written per second
`rkB/s`	Kilobytes read per second
`wkB/s`	Kilobytes written per second
`avgrq-sz`	Average size of requests in sectors
`avgqu-sz`	Average queue length of requests
`await`	Average time in milliseconds for I/O requests sent to the device to be served
`svctm`	Average service time in milliseconds
`%util`	Percentage of CPU time during which I/O requests were issued to the device. Near 100% indicates device saturation

You can also instruct `iostat` to display reports at a specified interval. The first report displayed will report averages since the last server reboot; each subsequent report shows statistics since the previously generated snapshot. The following example displays a device statistic report every three seconds:

```
$ iostat -d 3
```

You can also specify a finite number of reports that you want generated. This is useful for gathering metrics to be analyzed over a period of time. This example instructs `iostat` to report every 2 seconds for a total of 15 reports:

```
$ iostat 2 15
```

When working with locally attached disks, the output of the `iostat` command will clearly show where the I/O is occurring. However, it is not that clear-cut in environments that use external arrays for storage. What you are presented with at the file system layer is some sort of a virtual disk that might also have been configured by a volume manager. In virtualized storage environments, you'll have to work with your system administrator or storage administrator to determine exactly which disks are experiencing high I/O activity.

Once you have determined that you have a disk I/O contention issue, then you can use utilities such as AWR (if licensed), Statspack (no license required), or the `V$` views to determine if your database is I/O stressed. For example, the AWR report contains an I/O statistics section with the following subsections:

- IOStat by Function summary
- IOStat by Filetype summary
- IOStat by Function/Filetype summary
- Tablespace IO Stats
- File IO Stats

You can also directly query data dictionary views such as `V$SQL` to determine which SQL statements are using excessive I/O—for example:

```
SELECT *
FROM
(SELECT
  parsing_schema_name
 ,direct_writes
 ,SUBSTR(sql_text,1,75)
 ,disk_reads
FROM v$sql
ORDER BY disk_reads DESC)
WHERE rownum < 20;
```

To determine which sessions are currently waiting for I/O resources, query `V$SESSION`:

```
SELECT
 username
,program
,machine
,sql_id
FROM v$session
WHERE event LIKE 'db file%read';
```

To view objects that are waiting for I/O resources, run a query such as this:

```
SELECT
 object_name
,object_type
,owner
FROM v$session   a
    ,dba_objects b
WHERE a.event LIKE 'db file%read'
AND   b.data_object_id = a.row_wait_obj#;
```

Once you have identified queries (using the prior queries in this section), then consider the following factors, which can cause a SQL statement to consume inordinate amounts of I/O:

- Poorly written SQL
- Improper indexing
- Improper use of parallelism (which can cause excessive full table scans)

You'll have to examine each query and try to determine if one of the prior items is the cause of poor performance as it relates to I/O.

6-7. Identifying Network-Intensive Processes

Problem

You're investigating performance issues on a database server. As part of your investigation, you want to determine if there are network bottlenecks on the system.

Solution

Use the `netstat` (network statistics) command to display network traffic. Perhaps the most useful way to view `netstat` output is with the `-ptc` options. These options display the process ID and TCP connections, and they continuously update the output:

```
$ netstat -ptc
```

Press Ctrl+C to exit the previous command. Here's a partial listing of the output:

```
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address  Foreign Address  State       PID/Program name
tcp        0      0 rmug.com:62386 rmug.com:1521    ESTABLISHED 22864/ora_pmon_RMDB
tcp        0      0 rmug.com:53930 rmug.com:1521    ESTABLISHED 6091/sqlplus
tcp        0      0 rmug.com:1521  rmug.com:53930   ESTABLISHED 6093/oracleRMDB1
tcp        0      0 rmug.com:1521  rmug.com:62386   ESTABLISHED 10718/tnslsnr
```

If the `Send-Q` (bytes not acknowledged by remote host) column has an unusually high value for a process, this may indicate an overloaded network. The useful aspect about the previous output is that you can determine the operating system process ID (PID) associated with a network connection. If you suspect the connection in question is an `oracle` session, you can use the techniques described in the "Solution" section of Recipe 6-9 to map an operating system PID to an Oracle process or SQL statement.

How It Works

When experiencing performance issues, usually the network is not the cause. Most likely you'll determine that bad performance is related to a poorly constructed SQL statement, inadequate disk I/O, or not enough CPU or memory resources. However, as a DBA, you need to be aware of all sources of performance bottlenecks and how to diagnose them. In today's highly interconnected world, you must possess network troubleshooting and monitoring skills. The `netstat` utility is a good starting place for monitoring server network connections.

6-8. Troubleshooting Database Network Connectivity

Problem

A user has reported that he or she can't connect to a database. You know there are many components involved with network connectivity and want to figure out the root cause of the problem.

Solution

Use these steps as guidelines when diagnosing Oracle database network connectivity issues:

1. Use the operating system `ping` utility to determine whether the remote box is accessible—for example:

   ```
   $ ping dwdb
   dwdb is alive
   ```

   If `ping` doesn't work, work with your system or network administrator to ensure you have server-to-server connectivity in place.

2. Use `telnet` to see if you can connect to the remote server and port (that the listener is listening on)—for example:

   ```
   $ telnet ora03 1521
   Trying 127.0.0.1...
   Connected to ora03.
   Escape character is '^]'.
   ```

   The prior output indicates that connectivity to a server and port is okay. If the prior command hangs, then contact your SA or network administrator for further assistance.

3. Use `tnsping` to determine whether Oracle Net is working. This utility will verify that an Oracle Net connection can be made to a database via the network—for example:

```
$ tnsping dwrep
.........
Used TNSNAMES adapter to resolve the alias
Attempting to contact (DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)
(HOST = dwdb1.us.farm.com)(PORT = 1521))
(CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = DWREP)))
OK (500 msec)
```

   If `tnsping` can't contact the remote database, verify that the remote listener and database are both up and running. On the remote box, use the `lsnrctl status` command to verify that the listener is up. Verify that the remote database is available by establishing a local connection as a non-`SYS` account (`SYS` can often connect to a troubled database when other schemas will not work).

4. Verify that the `TNS` information is correct. If the remote listener and database are working, then ensure that the mechanism for determining `TNS` information (like the `tnsnames.ora` file) contains the correct information.

   Sometimes the client machine will have multiple `TNS_ADMIN` locations and `tnsnames.ora` files. One way to verify whether a particular `tnsnames.ora` file is being used is to rename it and see whether you get a different error when attempting to connect to the remote database.

How It Works

Network connectivity issues can be troublesome to diagnose because there are several architectural components that have to be in place for it to work correctly. You need to have the following in place:

- A functional network
- Open ports from point to point
- Oracle Net correctly installed and configured
- Target database and listener up and running
- Correct navigational information from the client to the target database

If you're still having issues, examine the client `sqlnet.log` file and the remote server `listener.log` file. Sometimes these log files will show additional information that will pinpoint the issue.

6-9. Mapping a Resource-Intensive Process to a Database Process

Problem

It's a dark and stormy night, and the system is performing poorly. You identify an operating system–intensive process on the box. You want to map an operating system process back to a database process. If the database process is a SQL process, you want to display the user of the SQL statement and also the SQL.

Solution

In Linux/Unix environments, if you can identify the resource-intensive operating system process, then you can easily check to see if that process is associated with a database process. The process consists of the following:

1. Run an OS command to identify resource-intensive processes and associated IDs.
2. Identify the database associated with the process.
3. Extract details about the process from the database data dictionary views.
4. If it's a SQL statement, get those details.
5. Generate an execution plan for the SQL statement.

For example, suppose you identify the top CPU-consuming queries with the `ps` command:

```
$ ps -e -o pcpu,pid,user,tty,args|grep -i oracle|sort -n -k 1 -r|head
```

Here is some sample output:

```
16.4 11026   oracle ?       oracleDWREP (DESCRIPTION=(LOCAL=YES)(ADDRESS=(PROTOCOL=beq)))
 0.1  6448   oracle ?       oracleINVPRD (LOCAL=NO)
 0.5  3639   oracle ?       ora_dia0_STAGE
 0.4 28133   oracle ?       ora_dia0_DEVSEM
 0.4  4093   oracle ?       ora_dia0_DWODI
 0.4  3534   oracle ?       ora_dia0_ENGDEV
 0.2  4111   oracle ?       ora_mmnl_DWODI
```

The prior output identifies one operating system process consuming an excessive amount of CPU (16.4%). The process ID is 11026 and name is `oracleDWREP`. From the process name, it's an Oracle process associated with the `DWREP` database.

You can determine what type of Oracle process this is by querying the data dictionary:

```
SELECT
  'USERNAME   : ' || s.username     || CHR(10) ||
  'SCHEMA     : ' || s.schemaname   || CHR(10) ||
  'OSUSER     : ' || s.osuser       || CHR(10) ||
  'PROGRAM    : ' || s.program      || CHR(10) ||
```

```
  'SPID       : ' || p.spid         || CHR(10) ||
  'SID        : ' || s.sid          || CHR(10) ||
  'SERIAL#    : ' || s.serial#      || CHR(10) ||
  'KILL STRING: ' || '''' || s.sid || ',' || s.serial# || ''''  || CHR(10) ||
  'MACHINE    : ' || s.machine      || CHR(10) ||
  'TYPE       : ' || s.type         || CHR(10) ||
  'TERMINAL   : ' || s.terminal     || CHR(10) ||
  'SQL ID     : ' || q.sql_id       || CHR(10) ||
  'SQL TEXT   : ' || q.sql_text
FROM v$session s
    ,v$process p
    ,v$sql     q
WHERE s.paddr  = p.addr
AND   p.spid   = '&&PID_FROM_OS'
AND   s.sql_id = q.sql_id(+);
```

The prior script prompts you for the operating system process ID. Here is the output for this example:

```
USERNAME   : MV_MAINT
SCHEMA     : MV_MAINT
OSUSER     : oracle
PROGRAM    : sqlplus@dwdb (TNS V1-V3)
SPID       : 11026
SID        : 410
SERIAL#    : 30653
KILL STRING: '410,30653'
MACHINE    : dwdb
TYPE       : USER
TERMINAL   : pts/2
SQL ID     : by3c8848gyngu
SQL TEXT   : SELECT "A1"."REGISTRATION_ID","A1"."PRODUCT_INSTANCE_ID"
,"A1"."SOA_ID","A1"."REG_SOURCE_IP_ADDR","A1"...
```

The output indicates that this is a SQL*Plus process with a database `SID` of 410 and `SERIAL#` of 30653. You'll need this information if you decide to terminate the process with the `ALTER SYSTEM KILL SESSION` statement (see Recipe 6-10 for details).

In this example, since the process is running a SQL statement, further details about the query can be extracted by generating an execution plan:

```
SQL> SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR(('&&sql_id')));
```

You'll be prompted for the `sql_id` when you run the prior statement (in this example, the `sql_id` is `by3c8848gyngu`). Here is a partial listing of the output:

```
SQL_ID  by3c8848gyngu, child number 0
-------------------------------------
SELECT "A1"."REGISTRATION_ID","A1"."PRODUCT_INSTANCE_ID","A1"."SOA_ID","
A1"."REG_SOURCE_IP_ADDR","A1"."REGISTRATION_STATUS","A1"."CREATE_DTT","A
1"."DOMAIN_ID","A1"."COUNT_FLG","A2"."PRODUCT_INSTANCE_ID","A2"."SVC_TAG
Plan hash value: 4286489280
```

```
-------------------------------------------------------------------------------------------
| Id|Operation                    |Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT             |              |       |       | 64977 (100)|          |
|  1| NESTED LOOPS                |              |       |       |            |          |
|  2|  NESTED LOOPS               |              |     1 |   499 | 64977   (5)| 00:13:00 |
|  3|   NESTED LOOPS OUTER        |              |     1 |   462 | 64975   (5)| 00:13:00 |
|  4|    NESTED LOOPS OUTER       |              |     1 |   454 | 64973   (5)| 00:13:00 |
|  5|     NESTED LOOPS            |              |     1 |   420 | 64972   (5)| 00:13:00 |
|  6|      NESTED LOOPS OUTER     |              |     1 |   351 | 64971   (5)| 00:13:00 |
|  7|       NESTED LOOPS          |              |     1 |   278 | 64969   (5)| 00:13:00 |
|  8|        NESTED LOOPS OUTER   |              |     1 |   188 | 64967   (5)| 00:13:00 |
|  9|         NESTED LOOPS        |              |     1 |   180 | 64966   (5)| 00:13:00 |
|*10|          TABLE ACCESS FULL  |REGISTRATIONS |     1 |    77 | 64964   (5)| 00:13:00 |
```

This output will help you determine the efficiency of the SQL statement and provide insight on how to tune it. Refer to Chapter 9 for details on how to manually tune a query and Chapter 11 for automated SQL tuning.

How It Works

The process described in the "Solution" section of this recipe allows you to quickly identify resource-intensive processes, then map the OS process to a database process, and subsequently map the database process to a SQL statement. Once you know which SQL statement is consuming resources, then you can generate an execution plan to further attempt to determine any possible inefficiencies.

Sometimes the resource-consuming process will not be associated with a database. In these scenarios, you'll have to work with your SA to determine what the process is and if it can be tuned or terminated.

Also, you may encounter resource-intensive processes that are database-specific but not associated with a SQL statement. For example, you might have a long-running RMAN backup process, Data Pump, or PL/SQL jobs running. In these cases, work with your DBA to identify whether these types of processes can be tuned or killed.

ORADEBUG

You can use Oracle's `oradebug` utility to display top consuming SQL statements if you know the operating system ID. For example, suppose that you have used a utility such as `top` or `ps` to identify a high CPU-consuming operating system process, and from the name of the process you determine it's a database process. Now log in to SQL*Plus and use `oradebug` to display any SQL associated with the process. In this example, the OS process ID is 7853:

```
SQL> oradebug setospid 7853;
Oracle pid: 18, Unix process pid: 7853, image: oracle@xengdb (TNS V1-V3)
```

Now show the SQL associated with this process (if any):

```
SQL> oradebug current_sql;
```

If there is a SQL statement associated with the process, it will be displayed—for example:

```
select
 a.table_name
from dba_tables a, dba_indexes b, .....
```

The `oradebug` utility can be used in a variety of methods to help troubleshoot performance issues. You can display the name of the trace file associated with the session by issuing the following:

```
SQL> oradebug tracefile_name;
```

Use `oradebug help` to display all options available.

6-10. Terminating a Resource-Intensive Process

Problem

You have identified a process that is consuming inordinate amounts of system resources (see Recipe 6-9) and determined that it's a runaway SQL statement that needs to be killed.

Solution

There are three basic ways to terminate a SQL process:

- If you have access to the terminal where the SQL statement is running, you can press Ctrl+C and attempt to halt the process.
- Determine the session ID and serial number, and then use the SQL `ALTER SYSTEM KILL SESSION` statement.
- Determine the operating system process ID, and use the `kill` utility to stop the process.

If you happen to have access to the terminal from which the resource-consuming SQL statement is running, you can attempt to press Ctrl+C to terminate the process. Oftentimes you don't have access to the terminal and will have to use a SQL statement or an operating system command to terminate the process.

Using SQL to Kill a Session

If it's an Oracle process and you have identified the `SID` and `SERIAL#` (see Recipe 6-9), you can terminate a process from within SQL*Plus. Here is the general syntax:

```
alter system kill session 'integer1, integer2 [,integer3]' [immediate];
```

In the prior syntax statement, `integer1` is the value of the `SID` column and `integer2` is the value from the `SERIAL#` column (of `V$SESSION`). In a RAC environment, you can optionally specify the value of the instance ID for `integer3`. The instance ID can be retrieved from the `GV$SESSION` view.

Here's an example that terminates a process with a SID of 1177 and a SERIAL# of 38583:

```
SQL> alter system kill session '1177,38583';
```

If successful, you should see this output:

```
System altered.
```

When you kill a session, this will mark the session as terminated, roll back active transactions (within the session), and release any locks (held by the session). The session will stay in a terminated state until any dependent transactions are rolled back. If it takes a minute or more to roll back the transaction, Oracle reports the session as "marked to be terminated" and returns control to the SQL prompt. If you specify IMMEDIATE (optional), Oracle will roll back any active transactions and immediately return control back to you.

Using the OS Kill Command

If you have access to the database server and access to an operating system account that has privileges to terminate an Oracle process (such as the oracle OS account), you can also terminate a process directly with the kill command.

For example, suppose you run the ps command and have done the associated work to determine that you have a SQL statement that has been running for hours and needs to be terminated. The kill command directly terminates the operating system process. In this example, the process ID of 6254 is terminated:

```
$ kill -9 6254
```

■ **Caution** Ensure that you don't kill the wrong Oracle process. If you accidentally kill a required Oracle background process, this will cause your instance to abort.

How It Works

Sometimes you'll find yourself in a situation where you need to kill hung SQL processes, database jobs, or SQL statements that are consuming inordinate amounts of resources. For example, you may have a test server where a job has been running for several hours, is consuming much of the server resources, and needs to be stopped so that other jobs can continue to process in a timely manner.

Manually killing a SQL statement will cause the transaction to be rolled back. Therefore take care when doing this. Ensure that you are killing the correct process. If you erroneously terminate a critical process, this obviously will have an adverse impact on the application and associated data.

CHAPTER 7

Troubleshooting the Database

Oracle Database 11g offers new ways of diagnosing the health of your database. This chapter contains several recipes that show how to use the database's built-in diagnostic infrastructure to resolve database performance issues. You'll learn how to use ADRCI, the Automatic Diagnostic Repository Command Interpreter, to perform various tasks such as checking the database alert log, creating a diagnostic package for sending to Oracle Support engineers, and running a proactive health check of the database.

Many common Oracle database performance-related issues occur when you have space issues with the temporary tablespace or when you're creating a large index or a large table with the `create table as select` (`CTAS`) technique. Undo tablespace space issues are another common source of trouble for many DBAs. This chapter has several recipes that help you proactively monitor, diagnose, and resolve temporary tablespace and undo tablespace–related issues. When a production database seems to hang, there are ways to collect critical diagnostic data for analyzing the causes, and this chapter shows you how to log in to an unresponsive database to collect diagnostic data.

7-1. Determining the Optimal Undo Retention Period

Problem

You need to determine the optimal length of time for undo retention in your database.

Solution

You can specify the length of time Oracle will retain undo data after a transaction commits, by specifying the `UNDO_RETENTION` parameter. Here is how to set the undo retention to 30 minutes for an instance, by updating the value of the `UNDO_RETENTION` parameter in the `SPFILE`.

```
SQl> alter system set undo_retention=1800 scope=both;

System altered.

SQL>
```

To determine the optimal value for the UNDO_RETENTION parameter, you must first calculate the actual amount of undo that the database is generating. Once you know approximately how much undo the database is generating, you can calculate a more precise value for the UNDO_RETENTION parameter. Use the following formula to calculate the value of the UNDO_RETENTION parameter:

```
UNDO_RETENTION = UNDO SIZE/(DB_BLOCK_SIZE*UNDO_BLOCK_PER_SEC)
```

You can calculate the actual undo that's generated in your database by issuing the following query:

```
SQL> select sum(d.bytes) "undo"
  2  from v$datafile d,
  3  v$tablespace t,
  4  dba_tablespaces s
  5  where s.contents = 'UNDO'
  6  and s.status = 'ONLINE'
  7  and t.name = s.tablespace_name
  8  and d.ts# = t.ts#;

 UNDO
----------
 104857600
SQL>
```

You can calculate the value of UNDO_BLOCKS_PER_SEC with the following query:

```
SQL> select max(undoblks/((end_time-begin_time)*3600*24))
  2  "UNDO_BLOCK_PER_SEC"
  3  FROM v$undostat;

UNDO_BLOCK_PER_SEC
------------------
             7.625
SQL>
```

You most likely remember the block size for your database—if not, you can look it up in the SPFILE or find it by issuing the command show parameter db_block_size. Let's say the db_block_size is 8 KB (8,192 bytes) for your database. You can then calculate the optimal value for the UNDO_RETENTION parameter using the formula shown earlier in this recipe—for example, giving a result in seconds: 1,678.69 = 104,857,600/(7.625 * 8,192). In this case, assigning a value of 1,800 seconds for the undo_retention parameter is appropriate, because it's a bit more than what is indicated by our formula for computing the value of this parameter.

How It Works

Automatic undo management is the default mode for undo management starting with release 11g. If you create a database with the Database Configuration Assistant (DBCA), Oracle automatically creates an auto-extending undo tablespace named UNDOTBS1. If you're manually creating a database, you specify the undo tablespace in the database creation statement, or you can add the undo tablespace at any point. If a database doesn't have an explicit undo tablespace, Oracle will store the undo records in the SYSTEM tablespace.

Once you set the UNDO_TABLESPACE initialization parameter, Oracle automatically manages undo retention for you. Optionally, you can set the UNDO_RETENTION parameter to specify how long Oracle retains older undo data before overwriting it with newer undo data.

The formula in the "Solution" section shows how to base the undo retention period on current database activity. Note that we rely on the dynamic view V$UNDOSTAT to calculate the value for the undo retention period. Therefore, it's essential that you execute your queries after the database has been running for some time, thus ensuring that it has had the chance to process a typical workload.

If you configure the UNDO_RETENTION parameter, the undo tablespace must be large enough to hold the undo generated by the database within the time you specify with the UNDO_RETENTION parameter. When a transaction commits, the database may overwrite its undo data with newer undo data. The undo retention period is the minimum time for which the database will attempt to retain older undo data. Oracle retains the undo data for both read consistency purposes as well as to support Oracle Flashback operations, for the duration you specify with the UNDO_RETENTION parameter. After it saves the undo data for the period you specified for the UNDO_RETENTION parameter, the database marks that undo data as *expired* and makes the space occupied by that data available to write undo data for new transactions.

By default, the database uses the following criteria to determine how long it needs to retain undo data:

- Length of the longest-running query
- Length of the longest-running transaction
- Longest flashback duration

It's somewhat difficult to understand how the Oracle database handles the retention of undo data. Here's a brief summary of how things work:

- If you don't configure the undo tablespace with the AUTOEXTEND option, the database simply ignores the value you set for the UNDO_RETENTION parameter. The database will automatically tune the undo retention period based on database workload and the size of the undo tablespace. So, make sure you set the undo tablespace to a large value if you're receiving errors indicating that the database is not retaining undo for a long enough time. Typically, the undo retention in this case is for a duration significantly longer than the longest-running active query in the database.
- If you want the database to try to honor the settings you specify for the UNDO_RETENTION parameter, make sure that you enable the AUTOEXTEND option for the undo tablespace. This way, Oracle will automatically extend the size of the undo tablespace to make room for undo from new transactions, instead of overwriting the older undo data. However, if you're receiving ORA-0155 (snapshot too old) errors, say due to Oracle Flashback operations, it means that the database isn't able to dynamically tune the undo retention period effectively. In a case such as this, try increasing the value of the UNDO_RETENTION parameter to match the length of the longest Oracle Flashback operation. Alternatively, you can try going to a larger fixed-size undo tablespace (without the AUTOEXTEND option).

The key to figuring out the right size for the undo tablespace or the correct setting for the UNDO_RETENTION parameter is to understand the nature of the current database workload. In order to understand the workload characteristics, it's important to examine the V$UNDOSTAT view, because it contains statistics showing how the database is utilizing the undo space, as well as information such as the length of the longest-running queries. You can use this information to calculate the size of the undo space for the current workload your database is processing. Note that each row in the V$UNDOSTAT view

shows undo statistics for a ten-minute time interval. The table contains a maximum of 576 rows, each for a ten-minute interval. Thus, you can review undo usage for up to four days in the past.

Here are the key columns you should monitor in the V$UNDOSTAT view for the time period you're interested in—ideally, the time period should include the time when your longest-running queries are executing. You can use these statistics to size both the UNDO_TABLESPACE as well as the UNDO_RETENTION initialization parameters.

begin_time: Beginning of the time interval.

end_time: End of the time interval.

undoblks: Number of undo blocks the database consumed in a ten-minute interval; this is what we used in our formula for the estimation of the size of the undo tablespace.

txncount: Number of transactions executed in a ten-minute interval.

maxquerylen: This shows the length of the longest query (in seconds) executed in this instance during a ten-minute interval. You can estimate the size of the UNDO_RETENTION parameter based on the maximum value of the MAXQUERYLEN column.

maxqueryid: Identifier for the longest-running SQL statement in this interval.

nospaceerrcnt: The number of times the database didn't have enough free space available in the undo tablespace for new undo data, because the entire undo tablespace was being used by active transactions; of course, this means that you need to add space to the undo tablespace.

tuned_undoretention: The time, in seconds, for which the database will retain the undo data after the database commits the transaction to which the undo belongs.

The following query based on the V$UNDOSTAT view shows how Oracle automatically tunes undo retention (check the TUNED_UNDORETENTION column) based on the length of the longest-running query (MAXQUERYLEN column) in the current instance workload.

```
SQL> select to_char(begin_time,'hh24:mi:ss') BEGIN_TIME,
  2  to_char(end_time,'hh24:mi:ss') END_TIME,
  3  maxquerylen,nospaceerrcnt,tuned_undoretention
  4  from v$undostat;

BEGIN_TI END_TIME MAXQUERYLEN NOSPACEERRCNT TUNED_UNDORETENTION
-------- -------- ----------- ------------- -------------------
12:25:35 12:29:30         892             0                1673
12:15:35 12:25:35         592             0                1492
12:05:35 12:15:35        1194             0                2094
11:55:35 12:05:35         592             0                1493
11:45:35 11:55:35        1195             0                2095
11:35:35 11:45:35         593             0                1494
11:25:35 11:35:35        1196             0                2097
11:15:35 11:25:35         594             0                1495
11:05:35 11:15:35        1195             0                2096
```

```
10:55:35 11:05:35         593             0               1495
10:45:35 10:55:35        1198             0               2098
…
SQL>
```

Note that the value of the `TUNED_UNDORETENTION` column fluctuates continuously, based on the value of the maximum query length (`MAXQUERYLEN`) during any interval. You can see that the two columns are directly related to each other, with Oracle raising or lowering the tuned undo retention based on the maximum query length during a given interval (of ten minutes). The following query shows the usage of undo blocks and the transaction count during each ten-minute interval.

```
SQL> select to_char(begin_time,'hh24:mi:ss'),to_char(end_time,'hh24:mi:ss'),
  2  maxquerylen,ssolderrcnt,nospaceerrcnt,undoblks,txncount from v$undostat
  3  order by undoblks
  4  /

TO_CHAR( TO_CHAR( MAXQUERYLEN SSOLDERRCNT NOSPACEERRCNT   UNDOBLKS   TXNCOUNT
-------- -------- ----------- ----------- ------------- ---------- ----------
17:33:51 17:36:49         550           0             0          1         18
17:23:51 17:33:51         249           0             0         33        166
17:13:51 17:23:51         856           0             0         39        520
17:03:51 17:13:51         250           0             0         63        171
16:53:51 17:03:51         850           0             0        191        702
16:43:51 16:53:51         245           0             0        429        561

6 rows selected.

SQL>
```

Oracle provides an easy way to help set the size of the undo tablespace as well as the undo retention period, through the OEM Undo Advisor interface. You can specify the length of time for the advisor's analysis, for a period going back to a week—the advisor uses the AWR hourly snapshots to perform its analysis. You can specify the undo retention period to support a flashback transaction query. Alternatively, you can let the database determine the desired undo retention based on the longest query in the analysis period.

7-2. Finding What's Consuming the Most Undo

Problem

Often, one or two user sessions seem to be hogging the undo tablespaces. You'd like to identify the user and the SQL statement that are using up all that undo space.

Solution

High undo usage often involves a long-running query. Use the following query to find out which SQL statement has run for the longest time in your database.

```
SQL> select s.sql_text from v$sql s, v$undostat u
     where u.maxqueryid=s.sql_id;
```

You can join the V$TRANSACTION and the V$SESSION views to find out the most undo used by a session for a currently executing transaction, as shown here:

```
SQL> select s.sid, s.username, t.used_urec, t.used_ublk
     from v$session s, v$transaction t
     where s.saddr = t.ses_addr
     order by t.used_ublk desc;
```

You can also issue the following query to find out which session is currently using the most undo in an instance:

```
SQL>select s.sid, t.name, s.value
    from v$sesstat s, v$statname t
    where s.statistic# = t.statistic#
    and t.name = 'undo change vector size'
    order by s.value desc;
```

The query's output relies on the statistic `undo change vector size` in the V$STATNAME view, to show the SID for the sessions consuming the most undo right now. The V$TRANSACTION view shows details about active transactions. Here's another query that joins the V$TRANSACTION, V$SQL and V$SESSION views:

```
SQL> select sql.sql_text sql_text, t.USED_UREC Records, t.USED_UBLK Blocks,
(t.USED_UBLK*8192/1024) KBytes from v$transaction t,
  2  v$session s,
  3  v$sql sql
  4  where t.addr = s.taddr
  5  and s.sql_id = sql.sql_id
  6* and s.username ='&USERNAME'
SQL>
```

The column USED_UREC shows the number of undo records used, and the USED_UBLK column shows the undo blocks consumed by a transaction.

How It Works

You can issue the queries described in the "Solution" section to identify the sessions that are responsible for the most undo usage in your database, as well as the users that are responsible for those sessions. You can query the V$UNDOSTAT with the appropriate begin_time and end_time values to get the SQL identifier of the longest-running SQL statement during a time interval. The MAXQUERYID column captures the SQL identifier. You can use this ID to query the V$SQL view in order to find out the actual SQL statement. Similarly, the V$TRANSACTION and the V$SESSION views together help identify the users that are consuming the most undo space. If excessive undo usage is affecting performance, you might want to look at the application to see why the queries are using so much undo.

7-3. Resolving an ORA-01555 Error

Problem

You're receiving the `ORA-01555` (snapshot too old) errors during nightly runs of key production batch jobs. You want to eliminate these errors.

Solution

While setting a high value for the `UNDO_RETENTION` parameter can potentially minimize the possibility of receiving "snapshot too old" errors, it doesn't guarantee that the database won't overwrite older undo data that may be needed by a running transaction. You can move long-running batch jobs to a separate time interval when other programs aren't running in the database, to avoid these errors.

Regardless, while you can minimize the occurrence of "snapshot too old" errors with these approaches, you can't completely eliminate such errors without specifying the *guaranteed undo retention* feature. When you configure guaranteed undo retention in a database, no transaction can fail because of the "snapshot too old" error. Oracle will keep new DML statements from executing when you set up guaranteed undo retention. Implementing the guaranteed undo feature is simple. Suppose you want to ensure that the database retains undo for at least an hour (3,600 seconds). First set the undo retention threshold with the `alter system` command shown here, and then set up guaranteed undo retention by specifying the `retention guarantee` clause to alter the undo tablespace.

```
SQL> alter system set undo_retention=3600;
System altered.
SQL> alter tablespace undotbs1 retention guarantee;
Tablespace altered.
SQL>
```

You can switch off guaranteed undo retention by executing the `alter tablespace` command with the `retention noguarantee` clause.

■ **Tip** You can enable guaranteed undo retention by using the `alter system` command as shown in this recipe, as well as with the `create database` and `create undo tablespace` statements.

How It Works

Oracle uses the undo records stored in the undo tablespace to help roll back transactions, provide read consistency, and to help recover the database. In addition, the database also uses undo records to read data from a past point in time using Oracle Flashback Query. Undo data serves as the underpinning for several Oracle Flashback features that help you recover from logical errors.

Occurrence of the Error

The `ORA-01555` error (snapshot too old) may occur in various situations. The following is a case where the error occurs during an export.

```
EXP-00008: ORACLE error 1555 encountered
ORA-01555: snapshot too old: rollback segment number 10 with name "_SYSSMU10$" too small
EXP-00000: Export terminated unsuccessfully
```

And you can receive the same error when performing a flashback transaction:

```
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number  with name "" too small
ORA-06512: at "SYS.DBMS_FLASHBACK", line 37
ORA-06512: at "SYS.DBMS_FLASHBACK", line 70
ORA-06512: at li
```

The "snapshot too old" error occurs when Oracle overwrites undo data that's needed by another transaction. The error is a direct result of how Oracle's read consistency mechanism works. The error occurs during the execution of a long-running query when Oracle tries to read the "before image" of any changed rows from the undo segments. For example, if a long-running query starts at 1 a.m. and runs until 6 a.m., it's possible for the database to change the data that's part of this query during the period in which the query executes. When Oracle tries to read the data as it appeared at 1 a.m., the query may fail if that data is no longer present in the undo segments.

If your database is experiencing a lot of updates, Oracle may not be able to fetch the changed rows, because the before changes recorded in the undo segments may have been overwritten. The transactions that changed the rows will have already committed, and the undo segments don't have a record of the before change row values because the database overwrote the relevant undo data. Since Oracle fails to return consistent data for the current query, it issues the `ORA-01555` error. The query that's currently running requires the before image to construct read-consistent data, but the before image isn't available.

The `ORA-01555` error may be the result of one or both of the following: too many updates to the database or too small an undo tablespace. You can increase the size of the undo tablespace, but that doesn't ensure that the error won't occur again.

Influence of Extents

The database stores undo data in undo extents, and there are three distinct types of undo extents:

Active: Transactions are currently using these extents.

Unexpired: These are extents that contain undo that's required to satisfy the undo retention time specified by the `UNDO_RETENTION` initialization parameter.

Expired: These are extents with undo that's been retained longer than the duration specified by the `UNDO_RETENTION` parameter.

If the database doesn't find enough expired extents in the undo tablespace or it can't get new undo extents, it'll re-use the unexpired (but never an active undo extent) extents, and this leaves the door open for an `ORA-01555`, "snapshot too old" error. By default, the database will essentially shrink the undo retention period you specify, if it encounters space pressure to accommodate the undo from new transactions. Since the unexpired undo extents contain undo records needed to satisfy the undo retention period, overwriting those extents in reality means that the database is lowering the undo

retention period you've set. Enabling the undo retention guarantee helps assure the success of long-running queries as well as Oracle Flashback operations. The "guarantee" part of the undo retention guarantee is *real*—Oracle will certainly retain undo at least for the time you specify and will never overwrite any of the unexpired undo extents that contain the undo required to satisfy the undo retention period. However, there's a stiff price attached to this guarantee—Oracle will guarantee retention even if it means that DML transactions fail because the database can't find space to record the undo for those transactions. Therefore, you must exercise great caution when enabling the guaranteed undo retention capability.

7-4. Monitoring Temporary Tablespace Usage

Problem

You want to monitor the usage of the temporary tablespace.

Solution

Execute the following query to find out the used and free space in a temporary tablespace.

```
SQL> select * from (select a.tablespace_name,
      sum(a.bytes/1024/1024) allocated_mb
      from dba_temp_files a
      where a.tablespace_name = upper('&&temp_tsname') group by a.tablespace_name) x,
      (select sum(b.bytes_used/1024/1024) used_mb,
      sum(b.bytes_free/1024/1024) free_mb
      from v$temp_space_header b
      where b.tablespace_name=upper('&&temp_tsname') group by b.tablespace_name);

Enter value for temp_tsname: TEMP
…
TABLESPACE_NAME                ALLOCATED_MB    USED_MB    FREE_MB
------------------------------ ------------ ---------- ----------
TEMP                             52.9921875 52.9921875          0
SQL>
```

Obviously, the temporary tablespace shown in this example is in serious need of some help from the DBA.

How It Works

Oracle uses temporary tablespaces for storing intermediate results from sort operations as well as any temporary tables, temporary LOBs, and temporary B-trees. You can create multiple temporary tablespaces, but only one of them can be the default temporary tablespace. If you don't explicitly assign a temporary tablespace, that user is assigned the default temporary tablespace.

You won't find information about temporary tablespaces in the `DBA_FREE_SPACE` view. Use the `V$TEMP_SPACE_HEADER` as shown in this example to find how much free and used space there is in any temporary tablespace.

7-5. Identifying Who Is Using the Temporary Tablespace

Problem

You notice that the temporary tablespace is filling up fast, and you want to identify the user and the SQL statements responsible for the high temporary tablespace usage.

Solution

Issue the following query to find out which SQL statement is using up space in a sort segment.

```
SQL> select s.sid || ',' || s.serial# sid_serial, s.username,
     o.blocks * t.block_size / 1024 / 1024 mb_used, o.tablespace,
     o.sqladdr address, h.hash_value, h.sql_text
     from v$sort_usage o, v$session s, v$sqlarea h, dba_tablespaces t
     where o.session_addr = s.saddr
     and o.sqladdr = h.address (+)
     and o.tablespace = t.tablespace_name
     order by s.sid;
```

The preceding query shows information about the session that issued the SQL statements well as the name of the temporary tablespace and the amount of space the SQL statement is using in that tablespace.

You can use the following query to find out which sessions are using space in the temporary tablespace. Note that the information is in the summary form, meaning it doesn't separate the various sort operations being run by a session—it simply gives the total temporary tablespace usage by each session.

```
SQL> select s.sid || ',' || s.serial# sid_serial, s.username, s.osuser, p.spid,
     s.module,s.program,
     sum (o.blocks) * t.block_size / 1024 / 1024 mb_used, o.tablespace,
     count(*) sorts
     from v$sort_usage o, v$session s, dba_tablespaces t, v$process p
     where o.session_addr = s.saddr
     and s.paddr = p.addr
     and o.tablespace = t.tablespace_name
     group by s.sid, s.serial#, s.username, s.osuser, p.spid, s.module,
     s.program, t.block_size, o.tablespace
     order by sid_serial;
```

The output of this query will show you the space that each session is using in the temporary tablespace, as well as the number of sort operations that session is performing right now.

How It Works

Oracle tries to perform sort and hash operations in memory (PGA), but if a sort operation is too large to fit into memory, it uses the temporary tablespace to do the work. It's important to understand that even a single large sort operation has the potential to use up an entire temporary tablespace. Since all database sessions share the temporary tablespace, the session that runs the large sort operation could

potentially result in other sessions receiving errors due to lack of room in that tablespace. Once the temporary tablespace fills up, all SQL statements that seek to use the temporary tablespace will fail with the `ORA-1652: unable to extend temp segment` error. New sessions may not be able to connect, and queries can sometimes hang and users may not be able to issue new queries. You try to find any blocking locks, but none exists. If the temporary tablespace fills up, transactions won't complete. If you look in the alert log, you'll find that the temporary tablespace ran out of space.

Operations that use an `ORDER BY` or `GROUP BY` clause frequently use the temporary tablespace to do their work. You must also remember that creating an index or rebuilding one also makes use of the temporary tablespace for sorting the index.

Oracle uses the PGA memory for performing the sort and hash operations. Thus, one of the first things you must do is to review the current value set for the `PGA_AGGREGATE_TARGET` initialization parameter and see if bumping it up will help. Nevertheless, even a larger setting for the `PGA_AGGREGATE_TARGET` parameter doesn't guarantee that Oracle will perform a huge sort entirely in memory. Oracle allocates each session a certain amount of PGA memory, with the amount it allocates internally determined, based on the value of the `PGA_AGGREGATE_TARGET` parameter. Once a large operation uses its share of the PGA memory, Oracle will write intermediary results to disk in the temporary tablespace. These types of operations are called *one-pass* or *multi-pass* operations, and since they are performed on disk, they are much slower than an operation performed entirely in the PGA.

If your database is running out of space in the temporary tablespace, you must increase its size by adding a tempfile. Enabling *autoextend* for a temporary tablespace will also help prevent "out of space" errors. Since Oracle allocates space in a temporary tablespace that you have assigned for the user performing the sort operation, you can assign users that need to perform heavy sorting a temporary tablespace that's different from that used by the rest of the users, thus preventing the heavy sorting activity from hurting database performance.

Note that unlike table or index segments, of which there are several for each object, a temporary tablespace has just one segment called the sort segment. All sessions share this sort segment. A single SQL statement can use multiple sort and hash operations. In addition, the same session can have multiple SQL statements executing simultaneously, with each statement using multiple sort and hash operations. Once a sort operation completes, the database immediately marks the blocks used by the operations as free and allocates them to another sort operation. The database adds extents to the sort segment as the sort operation gets larger, but if there's no more free space in the temporary tablespace to allocate additional extents, it issues the `ORA-1652:unable to extend temp segment` error. The SQL statement that's using the sort operation will fail as a result.

■ **Note** Although you'll receive an `ORA-1652` error when a SQL statement performing a huge sort fails due to lack of space in the temporary tablespace, that's not the only reason you'll get this error. You'll also receive this error when performing a table move operation (`alter table …move`), if the tablespace to which you're moving the table doesn't have room to hold the table. Same is the case sometimes when you're creating a large index. Please see Recipe 7-6 for an explanation of this error.

7-6. Resolving the "Unable to Extend Temp Segment" Error

Problem

While creating a large index, you receive an Oracle error indicating that the database is unable to extend a TEMP segment. However, you have plenty of free space in the temporary tablespace.

Solution

When you get an error such as the following, your first inclination may be to think that there's no free space in the temporary tablespace.

```
ORA-01652: unable to extend temp segment by 1024 in tablespace INDX_01
```

You cannot fix this problem by adding space to the temporary tablespace. The error message clearly indicates the tablespace that ran out of space. In this case, the offending tablespace is INDX_01, and not the TEMP tablespace. Obviously, an index creation process failed because there was insufficient space in the INDX_01 tablespace. You can fix the problem by adding a datafile to the INDX_01 tablespace, as shown here:

```
SQL>alter tablespace INDX_01 add datafile '/u01/app/oracle/data/indx_01_02.dbf'
  2 size 1000m;
```

How It Works

When you receive the ORA-01652 error, your normal tendency is to check the temporary tablespace. You check the DBA_TEMP_FREE_SPACE view, and there's plenty of free space in the default temporary tablespace, TEMP. Well, if you look at the error message carefully, it tells you that the database is unable to extend the temp segment in the INDX_01 tablespace. When you create an index, as in this case, you provide the name of the permanent tablespace in which the database must create the new index. Oracle starts the creation of the new index by putting the new index structure into a temporary segment in the tablespace you specify (INDX_01 in our example) for the index. The reason is that if your index creation process fails, Oracle (to be more specific, the SMON process) will remove the temporary segment from the tablespace you specified for creating the new index. Once the index is successfully created (or rebuilt), Oracle converts the temporary segment into a permanent segment within the INDX_01 tablespace. However, as long as Oracle is still creating the index, the database deems it a temporary segment and thus when an index creation fails, the database issues the ORA-01652 error, which is also the error code for an "out of space" error for a temporary tablespace. The TEMP segment the error refers to is the segment that was holding the new index while it was being built. Once you increase the size of the INDX_01 tablespace, the error will go away.

■ **Tip** The temporary segment in an ORA-1652 error message may not be referring to a temporary segment in a temporary tablespace.

The key to resolving the ORA-01652 error is to understand that Oracle uses temporary segments in places other than a temporary tablespace. While a temporary segment in the temporary tablespace is for activities such as sorting, a permanent tablespace can also use temporary segments when performing temporary actions necessary during the creation of a table (CTAS) or an index.

■ **Tip** When you create an index, the creation process uses two different temporary segments. One temporary segment in the TEMP tablespace is used to sort the index data. Another temporary segment in the permanent tablespace holds the index while it is being created. After creating the index, Oracle changes the temporary segment in the index's tablespace into a permanent segment. The same is the case when you create a table with the CREATE TABLE…AS SELECT (CTAS) option.

As the "Solution" section explains, the ORA-01652 error refers to the tablespace where you're rebuilding an index. If you are creating a new index, Oracle uses the temporary tablespace for sorting the index data. When creating a large index, it may be a smart idea to create a large temporary tablespace and assign it to the user who's creating the index. Once the index is created, you can re-assign the user the original temporary tablespace and remove the large temporary tablespace. This strategy helps avoid enlarging the default temporary tablespace to a very large size to accommodate the creation of a large index.

If you specify autoextend for a temporary tablespace, the temp files may get very large, based on one or two large sorts in the database. When you try to reclaim space for the TEMP tablespace, you may get the following error.

```
SQL> alter database tempfile '/u01/app/oracle/oradata/prod1/temp01.dbf' resize 500M;
alter database tempfile '/u01/app/oracle/oradata/prod1/temp01.dbf' resize 500M
*ERROR at line 1:
ORA-03297: file contains used data beyond requested RESIZE value
```

One solution is to create a new temporary tablespace, make that the default temporary tablespace, and then drop the larger temporary tablespace. In Oracle Database 11g, you can simplify matters by using the following alter tablespace command to shrink the temporary tablespace:

```
SQL> alter tablespace temp shrink space;

Tablespace altered.

SQL>
```

In this example, we shrank the entire temporary tablespace, but you can shrink a specific tempfile by issuing the command alter tablespace temp shrink tempfile <file_name>. The command will shrink the tempfile to the smallest size possible.

7-7. Resolving Open Cursor Errors

Problem

You are frequently getting the `Maximum Open Cursors exceeded error`, and you want to resolve the error.

Solution

One of the first things you need to do when you receive the `ORA-01000`: "maximum open cursors exceeded" error is to check the value of the initialization parameter `open_cursors`. You can view the current limit for open cursors by issuing the following command:

```
SQL> sho parameter open_cursors

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------
open_cursors                         integer     300
SQL>
```

The parameter `OPEN_CURSORS` sets the maximum number of cursors a session can have open at once. You specify this parameter to control the number of open cursors. Keeping the parameter's value too low will result in a session receiving the `ORA-01000` error. There's no harm in specifying a very large value for the `OPEN_CURSORS` parameter (unless you expect all sessions to simultaneously max out their cursors, which is unlikely), so you can usually resolve cursor-related errors simply by raising the parameter value to a large number. However, you may sometimes find that raising the value of the `open_cursors` parameter doesn't "fix" the problem. In such cases, investigate which processes are using the open cursors by issuing the following query:

```
SQL> select a.value, s.username,s.sid,s.serial#,s.program,s.inst_id
     from gv$sesstat a,gv$statname b,gv$session s
     where a.statistic# = b.statistic# and s.sid=a.sid
     and b.name='opened cursors current'
```

The `GV$OPEN_CURSOR` (or the `V$OPEN_CURSOR`) view shows all the cursors that each user session has currently opened and parsed, or cached. You can issue the following query to identify the sessions with a high number of opened and parsed or cached cursors.

```
SQL> select saddr, sid, user_name, address,hash_value,sql_id, sql_text
     from gv$open_cursor
     where sid in
     (select sid from v$open_cursor
     group by sid having count(*)  > &threshold);
```

The query lists all sessions with an open cursor count greater than the threshold you specify. This way, you can limit the query's output and focus just on the sessions that have opened, parsed, or cached a large number of cursors.

You can get the actual SQL code and the open cursor count for a specific session by issuing the following query:

```
SQl> select sql_id,substr(sql_text,1,50) sql_text, count(*)
     from gv$open_cursor where sid=81
     group by sql_id,substr(sql_text,1,50)
     order by sql_id;
```

The output shows the SQL code for all open cursors in the session with the SID 81. You can examine all SQL statements with a high open cursor count, to see why the session was keeping a large number of cursors open.

How It Works

If your application is not closing open cursors, then setting the `OPEN_CURSORS` parameter to a higher value won't really help you. You may momentarily resolve the issue, but you're likely to run into the same issue a little later. If the application layer never closes the ref cursors created by the PL/SQL code, the database will simply hang on to the server resources for the used cursors. You must fix the application logic so it closes the cursors—the problem isn't really in the database.

If you're using a Java application deployed on an application server such as the Oracle WebLogic Server, the WebLogic Server's JDBC connection pools provide open database connections for applications. Any prepared statements in each of these connections will use a cursor. Multiple application server instances and multiple JDBC connection pools will mean that the database needs to support all the cursors. If multiple requests share the same session ID, the open cursor problem may be due to implicit cursors. The only solution then is to close the connection after each request.

A *cursor leak* is when the database opens cursors but doesn't close them. You can run a 10046 trace for a session to find out if it's closing its cursors:

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

If you notice that the same SQL statement is associated with different cursors, it means that the application isn't closing its cursors. If the application doesn't close its cursors after opening them, Oracle assigns different cursor numbers for the next SQL statement it executes. If the cursor is closed, instead, Oracle will re-use the same cursor number for the next cursor it assigns. Thus, if you see the item `PARSING IN CURSOR #nnnn` progressively increase in the output for the 10046 trace, it means that the application is not closing the cursors. Note that while leaving cursors open may be due to a faulty application design, developers may also intentionally leave cursors open to reduce soft parsing, or when they use the session cursor cache.

You can use the `SESSION_CACHED_CURSORS` initialization parameter to set the maximum number of cached closed cursors for each session. The default setting is 50. You can use this parameter to prevent a session from opening an excessive number of cursors, thereby filling the library cache or forcing excessive hard parses. Repeated parse calls for a SQL statement leads Oracle to move the session cursor for that statement into the session cursor cache. The database satisfies subsequent parse calls by using the cached cursor instead of re-opening the cursor.

When you re-execute a SQL statement, Oracle will first try to find a parsed version of that statement in the shared pool—if it finds the parsed version in the shared pool, a soft parse occurs. Oracle is forced to perform the much more expensive hard parse if it doesn't find the parsed version of the statement in the shared pool. While a soft parse is much less expensive than a hard parse, a large number of soft parses can affect performance, because they do require CPU usage and library cache latches. To reduce the number of soft parses, Oracle caches the recent closed cursors of each session in a local session

cache for that session—Oracle stores any cursor for which a minimum of three parse calls were made, thus avoiding having to cache every single session cursor, which will fill up the cursor cache.

The default value of 50 for the `SESSION_CACHED_CURSORS` initialization parameter may be too low for many databases. You can check if the database is bumping against the maximum limit for session-cached cursors by issuing the following statement:

```
SQL> select max(value) from v$sesstat
  2  where statistic# in (select statistic# from v$statname
  3* where name = 'session cursor cache count');


MAX(VALUE)
----------
        49

SQL>
```

The query shows the maximum number of session cursors that have been cached in the past. Since this number (49) is virtually the same as the default value (or the value you've set) for the `SESSION_CACHED_CURSORS` parameter, you must set the parameter's value to a larger number. Session cursor caches use the shared pool. If you're using automatic memory management, there's nothing for you to do after you reset the `SESSION_CACHED_CURSORS` parameter—the database will bump up the shared pool size if necessary. You can find out how many cursors each session has in its session cursor cache by issuing the following query:

```
SQL> select a.value,s.username,s.sid,s.serial#
  2  from v$sesstat a, v$statname b,v$session s
  3  where a.statistic#=b.statistic# and s.sid=a.sid
  4* and b.name='session cursor cache count';
```

7-8. Resolving a Hung Database

Problem

Your database is hung. Users aren't able to log in, and existing users can't complete their transactions. The DBAs with SYSDBA privileges may also be unable to log in to the database. You need to find out what is causing the database to hang, and fix the problem.

Solution

Follow these general steps when facing a database that appears to be hung:

1. Check your alert log to see if the database has reported any errors, which may indicate why the database is hanging.
2. See if you can get an AWR or ASH report or query some of the ASH views, as explained in Chapter 5. You may notice events such as hard parses at the top of the Load Profile section of the AWR report, indicating that this is what is slowing down the database.

3. A single ad hoc query certainly has the potential to bring an entire database to its knees. See if you can identify one or more very poorly performing SQL statements that may be leading to the hung (or a very poorly performing) database.
4. Check the database for blocking locks as well as latch contention.
5. Check the server's memory usage as well as CPU usage. Make sure the sessions aren't stalling because you've sized the PGA too low, as explained in Chapter 3.
6. Don't overlook the fact that a scary-looking database hang may be caused by something as simple as the filling up of all archive log destinations. If the archive destination is full, the database will hang, and new user connections will fail. You can, however, still connect as the SYS user, and once you make room in the archive destination by moving some of the archived redo log files, the database becomes accessible to the users.
7. Check the Flash Recovery Area (FRA). A database also hangs when it's unable to write Flashback Database logs to the recovery area. When the FRA fills up, the database won't process new work and it won't spawn new database connections. You can fix this problem by making the recovery area larger with the `alter system set db_recovery_file_dest_size` command.

If you're still unable to resolve the reasons for the hung database, you most likely have a truly hung database. While you're investigating such a database, you may sometimes find yourself unable to connect and log in. In that case, use the "prelim" option to log in to the database. The prelim option doesn't require a real database connection. Here's an example that shows how to use the prelim option to log into a database:

```
C:\app\ora\product\11.2.0\dbhome_1\bin>sqlplus /nolog

SQL*Plus: Release 11.2.0.1.0 Production on Sun Mar 27 10:43:31 2011

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

SQL> set _prelim on
SQL> connect / as sysdba
Prelim connection established
SQL>
```

Alternatively, you can use the command `sqlplus -prelim "/ as sysdba"` to log in with the `-prelim` option. Note that you use the `nolog` option to open a SQL*Plus session. You can't execute the `set _prelim on` command if you're already connected to the database. Once you establish a `prelim` connection as shown here, you can execute the `oradebug hanganalyze` command to analyze a hung database—for example:

```
SQL> oradebug hanganalyze 3
Statement processed.
SQL>
```

In an Oracle RAC environment, specify the `oradebug hanganalyze` command with additional options, as shown here:

```
SQL> oradebug setinst all
SQL> oradebug -g def hanganalyze 3
```

You can repeat the `oradebug hanganalyze` command a couple of times to generate dump files for varying process states.

In addition to the dump files generated by the `hanganalyze` command, Oracle Support may often also request a process state dump, also called a `systemstate` dump, to analyze hung database conditions. The `systemstate` dump will report on what the processes are doing and the resources they're currently holding. You can get a `systemstate` dump from a non-RAC system by executing the following set of commands.

```
SQL> oradebug setmypid
Statement processed.
SQL> oradebug dump systemstate 266
Statement processed.
SQL>
```

Issue the following commands to get a `systemstate` dump in a RAC environment:

```
SQL> oradebug setmypid
SQL> oradebug unlimit
SQL> oradebug -g all dump systemstate 266
```

Note that unlike the `oradebug hanganalyze` command, you must connect to a process. The `setmypid` option specifies the process, in this case your own process. You can also specify a process ID other than yours, in which case you issue the command `oradebug setmypid <pid>` before issuing the `dump systemstate` command. If you try to issue the `dump systemstate` command without setting the PID, you'll receive an error:

```
SQL> oradebug dump systemstate 10
ORA-00074: no process has been specified
SQL>
```

You must take the `systemstate` dumps a few times, with an interval of about a minute or so in between the dumps. Oracle Support usually requests several systemstate dumps along with the trace files generated by the `hanganalyze` command.

How It Works

The key thing you must ascertain when dealing with a "hung" database is whether the database is really hung, or just slow. If one or two users complain about a slow-running query, you need to analyze their sessions, using the techniques described in Chapter 5, to see if the slowness is due to a blocking session or to an Oracle wait event. If several users report that their work is going slowly, it could be due to various reasons, including CPU, memory (SGA or PGA), or other system resource issues.

Check the server's CPU usage as one of your first steps in troubleshooting a hung database. If your server is showing 100% CPU utilization, or if it's swapping or paging, the problem may not lie in the database at all. As for memory, if the server doesn't have enough free memory, new sessions can't connect to the database.

■ **Tip** The "prelim" option shown in the "Solution" section lets you connect to the SGA without opening a session. You can thus "log" in to the hung database even when normal SQL*Plus logins don't work. The `oradebug` session you start once you connect to the SGA actually analyzes what's in the SGA and dumps it into a trace file.

A true database hang can be due to a variety of reasons, including a system that has exhausted resources such as the CPU or memory, or because several sessions are stuck waiting for a resource such as a lock. While the database can automatically resolve deadlocks between sessions (by killing one of the sessions holding a needed lock), when there's a latch or pin on an internal kernel-level resource, Oracle is sometimes unable to automatically detect and resolve the internal deadlock—and this leads to what Oracle Support calls a "true database hang." A true database hang is thus an internal deadlock or a cyclical dependency among multiple processes. Oracle Support will usually ask you to provide them the `hanganalyze` trace files and multiple systemstate dumps to enable them to diagnose the root cause of your hang. At times like this, you may not even be able to log into the database. Your first instinct when you realize that you can't even log in to a database is to try shutting down and restarting, often referred to as bouncing the database. Unfortunately, while shutting down and restarting the database may "resolve" the issue, it'll also disconnect all users—and you're no wiser as to what exactly caused the problem. If you do decide to bounce your database, quickly generate a few `hanganalyze` and `systemstate` dumps first.

■ **Tip** As unpleasant as it may be at times, if you find that you simply can't connect to a hung database, then collect any trace dumps you may need, and quickly bounce the database so that users can access their applications. Especially when you're dealing with a database that's hanging because of memory issues, bouncing the instance may get things going again quickly.

If you find that the database is completely unresponsive, and you can't even log in to the database with the SYSDBA privilege, you can use the prelim option to log into the database. The prelim option stands for preliminary connection, and it starts an Oracle process and attaches that process to the SGA shared memory. However, this is not a full or complete connection, but a limited connection where the structures for query execution are not set up—so, you cannot even query the `V$` views. However, the prelim option lets you run `oradebug` commands to get error dump stacks for diagnostic purposes. The output of the `hanganalyze` command can tell Oracle Support engineers if your database is really hanging, because of sessions waiting for some resource. The command makes internal kernel calls to find out all sessions that are waiting for a resource and shows the relationship between the blocking and waiting sessions. The `hanganalyze` option that you can specify with either the `oradebug` command or an `alter session` statement produces details about hung sessions. Once you get the dump file, Oracle Support personnel can analyze it and let you know the reasons for the database hang.

You can invoke the `hanganalyze` command at various levels ranging from 1 to 10. Level 3 dumps processes that are in a hanging (`IN_HANG`) state. You normally don't need to specify a level higher than 3, because higher levels will produce voluminous reports with too many details about the processes.

■ **Note** The dump files you create with the `hanganalyze` and the `systemstate` commands are created in ADR's trace directory.

Note that we issued the `oradebug` command to get a `systemstate` dump with a level of 266. Level 266 (combination of Level 256, which produces short stack information, and Level 10) is for Oracle releases 9.2.0.6 and onward (earlier releases used `systemstate level 10`). Level 266 allows you to dump the short stacks for each process, which are Oracle function calls that help Oracle development teams determine which Oracle function is causing the problem. The short stack information also helps in matching known bugs in the code. On Solaris and Linux systems, you can safely specify level 266, but on other systems, it may take a long time to dump the short stacks. Therefore, you may want to stick with level 10 for the other operating systems.

If you can find out the blocking session, you can also take a dump just for that session, by using the command `oradebug setospid nnnn`, where `nnnn` is the blocking session's PID, and then invoking the `oradebug` command, as shown here:

```
SQL> oradebug setospid  9999
SQL> oradebug unlimit
SQL> oradebug dump errorstack  3
```

Note that you can generate the `hanganalyze` and `systemstate` dumps in a normal session (as well as in a prelim session), without using the `oradebug` command. You can invoke the `hanganalyze` command with an `alter session` command, as shown here.

```
SQL>alter session set  events 'immediate trace name hanganalyze level 3';
```

Similarly, you can get a `systemstate` dump with the following command:

```
SQL> alter session set events 'immediate trace name SYSTEMSTATE level 10';
Session altered.
SQL>
```

The `oradebug` and `systemstate` dumps are just two of the many dumps you can collect. Use the `oradebug dumplist` command to view the various error dumps you can collect.

```
SQL> oradebug dumplist
TRACE_BUFFER_ON
TRACE_BUFFER_OFF
LATCHES
PROCESSSTATE
SYSTEMSTATE
INSTANTIATIONSTATE
REFRESH_OS_STATS
CROSSIC
CONTEXTAREA
HANGDIAG_HEADER
HEAPDUMP
…
```

Note that while you can read some of the dump files in an editor, these files are mainly for helping Oracle Support professionals troubleshoot a database hang situation. There's not much you can do with the dump files, especially when a database hang situation is due to an Oracle bug or a kernel-level lock, except to send them along to Oracle Support for analysis.

7-9. Invoking the Automatic Diagnostic Repository Command Interpreter

Problem

You'd like to invoke the Automatic Diagnostic Repository Command Interpreter (ADRCI) and work with various components of the Automatic Diagnostic Repository (ADR).

Solution

ADRCI is a tool to help you manage Oracle diagnostic data. You can use ADRCI commands in both an interactive as well as a batch mode.

To start ADRCI in the interactive mode, type **adrci** at the command line, as shown here:

```
 $ adrci

ADRCI: Release 11.2.0.1.0 - Production on Mon Mar 14 11:41:41 2011

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.

ADR base = "c:\app\ora"
adrci>
```

You can issue the `adrci` command from any directory, so long as the PATH environment variable includes `ORACLE_HOME/bin/`. You can enter each command at the `adrci` prompt, and when you're done using the utility, you can type **EXIT** or **QUIT** to exit. You can view all the ADRCI commands available to you by typing **HELP** at the ADRCI command line, as shown here:

```
adrci> HELP

 HELP [topic]
   Available Topics:
        CREATE REPORT
        ECHO
        EXIT
        HELP
        HOST
        IPS
…
        SHOW HOMES | HOME | HOMEPATH
        SHOW INCDIR
        SHOW INCIDENT
        SHOW PROBLEM
```

```
        SHOW REPORT
        SHOW TRACEFILE
        SPOOL

 There are other commands intended to be used directly by Oracle, type
 "HELP EXTENDED" to see the list

adrci>
```

You can get detailed information for an individual ADRCI command by adding the name of the command as an attribute to the `HELP` command. For example, here is how to get the syntax for the `show tracefile` command:

```
adrci> help show tracefile

  Usage: SHOW TRACEFILE [file1 file2 ...] [-rt | -t]
                        [-i inc1 inc2 ...] [-path path1 path2 ...]

  Purpose: List the qualified trace filenames.
…
  Options:
  Examples:
…
adrci>
```

You can also execute ADRCI commands in the batch mode by incorporating the commands in a script or batch file. For example, if you want to run the ADRCI commands `SET HOMEPATH` and `SHOW ALERT` from within an operating system script, include the following line inside a shell script:

```
SET HOMEPATH diag/rdbms/orcl/orcl; SHOW ALERT -term
```

Let's say your script name is `myscript.txt`. You can then execute this script by issuing the following command inside an operating system shell script or batch file:

```
$ adrci script=myscript.txt
```

Note that the parameter `SCRIPT` tells ADRCI that it must execute the commands within the text file `myscript.txt`. If the text file is not within the same directory from where the shell script or batch file is running, you must provide the path for the directory where you saved the text file.

To execute an ADRCI command directly at the command line instead of invoking ADRCI first and working interactively with the ADR interface, specify the parameter `EXEC` with the ADRCI command, as shown here:

```
$ adrci EXEC="SHOW HOMES; SHOW INCIDENT"
```

This example shows how to include two ADRCI commands—`SHOW HOMES` and `SHOW INCIDENT`—by executing the ADRCI command at the command line.

How It Works

The Automatic Diagnostic Repository is a directory structure that you can access even when the database is down, because it's stored outside the database. The root directory for the ADR is called the *ADR base* and is set by the `DIAGNOSTIC_DEST` initialization parameter. Each Oracle product or component has its own *ADR home* under the ADR base. The location of each of the ADR homes follows the path

`diag/product_type/product_id/instance_id`. Thus, the ADR home for a database named `orcl1` with the instance name `orcl1` and the ADR base set to `/app/oracle` will be `/app/oracle/diag/rdbms/orcl1/orcl1`. Under each of the ADR homes are the diagnostic data, such as the trace and dump files and the alert log, as well as other diagnostic files, for that instance of an Oracle product.

The ADRCI utility helps you manage the diagnostic data in the ADR. ADRCI lets you perform the following types of diagnostic tasks:

- *View diagnostic data in the ADR (Automatic Diagnostic Repository)*: The ADR stores diagnostic data such as alert logs, dump files, trace files, health check reports, etc.
- *View health check reports*: The diagnosability infrastructure automatically runs health checks to capture details about the error and adds them to other diagnostic data it collects for that error. You can also manually invoke a health check.
- *Package incidents and problem information for transmission to Oracle Support*: A *problem* is a critical database error, and an *incident* is a single occurrence of a specific problem. An incident package is a collection of diagnostic data that you send to Oracle Support for troubleshooting purposes. ADRCI has special commands that enable you to create packages and generate zipped diagnostic packages to send to Oracle Support.

You can view all ADR locations for the current database instance by querying the `V$DIAG_INFO` view, as shown here.

```
SQL> select * from v$diag_info;

   INST_ID           NAME                 VALUE
-----------    ------------------     ----------------------------------------
1              Diag Enabled           TRUE
1              ADR Base               c:\app\ora
1              ADR Home               c:\app\ora\diag\rdbms\orcl1\orcl1
1              Diag Trace             c:\app\ora\diag\rdbms\orcl1\orcl1\trace
1              Diag Alert             c:\app\ora\diag\rdbms\orcl1\orcl1\alert
1              Diag Incident          c:\app\ora\diag\rdbms\orcl1\orcl1\incident
1              Diag Cdump             c:\app\ora\diag\rdbms\orcl1\orcl1\cdump
1              Health Monitor         c:\app\ora\diag\rdbms\orcl1\orcl1\hm
1              Default Trace File     c:\app\ora…\trace\orcl1_ora_6272.trc
1              Active Problem Count        2
1              Active Incident Count       3

11 rows selected.

SQL>
```

The ADR home is the root directory for a database's diagnostic data. All diagnostic files such as the alert log and the various trace files are located under the ADR home. The ADR home is located directly underneath the ADR base, which you specify with the `DIAGNOSTIC_DEST` initialization parameter. Here's how to find out the location of the ADR base directory:

```
adrci> show base
ADR base is "c:\app\ora"
adrci>
```

You can view all the ADR homes under the ADR base by issuing the following command:

```
adrci> show homes
ADR Homes:
diag\clients\user_salapati\host_3975876188_76
diag\clients\user_system\host_3975876188_76
diag\rdbms\orcl1\orcl1
diag\tnslsnr\miropc61\listener
adrci>
```

You can have multiple ADR homes under the ADR base, and multiple ADR homes can be current at any given time. Your ADRCI commands will work only with diagnostic data in the current ADR home. How do you know which ADR home is current at any point in time? The ADRCI homepath helps determine the ADR homes that are current, by pointing to the ADR home directory under the ADR base hierarchy of directories.

■ **Note** Some ADRCI commands require only one ADR home to be current—these commands will issue an error if multiple ADR homes are current.

You can use either the `show homes` or the `show homepath` command to view all ADR homes that are current:

```
adrci> show homepath
ADR Homes:
diag\clients\user_salapati\host_3975876188_76
diag\clients\user_system\host_3975876188_76
diag\rdbms\orcl1\orcl1
diag\tnslsnr\miropc61\listener
adrci>
```

If you want to work with diagnostic data from multiple database instances or components, you must ensure that all the relevant ADR homes are current. Most of the time, however, you'll be dealing with a single database instance or a single Oracle product or component such as the listener, for example. An ADR homepath is always relative to the ADR. If you specify `/u01/app/oracle/` as the value for the ADR base directory, for example, all ADR homes will be under the `ADR_Base/diag` directory. Issue the `set homepath` command to set an ADR home directory to a single home, as shown here:

```
adrci> set homepath diag\rdbms\orcl1\orcl1

adrci> show homepath

ADR Homes:
diag\rdbms\orcl1\orcl1
adrci>
```

■ **Note** Diagnostic data includes descriptions of incidents and problems, health monitoring reports, and traditional diagnostic files such as trace files, dump files, and alert logs.

Note that before you set the homepath with the `set homepath` command, the `show homepath` command shows all ADR homepaths. However, once you set the homepath to a specific home, the `show homepath` command shows just a single homepath. It's important to set the homepath before you execute several ADRCI commands, as they are applicable to only a single ADR home. For example, if you don't set the homepath before issuing the following command, you'll receive an error:

```
adrci> ips create package
DIA-48448: This command does not support multiple ADR homes
adrci>
```

The error occurs because the `ips create package` command is not valid with multiple ADR homes. The command will work fine after you issue the `set homepath` command to set the homepath to a single ADR home. Commands such as the one shown here work only with a single current ADR home, but others work with multiple current ADR homes—there are also commands that don't need a current ADR home. The bottom line is that all ADRCI commands will work with a single current ADR home.

7-10. Viewing an Alert Log from ADRCI

Problem

You want to view an alert log by using ADRCI commands.

Solution

To view an alert log with ADRCI, follow these steps:

1. Invoke ADRCI.

   ```
   $ adrci
   ```

2. Set the ADR home with the `set homepath` command.

   ```
   adrci> set homepath diag\rdbms\orcl1\orcl1
   ```

3. Enter the following command to view the alert log:

   ```
   adrci>show alert

   ADR Home = c:\app\ora\diag\rdbms\orcl1\orcl1:
   ********************************************************************************
   Output the results to file: c:\temp\alert_10916_7048_orcl1_1.ado
   ```

The alert log will pop up in your default editor. The ADRCI prompt will return once you close the text file in the editor.

You can also query the V$DIAG_INFO view to find the path that corresponds to the Diag Trace entry. You can change the directory to that path and open the alert_<db_name>.log file with a text editor.

How It Works

The alert log holds runtime information for an Oracle instance and provides information such as the initialization parameters the instance is using, as well as a record of key changes such as redo log file switches and, most importantly, messages that show Oracle errors and their details. The alert log is critical for troubleshooting purposes, and is usually the first place you'll look when a problem occurs. Oracle provides the alert log as both a text file as well as an XML-formatted file.

The show alert command brings up the XML-formatted alert log without displaying the XML tags. You can set the default editor with the SET EDITOR command, as shown here:

```
adrci> set editor notepad.exe
```

The previous command changes the default editor to Notepad. The show alert -term command shows the alert log contents in the terminal window. If you want to examine just the latest events in the alert log, issue the following command:

```
adrci>show alert -tail 50
```

The tail option shows you a set of the most recent lines from the alert log in the command window. In this example, it shows the last 50 lines from the alert log. If you don't specify a value for the tail parameter, by default, it shows the last ten lines from the alert log.

The following command shows a "live" alert log, in the sense that it will show changes to the alert log as the entries are added to the log.

```
adrci> show alert -tail -f
```

The previous command shows the last ten lines of the alert log and prints all new messages to the screen, thus offering a "live" display of ongoing additions to the alert log. The CTRL+C sequence will take you back to the ADRCI prompt.

When troubleshooting, it is very useful to see if the database issued any ORA-600 errors. You can issue the following command to trap the ORA-600 errors.

```
adrci> show alert -p "MESSAGE_TEXT LIKE '%ORA-600%'"
```

Although you can view the alert log directly by going to the file system location where it's stored, it's much easier to do so through the ADRCI tool. ADRCI is especially useful for working with the trace files of an instance. The SHOW TRACEFILE command shows all the trace files in the trace directory of the instance. You can issue the SHOW TRACEFILE command with various filters—the following example looks for trace files that reference the background process mmon:

```
$ adrci> show tracefile %mmon%
     diag\rdbms\orcl1\orcl1\trace\orcl1_mmon_1792.trc
     diag\rdbms\orcl1\orcl1\trace\orcl1_mmon_2340.trc
adrci>
```

This command lists all trace files with the string mmon in their file names. You can apply filters to restrict the output to just the trace files associated with a specific incident number, as shown here:

```
adrci> show tracefile -I 43417
          diag\rdbms\orcl1\orcl1\incident\incdir_43417\orcl1_ora_4276_i43417.trc
adrci>
```

The previous command lists the trace files related to the incident number 43417.

7-11. Viewing Incidents with ADRCI

Problem

You want to use ADRCI to view incidents.

Solution

You can view all incidents in the ADR with the `show incident` command (be sure to set the homepath first):

```
$ adrci
$ set homepath diag\rdbms\orcl1\orcl1


adrci> show incident

ADR Home = c:\app\ora\diag\rdbms\orcl1\orcl1:
*******************************************************************************
INCIDENT_ID          PROBLEM_KEY
 CREATE_TIME
-------------------- ------------------------------------------------------------
 43417                ORA 600 [kkqctinvvm(2): Inconsistent state space!]
 2010-12-17 09:26:15.091000 -05:00
43369                ORA 600 [kkqctinvvm(2): Inconsistent state space!]
 2010-12-17 11:08:40.589000 -05:00
79451                ORA 445
 2011-03-04 03:00:39.246000 -05:00
84243                ORA 445
 2011-03-14 19:12:27.434000 -04:00
84244                ORA 445
 2011-03-20 16:55:54.501000 -04:00
5 rows fetched
```

You can specify the `detail` mode to view details about a specific incident, as shown here:

```
adrci> show incident -mode detail -p "incident_id=43369"

ADR Home = c:\app\ora\diag\rdbms\orcl1\orcl1:
*******************************************************************************

INCIDENT INFO RECORD 1
*******************************************************************************
   INCIDENT_ID                   43369
   STATUS                        ready
   CREATE_TIME                   2010-12-17 11:08:40.589000 -05:00
   PROBLEM_ID                    1
```

```
   CLOSE_TIME                    <NULL>
   FLOOD_CONTROLLED              none
   ERROR_FACILITY                ORA
   ERROR_NUMBER                  600
   ERROR_ARG1                    kkqctinvvm(2): Inconsistent state space!
   SIGNALLING_COMPONENT          SQL_Transform
   PROBLEM_KEY                   ORA 600 [kkqctin: Inconsistent state space!]
   FIRST_INCIDENT                43417
   FIRSTINC_TIME                 2010-12-17 09:26:15.091000 -05:00
   LAST_INCIDENT                 43369
   LASTINC_TIME                  2010-12-17 11:08:40.589000 -05:00
   KEY_VALUE                     ORACLE.EXE.3760_3548
   KEY_NAME                      PQ
   KEY_NAME                      SID
   KEY_VALUE                     71.304
   OWNER_ID                      1
   INCIDENT_FILE                 c:\app\ora\diag\rdbms\orcl1\orcl1\trace\orcl1_ora_3548.trc

adrci>
```

How It Works

The `show incident` command reports on all open incidents in the database. For each incident, the output for this command shows the problem key, incident ID, and the time when the incident occurred. In this example, we first set the ADRCI homepath, so the command shows incidents from just this ADR home. If you don't set the homepath, you'll see incidents from all the current ADR homes.

As mentioned earlier, an incident is a single occurrence of a problem. A problem is a critical error such as an `ORA-600` (internal error) or an `ORA-07445` error relating to operating system exceptions. The problem key is a text string that shows the problem details. For example, the problem key `ORA 600 [kkqctinvvm(2): Inconsistent state space!]` shows that the problem is due to an internal error.

When a problem occurs several times, the database creates an incident for each occurrence of the problem, each with a unique incident ID. The database logs the incident in the alert log and sends an alert to the Oracle Enterprise Manager, where they show up in the Home page. The database automatically gathers diagnostic data for the incident, called incident dumps, and stores them in the ADR trace directory.

Since a critical error can potentially generate numerous identical incidents, the fault diagnosability infrastructure applies a "flood control" mechanism to limit the generation of incidents. For a given problem key, the database allows only 5 incidents within one hour and a maximum of 25 incidents in one day. Once a problem triggers incidents beyond these thresholds, the database merely logs the incidents in the alert log and the Oracle Enterprise Manager, but stops generating new incident dumps for them. You can't alter the default threshold settings for the incident flood control mechanism.

7-12. Packaging Incidents for Oracle Support

Problem

You want to send the diagnostic files related to a specific problem to Oracle Support.

Solution

You can package the diagnostic information for one more incidents through either Database Control or through commands that you can execute from the ADRCI interface. In this solution, we show you how to package incidents through ADRCI. You can use various IPS commands to package all diagnostic files related to a specific problem in a zipped format and send the file to Oracle Support. Here are the steps to create an incident package.

1. Create an empty logical package as shown here:

```
adrci> ips create package
Created package 1 without any contents, correlation level typical
adrci>
```

   In this example, we created an empty package, but you can also create a package based on an incident number, a problem number, a problem key or a time interval. In all these cases, the package won't be empty - it'll include the diagnostic information for the incident or problem that you specify. Since we created an empty package, we need to add diagnostic information to that package in the next step.

2. Add diagnostic information to the logical package with the `ips add incident` command:

```
adrci> ips add incident 43369 package 1
Added incident 43369 to package 1
adrci>
```

   At this point, the incident 43369 is associated with package 1, but there'sno diagnostic data in it yet.

3. Generate the physical package.

```
adrci> ips generate package 1 in \app\ora\diagnostics
Generated package 1 in file \app\ora\diagnostics\IPSPKG_20110419131046_COM_1.zip,
mode complete
adrci>
```

   When you issue the `generate package` command, ADRCI gathers all relevant diagnostic files and adds them to a zip file in the directory you designate.

4. Send the resulting zip file to Oracle Support.

   If you decide to add supplemental diagnostic data to an existing physical package (zipped file), you can do so by specifying the `incremental` option with the `generate package` command:

```
adrci> ips generate package 1 in \app\ora\diagnostics incremental
```

   The incremental zip file created by this command will have the term `INC` in the file name, indicating that it is an incremental zip file.

How It Works

A physical package is the zip file that you can send to Oracle Support for diagnosing a problem in your database. Since an incident is a single occurrence of a problem, adding the incident number to the logical package and generating the physical package rolls up all the diagnostic data for that problem (incident) into a single zipped file. In this example, we showed you how to first create an empty logical package and then associate it with an incident number. However, the `ipc create package` command has several options: you can specify the incident number or a problem number directly when you create the logical package, and skip the `add incident` command. You can also create a package that contains all incidents between two points in time, as shown here:

```
adrci> ips create package time '2011-04-12 10:00:00.00 -06:00' to '2011-04-12 23
:00:00.00 -06:00'
Created package 2 based on time range 2011-04-12 12:00:00.000000 -06:00 to 2011-
04-12 23:00:00.000000 -06:00, correlation level typical
adrci>
```

The package generated by the previous command contains all incidents that occurred between 10 a.m. and 11 p.m. on April 12, 2011.

Note that you can also manually add a specific diagnostic file to an existing package. To add a file, you specify the file name in the `ips add file` command—you are limited to adding only those diagnostic files that are within the ADR base directory. Here is an example:

```
adrci> ips add file <ADR_BASE>/diag/rdbms/orcl1/orcl1/trace/orcl_ora12345.trc package 1
```

By default, the `ips generate package` command generates a zip file that includes all files for a package. The incremental option will limit the files to those that the database has generated since you originally generated the zipped file for that package. The `ips show files` command shows all the files in a package and the `ips show incidents` command shows all the incidents in a package. You can issue the `ips remove file` command to remove a diagnostic file from a package.

7-13. Running a Database Health Check

Problem

You'd like to run a comprehensive diagnostic health check on your database. You'd like to find out if there's any data dictionary or file corruption, as well as any other potential problems in the database.

Solution

You can use the database health monitoring infrastructure to run a health check of your database. You can run various integrity checks, such as transaction integrity checks and dictionary integrity checks. You can get a list of all the health checks you can run by querying the `V$HM_CHECK` view:

```
SQL> select name from v$hm_check where internal_check='N';
```

Once you decide on the type of check, specify the name of the check in the DBMS_HM package's RUN_CHECK procedure, as shown here:

```
SQL> begin
  2   dbms_hm.run_check('Dictionary Integrity Check','testrun1');
  3   end;
  4  /

PL/SQL procedure successfully completed.

SQL>
```

You can also run a health check from the Enterprise Manager. Go to Advisor Central ➤ Checkers, and select the specific checker from the Checkers subpage to run a health check.

How It Works

Oracle automatically runs a health check when it encounters a critical error. You can run a manual check using the procedure shown in the "Solution" section. The database stores all health check findings in the ADR.

You can run most of the health checks while the database is open. You can run the Redo Integrity Check and the DB Structure Integrity Check only when the database is closed—you must place the database in the NOMOUNT state to run these two checks.

You can view a health check's findings using either the DBMS_HM package or through the Enterprise Manager. Here is how to get a health check using the DBMS_HM package:

```
SQL> set long 100000
SQL> set longchunksize 1000
SQL> set pagesize 1000
SQL> set linesize 512
SQL> select dbms_hm.get_run_report('testrun1') from dual;

DBMS_HM.GET_RUN_REPORT('TESTRUN1')
------------------------------------------------------------------
Basic Run Information
 Run Name                     : testrun1
 Run Id                       : 61
 Check Name                   : Dictionary Integrity Check
 Mode                         : MANUAL
 Status                       : COMPLETED
 Start Time                   : 2011-04-19 15:46:50.313000 -04:00
 End Time                     : 2011-04-19 15:46:54.117000 -04:00
 Error Encountered            : 0
 Source Incident Id           : 0
 Number of Incidents Created  : 0

Input Paramters for the Run
 TABLE_NAME=ALL_CORE_TABLES
 CHECK_MASK=ALL
```

```
Run Findings And Recommendations

SQL>
```

In this example, fortunately, there are no findings and thus no recommendations, since the dictionary health check didn't find any problems. You can also go to Advisor Central ➤ Checkers and run a report from the Run Detail page for any health check you have run. Use the `show hm_run`, `create report`, and `show report` commands to view health check reports with the ADRCI utility. You can use the views `V$HM_FINDING` and `V$HM_RECOMMENDATION` to investigate the findings as well as the recommendations pursuant to a health check.

7-14. Creating a SQL Test Case

Problem

You need to create a SQL test case in order to reproduce a SQL failure on a different machine, either to support your own diagnostic efforts, or to enable Oracle Support to reproduce the failure.

Solution

In order to create a SQL test case, first you must export the SQL statement along with several bits of useful information about the statement. The following example shows how to capture the SQL statement that is throwing an error. In this example, the user SH is doing the export (you can't do the export as the user SYS).

First, connect to the database as SYSDBA and create a directory to hold the test case:

```
SQL> conn / as sysdba
Connected.
SQL> create or replace directory TEST_DIR1 as 'c:\myora\diagnsotics\incidents\';

Directory created.
SQL> grant read,write on directory TEST_DIR1 to sh;

Grant succeeded.
SQL>
```

Then grant the DBA role to the user through which you will create the test case, and connect as that user:

```
SQL> grant dba to sh;

Grant succeeded.

SQL> conn sh/sh
Connected.
```

Issue the SQL command that's throwing the error:

```
SQL> select * from  my_mv  where max_amount_sold >100000 order by 1;
```

Now you're ready to export the SQL statement and relevant information, which you can import to a different system later on. Use the `EXPORT_SQL_TESTCASE` procedure to export the data, as shown here:

```
SQL> set serveroutput on

SQL> declare mycase clob;
  2  begin
  3  dbms_sqldiag.export_sql_testcase
  4  (directory    =>'TEST_DIR1',
  5  sql_text      => 'select * from my_mv where max_amount_sold >100000 order by 1',
  6  user_name     => 'SH',
  7  exportData    =>  TRUE,
  8  testcase      => mycase
  9  );
 10  end;
 11  /

PL/SQL procedure successfully completed.

SQL>
```

Once the export procedure completes, you are ready to perform the import, either on the same or on a different server. The following example creates a new user named TEST, and imports the test case into that user's schema. Here are the steps for importing the SQL statement and associated information into a different schema.

```
SQL> conn /as sysdba
Connected.
SQL> create or replace directory TEST_DIR2 as 'c:\myora\diagnsotics\incidents\'; /

Directory created.
SQL> grant read,write on directory TEST_dir2 to test;
```

Transfer all the files in the `TEST_DIR1` directory to the `TEST_DIR2` directory. Then grant the DBA role to user TEST, and connect as that user:

```
SQL> grant dba to test;

Grant succeeded.

SQL> conn test/test
Connected.
```

Perform the import of the SQL data as the user TEST, by invoking the `IMPORT_SQL_TESTCASE` procedure, as shown here:

```
SQL> begin
  2  dbms_sqldiag.import_sql_testcase
  3  (directory=>'TEST_DIR2',
  4  filename=>'oratcb1_008602000001main.xml',
  5  importData=>TRUE
```

```
  6  );
  7  end;
  8  /

PL/SQL procedure successfully completed.

SQL>
```

The user TEST will now have all the objects to execute the SQL statement that you want to investigate. You can verify this by issuing the original `select` statement. It should give you the same output as under the SH schema.

How It Works

Oracle offers the SQL Test Case Builder (TCB) to reproduce a SQL failure. You can create a test case through Enterprise Manager or through a PL/SQL package. The "Solution" section of this recipe shows how create a test case using the `EXPORT_SQL_TESTCASE` procedure of the `DBMS_SQLDIAG` package. There are several variants of this package, and our example shows how to use a SQL statement as the source for creating a SQL test case. Please review the `DBMS_SQLDIAG.EXPORT_TESTCASE` procedure in Oracle's PL/SQL Packages manual for details about other options to create test cases.

■ **Note** Remember that you should run the Test Case Builder as any user, other than SYS, who has been granted the DBA role.

Often, you'll find yourself trying to provide a test case for Oracle, without which the Oracle Support personnel won't be able to investigate a particular problem they are helping you with. The SQL Test Case Builder is a tool that is part of Oracle Database 11g, and its primary purpose is to help you quickly obtain a reproducible test case. The SQL Test Case Builder helps you easily capture pertinent information relating to a failed SQL statement and package it in a format that either a developer or an Oracle support person can use to reproduce the problem in a different environment.

You access the SQL Test Case Builder through the `DBMS_SQLDIAG` package. To create a test case, you must first export the SQL statement, including all the objects that are part of the statement, and all other related information. The export process is very similar to an Oracle export with the `EXPDP` command, and thus uses a directory, just as `EXPDP` does. Oracle creates the SQL test case as a script that contains the statements that will re-create the necessary database objects, along with associated runtime information such as statistics, which enable you to reproduce the error. The following are the various types of information captured and exported as part of the test case creation process:

- SQL text for the problem statement
- Table data—this is optional, and you can export a sample or complete data.
- The execution plan
- Optimizer statistics
- PL/SQL functions, procedure, and packages

- Bind variables
- User privileges
- SQL profiles
- Metadata for all the objects that are part of the SQL statement
- Dynamic sampling results
- Runtime information such as the degree of parallelism, for example

In the `DBMS_SQLDIAG` package, the `EXPORT_SQL_TESTCASE` procedure exports a SQL test case for a SQL statement to a directory. The `IMPORT_SQL_TESTCASE` procedure imports the test case from a directory.

In the `EXPORT_SQL_TESTCASE` procedure, here is what the attributes stand for:

`DIRECTORY`: The directory where you want to store the test case files

`SQL_TEXT`: The actual SQL statement that's throwing the error

`TESTCASE`: The name of the test case

`EXPORTDATA`: By default, Oracle doesn't export the data. You can set this parameter to `TRUE` in order to export the data. You can optionally limit the amount of data you want to export, by specifying a value for the Sampling Percent attribute. The default value is 100.

The Test Case Builder automatically exports the PL/SQL package specifications but not the package body. However, you can specify that the TCB export the package body as well. The export process creates several files in the directory you specify. Of these files, the file in the format `oratcb1_O08602000001main.xml` contains the metadata for the test case.

7-15. Generating an AWR Report

Problem

You'd like to generate an AWR report to analyze performance problems in your database.

Solution

The database automatically takes an AWR snapshot every hour, and saves the statistics in the AWR for eight days. An AWR report contains data captured between two snapshots, which need not be consecutive. Thus, an AWR report lets you examine instance performance between two points in time. You can generate an AWR report through Oracle Enterprise Manager. However, we show you how to create an AWR report using Oracle-provided scripts.

To generate an AWR report for a single instance database, execute the `awrrpt.sql` script as shown here.

```
SQL> @?/rdbms/admin/awrrpt.sql

Current Instance
~~~~~~~~~~~~~~~~

   DB Id    DB Name      Inst Num Instance
----------- ------------ -------- ------------
 1118243965 ORCL1               1 orcl1
Specify the Report Type
~~~~~~~~~~~~~~~~~~~~~~~
Would you like an HTML report, or a plain text report?
Enter 'html' for an HTML report, or 'text' for plain text
Defaults to 'html'
Enter value for report_type: text

Type Specified:
```

Select a text- or an HTML-based report. The HTML report is the default report type, and it provides a nice-looking, well-formatted, easy-to-read report. Press Enter to select the default HTML-type report.

```
Instances in this Workload Repository schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   DB Id     Inst Num DB Name      Instance     Host
------------ -------- ------------ ------------ ------------
* 1118243965        1 ORCL1        orcl1        MIROPC61

Using 1118243965 for database Id
Using          1 for instance number
```

You must specify the DBID for the database at this point. In our example, however, there's only one database and therefore one DBID, so there's no need to enter the DBID.

```
Specify the number of days of snapshots to choose from
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entering the number of days (n) will result in the most recent
(n) days of snapshots being listed.  Pressing <return> without
specifying a number lists all completed snapshots.
Enter value for num_days: 1
```

Enter the number of days for which you want the database to list the snapshot IDs. In this example, we chose 1 because we want to generate an AWR for a time period that falls in the last day.

```
Listing the last day's Completed Snapshots
                                                        Snap
Instance     DB Name        Snap Id    Snap Started    Level
------------ ------------ --------- ------------------ -----
orcl1        ORCL1             1877 17 Apr 2011 00:00      1
                               1878 17 Apr 2011 07:47      1
```

Specify a beginning and an ending snapshot for the AWR report.

```
Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Enter value for begin_snap: 1877
Begin Snapshot Id specified: 1877

Enter value for end_snap: 1878
End Snapshot Id specified: 1878
```

You can either accept the default name for the AWR report by pressing Enter, or enter a name for the report.

```
Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~

The default report file name is awrrpt_1_1877_1878.txt.  To use this name,
press <return> to continue, otherwise enter an alternative.

Enter value for report_name:
Using the report name awrrpt_1_1877_1878.html
```

The database generates an AWR report in the same directory from which you invoked the `awrrpt.sql` script. For example, if you choose an HTML-based report, the AWR report will be in the following format: `awrrpt_1_1881_1882.html`.

■ **Tip** You can generate an AWR report to analyze the performance of a single SQL statement by executing the `awrsqrpt.sql` script.

How It Works

The AWR reports that you generate show performance statistics captured between two points in time, each of which is called a snapshot. You can gain significant insights into your database performance by reading the AWR reports carefully. An AWR report takes less than a minute to run in most cases, and holds a treasure trove of performance information. The report consists of multiple sections. Walking through an AWR report usually shows you the reason that your database isn't performing at peak levels.

To generate an AWR report for all instances in an Oracle RAC environment, use the `awrgrpt.sql` script instead. You can generate an AWR report for a specific instance in a RAC environment by using the `awrrpti.sql` script. You can also generate an AWR report for a single SQL statement by invoking the `awrsqrpt.sql` script and providing the `SQL_ID` of the SQL statement.

You can generate an AWR report to span any length of time, as long as the AWR has snapshots covering that period. By default, the AWR retains its snapshots for a period of eight days.

You can generate an AWR snapshot any time you want, either through the Oracle Enterprise Manager or by using the `DBMS_WORKLOAD_REPOSITORY` package. That ability is useful when, for example, you want to investigate a performance issue from 20 minutes and the next snapshot is 40 minutes away. In that case, you must either wait 40 minutes or create a manual snapshot. Here's an example that shows how to create an AWR snapshot manually:

```
SQL> exec dbms_workload_repository.create_snapshot();

PL/SQL procedure successfully completed.
SQL>
```

Once you create the snapshot, you can run the `awrrpt.sql` script. Then select the previous two snapshots to generate an up-to-date AWR report.

You can generate an AWR report when your user response time increases suddenly, say from one to ten seconds during peak hours. An AWR report can also be helpful if a key batch job is suddenly taking much longer to complete. Of course, you must check the system CPU, I/O, and memory usage during the period as well, with the help of operating system tools such as `sar`, `vmstat`, and `iosat`.

If the system CPU usage is high, that doesn't necessarily mean that it's the CPU that's the culprit. CPU usage percentages are not always a true measure of throughput, nor is CPU usage always useful as a database workload metric. Make sure to generate the AWR report for the precise period that encompasses the time during which you noticed the performance deterioration. An AWR report that spans a 24-hour period is of little use in diagnosing a performance dip that occurred 2 hours ago for only 30 minutes. Match your report to the time period of the performance problem.

7-16. Comparing Database Performance Between Two Periods

Problem

You want to examine and compare how the database performed during two different periods.

Solution

Use the `awrddrpt.sql` script (located in the `$ORACLE_HOME/rdbms/admin` directory) to generate an AWR Compare Periods Report that compares performance between two periods. Here are the steps.

1. Invoke the `awrddrpt.sql` script.

   ```
   SQL> @$ORACLE_HOME/rdbms/admin/awrddrpt.sql
   ```

2. Select the report type (default is text).

   ```
   Enter value for report_type: html
   Type Specified:  html
   ```

3. Specify the number of days of snapshots from which you want to select the beginning and ending snapshots for the first time period.

   ```
   Specify the number of days of snapshots to choose from
   Enter value for num_days: 4
   ```

4. Choose a pair of snapshots over which you want to analyze the first period's performance.

```
Specify the First Pair of Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Enter value for begin_snap: 2092
First Begin Snapshot Id specified: 2092

Enter value for end_snap: 2093
First End   Snapshot Id specified: 2093
```

5. Select the number of days of snapshots from which you want to select the pair of snapshots for the second period. Enter the value 4 so you can select a pair of snapshots from the previous 4 days.

```
Specify the number of days of snapshots to choose from
Enter value for num_days: 4
```

6. Specify the beginning and ending snapshots for the second period.

```
Specify the Second Pair of Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Enter value for begin_snap2: 2134
Second Begin Snapshot Id specified: 2134

Enter value for end_snap2: 2135
Second End   Snapshot Id specified: 2135
```

7. Specify a report name or accept the default name.

```
Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~

The default report file name is awrdiff_1_2092_1_2134.html  To use this name,
press <return> to continue, otherwise enter an alternative.

Enter value for report_name:
Using the report name awrdiff_1_2092_1_2134.html
Report written to awrdiff_1_2092_1_2134.html
SQL>
```

How It Works

Generating an AWR Compare Periods report is a process very similar to the one for generating a normal AWR report. The big difference is that the report does not show what happened between two snapshots, as the normal AWR report does. The AWR Compare Periods report compares performance between two different time periods, with each time period involving a different pair of snapshots. If you want to compare the performance of your database between 9 a.m. and 10 a.m. today and the same time period three days ago, you can do it with the AWR Compare Periods report. You can run an AWR Compare Periods report on one or all instances of a RAC database.

The AWR Compare Periods report is organized similarly to the normal AWR report, but it shows each performance statistic for the two periods side by side, so you can quickly see the differences (or similarities) in performance. Here's a section of the report showing how you can easily review the differences in performance statistics between the first and the second periods.

```
                                First               Second       Diff
                       ---------------     ---------------     ------
  % Blocks changed per Read:                61.48               15.44    -46.04
            Recursive Call %:               98.03               97.44     -0.59
 Rollback per transaction %:                 0.00                0.00      0.00
              Rows per Sort:                 2.51                2.07     -0.44
 Avg DB time per Call (sec):                 1.01                0.03     -0.98
```

7-17. Analyzing an AWR Report

Problem

You've generated an AWR report that covers a period when the database was exhibiting performance problems. You want to analyze the report.

Solution

An AWR report summarizes its performance-related statistics under various sections. The following is a quick summary of the most important sections in an AWR report.

Session Information

You can find out the number of sessions from the section at the very top of the AWR report, as shown here:

```
Snap Id      Snap Time      Sessions Curs/Sess
            --------- ------------------- -------- ---------
Begin Snap:      1878 17-Apr-11 07:47:33        38       1.7
  End Snap:      1879 17-Apr-11 09:00:48        34       3.7
   Elapsed:               73.25 (mins)
   DB Time:               33.87 (mins)
```

Be sure to check the Begin Snap and End Snap times, to confirm that the period encompasses the time when the performance problem occurred. If you notice a very high number of sessions, you can investigate if shadow processes are being created—for example, if the number of sessions goes up by 200 between the Begin Snap and End Snap times when you expect the number of sessions to be the same at both times, the most likely cause is an application startup issue, which is spawning all those sessions.

Load Profile

The load profile section shows the per-second and per-transaction statistics for various indicators of database load such as hard parses and the number of transactions.

```
Load Profile              Per Second    Per Transaction
~~~~~~~~~~~~         ---------------    ---------------
      DB Time(s):                0.5               1.4
       DB CPU(s):                0.1               0.3
       Redo size:            6,165.3          19,028.7
   Logical reads:              876.6           2,705.6
   Block changes:               99.2             306.0
  Physical reads:               10.3              31.8
 Physical writes:                1.9               5.9
      User calls:                3.4              10.4
          Parses:               10.2              31.5
     Hard parses:                1.0               3.0
          Logons:                0.1               0.2
    Transactions:                0.3
```

The Load Profile section is one of the most important parts of an AWR report. Of particular significance are the physical I/O rates and hard parses. In an efficiently performing database, you should see mostly soft parses and very few hard parses. A high hard parse rate usually is a result of not using bind variables. If you see a high per second value for logons, it usually means that your applications aren't using persistent connections. A high number of logons or an unusually high number of transactions tells you something unusual is happening in your database. However, the only way you'll know the numbers are unusual is if you regularly check the AWR reports and know what the various statistics look like in a normally functioning database, at various times of the day!

Instance Efficiency Percentages

The instance efficiency section shows several hit ratios as well as the "execute to parse" and "latch hit" percentages.

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Buffer Nowait %:  100.00       Redo NoWait %:  100.00
            Buffer  Hit   %:   99.10    In-memory Sort %:  100.00
            Library Hit   %:   95.13        Soft Parse %:   90.35
        Execute to Parse %:   70.71         Latch Hit %:   99.97
 Parse CPU to Parse Elapsd %: 36.71     % Non-Parse CPU:   83.60

 Shared Pool Statistics        Begin    End
                              ------  ------
             Memory Usage %:   81.08   88.82
    % SQL with executions>1:   70.41   86.92
  % Memory for SQL w/exec>1:   69.60   91.98
```

The *execute to parse ratio* should be very high in a well-running instance. A low value for the `% SQL with exec>1` statistic means that the database is not re-using shared SQL statements, usually because the SQL is not using bind variables.

Top 5 Foreground Events

The Top 5 Timed Foreground Events section shows the events that were responsible for the most waits during the time spanned by the AWR report.

```
Top 5 Timed Foreground Events
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                    Avg Wait % DB
Event                                 Waits     Time(s)   (ms)   time Wait Class
----------------------------- ------------ ----------- ------ ------ ----------
db file sequential read             13,735         475     35   23.4 User I/O
DB CPU                                             429          21.1
latch: shared pool                     801          96    120    4.7 Concurrenc
db file scattered read                 998          49     49    2.4 User I/O
control file sequential read         9,785          31      3    1.5 System I/O
```

The Top 5 Timed Foreground Events section is where you can usually spot the problem, by showing you why the sessions are "waiting." The Top 5 Events information shows the total waits for all sessions, but usually one or two sessions are responsible for most of the waits. Make sure to analyze the total waits and average waits (ms) separately, in order to determine if the waits are significant. Merely looking at the total number of waits or the total wait time for a wait event could give you a misleading idea about its importance. You must pay close attention to the average wait times for an event as well. In a nicely performing database, you should see CPU and I/O as the top wait events, as is the case here. If any wait events from the concurrent wait class such as latches show up at the top, investigate those waits further. For example, if you see events such as `enq: TX - row lock contention`, `gc_buffer_busy` (RAC), or `latch free`, it usually indicates contention in the database. If you see an average wait of more than 2 ms for the `log file sync` event, investigate the wait event further (Chapter 5 shows how to analyze various wait events). If you see a high amount of waits due to the db file sequential read or the db file scattered read wait events, there are heavy indexed reads (this is normal) or full table scans going on. You can find out the SQL statement and the tables involved in these read events in the AWR report.

Time Model Statistics

Time model statistics give you an idea about how the database has spent its time, including the time it spent on executing SQL statements as against parsing statements. If parsing time is very high, or if hard parsing is significant, you must investigate further.

```
Time Model Statistics                 DB/Inst: ORCL1/orcl1  Snaps: 1878-1879

Statistic Name                                       Time (s) % of DB Time
------------------------------------------ ------------------ ------------
sql execute elapsed time                           1,791.5         88.2
parse time elapsed                                   700.1         34.5
hard parse elapsed time                              653.7         32.2
```

Top SQL Statements

This section of the AWR report lets you quickly identify the most expensive SQL statements.

```
SQL ordered by Elapsed Time            DB/Inst: ORCL1/orcl1  Snaps: 1878-1879
-> Captured SQL account for   13.7% of Total DB Time (s):           2,032
-> Captured PL/SQL account for   19.8% of Total DB Time (s):           2,032

        Elapsed                  Elapsed Time
        Time (s)    Executions  per Exec (s)  %Total   %CPU    %IO    SQL Id
---------------- -------------- ------------- ------ ------ ------ -------------
           292.4              1        292.41   14.4    8.1   61.2 b6usrg82hwsas
…
```

You can generate an explain plan for the expensive SQL statements using the SQL ID from this part of the report.

PGA Histogram

The `PGA Aggregate Target Histogram` shows how well the database is executing the sort and hash operations—for example:

```
 PGA Aggr Target Histogram              DB/Inst: ORCL1/orcl1  Snaps: 1878-1879
-> Optimal Executions are purely in-memory operations

  Low     High
Optimal Optimal    Total Execs  Optimal Execs 1-Pass Execs M-Pass Execs
------- ------- -------------- ------------- ------------ ------------
     2K      4K         13,957        13,957            0            0
    64K    128K             86            86            0            0
   128K    256K             30            30            0            0
```

In this example, the database is performing all sorts and hashes optimally, in the PGA. If you see a high number of one-pass executions and even a few large multi-pass executions, that's an indication that the PGA is too small and you should consider increasing it.

How It Works

Analyzing an AWR report should be your first step when troubleshooting database performance issues such as a slow-running query. An AWR report lets you quickly find out things such as the number of connections, transactions per second, cache-hit rates, wait event information, and the SQL statements that are using the most CPU and I/O. It shows you which of your SQL statements are using the most resources, and which wait events are slowing down the database. Most importantly, probably, the report tells you if the database performance is unusually different from its typical performance during a given time of the day (or night). The AWR report sections summarized in the "Solution" section are only a small part of the AWR report. Here are some other key sections of the report that you must review when troubleshooting performance issues:

- Foreground Wait Events
- SQL Ordered by Gets
- SQL Ordered by Reads
- SQL Ordered by Physical Reads

- Instance Activity Stats
- Log Switches
- Enqueue Activity
- Reads by Tablespace, Datafile, and SQL Statement
- Segments by Table Scans
- Segments by Row Lock Waits
- Undo Segment Summary

Depending on the nature of the performance issue you're investigating, several of these sections in the report may turn out to be useful. In addition to the performance and wait statistics, the AWR report also offers advisories for both the PGA and the SGA. The AWR report is truly your best friend when you are troubleshooting just about any database performance issue. In a matter of minutes, you can usually find the underlying cause of the issue and figure out a potential fix. AWR does most of the work for you—you just need to know what to look for!

CHAPTER 8

Creating Efficient SQL

Structured Query Language is like any other programming language in that it can be coded well, coded poorly, and everywhere in between. Learning to create efficient SQL statements has been discussed in countless books. This chapter zeroes in on basic SQL coding fundamentals, and addresses some techniques to improve performance of your SQL statements. In addition, some emphasis is given to ramifications of poorly written SQL, along with a few common pitfalls to avoid in your SQL statements within your application.

Writing good SQL statements the first time is the best way to get good performance from your SQL queries. Knowing the fundamentals is the key to accomplishing the goal of good performance. This chapter focuses on the basics of the SQL language:

- `SELECT` statement
- `WHERE` clause
- Joining tables
- Subqueries
- Set operators

Then, we'll focus on basic techniques to improve performance of your queries, as well as help ensure your queries are not hindering the performance of other queries within your database. It's important to take the time to write efficient SQL statements the first time, which is easy to say, but tough to accomplish when balancing client requirements, budgets, and project timelines. However, if you adhere to basic coding practices and fundamentals, you can greatly improve the performance of your SQL queries.

■ **Note** Several times in this chapter, we make a distinction between ISO syntax and traditional Oracle syntax. Specifically, we do that with respect to join syntax. However, that distinction is a bit mis-stated. With the exception of Oracle's use of the `(+)` to indicate an outer join, all of Oracle's join syntax complies with the ISO SQL standard, so it is *all* ISO syntax. However, it is common in the field to refer to the more newly implemented syntax as "ISO syntax," and we follow that pattern in this chapter.

8-1. Retrieving All Rows from a Table

Problem

You need to write a query to retrieve all rows from a given table within your database.

Solution

Within the SQL language, you use the `SELECT` statement to retrieve data from the database. Everything following the `SELECT` statement tells Oracle what data you need from the database. The first thing you need to determine is from which table(s) you need to retrieve data. Once this has been determined, you have what you need to be able to run a query to get data from the database. If we have an `EMPLOYEES` table within our Oracle database, we can perform a describe on that table in order to see the structure of the table. By doing this, we can see the column names for the table, and can determine which columns we want to select from the database.

```
SQL> describe employees
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 EMPLOYEE_ID                               NOT NULL NUMBER(6)
 FIRST_NAME                                         VARCHAR2(20)
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 EMAIL                                     NOT NULL VARCHAR2(25)
 PHONE_NUMBER                                       VARCHAR2(20)
 HIRE_DATE                                 NOT NULL DATE
 JOB_ID                                    NOT NULL VARCHAR2(10)
 SALARY                                             NUMBER(8,2)
 COMMISSION_PCT                                     NUMBER(2,2)
 MANAGER_ID                                         NUMBER(6)
 DEPARTMENT_ID                                      NUMBER(4)
```

If we want to retrieve a list of all the employees' names from our `EMPLOYEES` table, we now have all the information we need to assemble a simple query against the `EMPLOYEES` table in the database. We know we are selecting from the `EMPLOYEES` table, which is needed for the `FROM` clause. We also know we want to select the names of the employees, which is needed to satisfy the `SELECT` clause. At this point, we can issue the following query against the database:

```
SELECT last_name, first_name
FROM employees;

LAST_NAME                 FIRST_NAME
------------------------- --------------------
Abel                      Ellen
Baer                      Hermann
Cabrio                    Anthony
Dilly                     Jennifer
Ernst                     Bruce
```

If we want to select all columns from the EMPLOYEES table, we can list every column from the table in the SELECT clause, or we can substitute listing every column with the asterisk, which indicates that we want to retrieve all the columns:

```
SELECT *
FROM employees;
```

If our manager wants the format of the output to be a comma-delimited list of all the employees' names, we can modify our query to accomplish this task:

```
SELECT last_name || ', ' || first_name AS "Employee Name"
FROM employees;

Employee Name
-----------------------------------------------
Abel, Ellen
Baer, Hermann
Cabrio, Anthony
Dilly, Jennifer
Ernst, Bruce
```

In the foregoing case, we placed the concatenation characters, which are comprised of two vertical bars, in the query to indicate that we are combining the contents of multiple columns into a single output column. At the same time, we are creating a column alias by using the AS clause, and calling the combined last and first names "Employee Name".

How It Works

SELECT is the most fundamental statement needed to retrieve data from an Oracle database. While there are many clauses and features of a SELECT statement, at its most basic, there are really only two clauses needed to first retrieve data out of an Oracle database—and those clauses are the SELECT clause and the FROM clause. Normally, more is required to accurately retrieve the desired result set. You may want only a subset of the columns within a database table, and you may want only a subset of rows from a given table. Furthermore, you may want to perform manipulation on data pulled from the database. All this requires more sophisticated components of the SQL language than the simple SELECT statement. However, the SELECT and FROM clauses are the basic building blocks to assemble a query, from the most simple of queries to the most complex of queries.

■ **Note** In order to select data from any database table, you need to either own the table or have been given the privilege to select data from the given set of tables.

8-2. Retrieve a Subset of Rows from a Table

Problem

You want to filter the data from a database `SELECT` query to return only a subset of rows from a database table.

Solution

The `WHERE` clause gives the user the ability to filter rows and return only the desired result set back from the database. There are various ways to construct a `WHERE` clause, a few of which will be reviewed within this recipe. The first thing that occurs within a `WHERE` clause is that one or more columns' values are compared to some other value. See Table 8-1 for a list of comparison operators that can be used within the `WHERE` clause. One of the more common comparison operators is the equal sign, which denotes an equality condition:

```
SELECT *
FROM EMP
WHERE deptno = 20;
```

In the foregoing query, we are selecting all columns from the `EMP` table, which is denoted by using the asterisk, and we want only those rows for department 20, which is determined by the `WHERE` clause.

Table 8-1. *Comparison Operators Used in the* `WHERE` *Clause*

Operator	Description
=	Equal to
!= , <> , ^=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
`IS NULL`	Checking for existence of null values
`IS NOT NULL`	
`LIKE`	Used to search when entire column value is not known
`NOT LIKE`	

How It Works

In many SQL statements, there can be multiple conditions in a `WHERE` clause. Coding multiple conditions is done by using the logical operators `OR`, `AND`, and `IN`. If you have multiple logical operators within your SQL statement, Oracle will first always evaluate all `AND` clauses prior to any of the `OR` clauses. If there are matching logical operators in the same statement, they are evaluated from left to right. This can be confusing when constructing a complex `WHERE` condition. Therefore, when coding multiple conditions within a `WHERE` clause, delimit each clause with parentheses, else you may not get the results you are expecting. This is good SQL coding practice, and makes SQL code simpler to read and maintain—for example:

```
SELECT last_name, first_name, salary, email
FROM employees
WHERE (department_id = 20
OR department_id = 30)
AND commission_pct > 0;
```

If we have the need for multiple `OR` logical operators within our statement, we can replace them with the `IN` logical operator, which can simplify our SQL statement. By rewriting the foregoing query to use the `IN` logical operator, our query would look like the following:

```
SELECT last_name, first_name, salary, email
FROM employees
WHERE department_id IN (20,30)
AND commission_pct > 0;
```

If you want to find all the same information for all departments except department 20 or 30, the SQL code would look like the following:

```
SELECT last_name, first_name, salary, email
FROM employees
WHERE (department_id != 20
AND department_id <> 30)
AND commission_pct > 0;
```

Note that in the foregoing, for demonstration, we used two of the "not equal" comparison operators. It is generally good coding practice to be consistent, and use the same operators across all of your SQL code. This avoids confusion with others who need to look at or modify your SQL code. Even subtle differences like this can make someone else ponder why one piece of SQL code was done one way, and another piece of SQL code was done a different way. When writing SQL code, writing for efficiency is important, but it is equally important to write the code with an eye on maintainability. If SQL code is consistent, it simply will be easier to read and maintain.

Taking the previous SQL statement, we again will use the logical `OR` operator, add the `NOT` operand, and accomplish the same task:

```
SELECT last_name, first_name, salary, email
FROM employees
WHERE department_id NOT IN(20,30)
AND commission_pct > 0;
```

The last two queries both provided the proper results, but in this case, using the `IN` clause simplified our SQL statement.

8-3. Joining Tables with Corresponding Rows

Problem

Within a single query, you wish to retrieve matching rows from multiple tables. These tables have at least one common column on which to match the data between the tables.

Solution

A join condition within the SQL language is used to combine data from multiple tables within a single query. When there are corresponding rows in all tables involved in the join process, it is called an "inner join." What this means is that based on the common join columns between the tables, only data that matches between the two tables will be returned.

Let's say you want to get the city where all departments in your company are based. There are two different ways to approach this before writing your SQL statement. You can use either traditional Oracle syntax, or the newer ISO syntax. Using traditional Oracle SQL, the syntax would be as follows:

```
SELECT d.location_id, department_name, city
FROM departments d, locations l
WHERE d.location_id = l.location_id;
```

To write the same statement using ISO syntax, there are several methods that can be used:

- Natural Join
- JOIN ... USING clause
- JOIN ... ON clause

If using the `NATURAL JOIN` clause, you are letting Oracle determine the natural join condition and which columns will be joined on, and therefore there are no join clauses or conditions in the statement:

```
SELECT location_id, department_name, city
FROM departments NATURAL JOIN locations;
```

If tables you are joining have common named join columns, you can specify the `JOIN ... USING` clause, and you specify this common column within parentheses:

```
SELECT location_id, department_name, city
FROM departments JOIN locations
USING (location_id);
```

It is very common for the join condition between tables to have differently named columns that are needed to complete the join criteria. In these cases, the `JOIN ... ON` clause is appropriate:

```
SELECT d.loc_id, department_name, city
FROM departments d JOIN locations l
ON l.location_id = d.loc_id;
```

How It Works

When using traditional Oracle SQL, you need to specify all join conditions in the `WHERE` clause. Therefore, the `WHERE` clause will contain all join conditions, along with any filtering criteria.

With ISO SQL, a key advantage is that the join conditions are done in the `FROM` clause, and the `WHERE` clause is used solely for filtering criteria. This makes SQL statements easier to read and decipher. No longer do you need to determine within a `WHERE` clause which statements are join conditions and which are filtering criteria. The advantage of this is more evident when you are joining three or more tables. The filtering criteria are solely in the `WHERE` clause and are easily visible:

```
SELECT last_name, first_name, department_name, city,
state_province state, postal_code zip, country_name
FROM employees
JOIN departments USING (department_id)
JOIN locations USING (location_id)
JOIN countries USING (country_id)
JOIN regions USING (region_id)
WHERE department_id = 20;
```

If you prefer to write SQL statements with traditional Oracle SQL, good practice just for readability and more maintainable SQL code is to place all join conditions first in the `WHERE` clause, and place all filtering criteria at the end of the `WHERE` clause. It also makes the code easier to read and maintain if you can simply line up the code. This is an optional practice, but helps anyone else who may need to look at your SQL code:

```
SELECT last_name, first_name, department_name, city,
state_province state, postal_code zip, country_name
FROM employees e, departments d, locations l, countries c, regions r
WHERE e.department_id = d.department_id
  AND d.location_id   = l.location_id
  AND l.country_id    = c.country_id
  AND c.region_id     = r.region_id
  and d.department_id = 20;
```

Also, when using the `JOIN ... ON` or `JOIN ... USING` clause, it may be more clear to specify the optional `INNER` keyword, as it would immediately be known it is an inner join that is being done:

```
SELECT location_id, department_name, city
FROM departments INNER JOIN locations
USING (location_id);
```

8-4. Joining Tables When Corresponding Rows May Be Missing

Problem

You need data from two or more tables, but some of the data is missing in one or more of the tables. For instance, you want to get a list of all of the departments for your company, along with their base

locations. For whatever reason, you've been told that there are locations listed within your company that do not map to a single department, in which case there are no department locations listed.

Solution

You need to show all locations, so an inner join will not work in this case. Instead, you can write what is termed an *outer join*. Notice the (+) syntax in the following example:

```
SELECT l.location_id, city, department_id, department_name
FROM locations l, departments d
WHERE l.location_id = d.location_id(+)
ORDER BY 1;

LOCATION_ID CITY                 DEPARTMENT_ID DEPARTMENT_NAME
----------- -------------------- ------------- -------------------------
       1100 Venice
       1400 Southlake                       60 IT
       1500 South San Francisco             50 Shipping
       1700 Seattle                        170 Manufacturing
       1700 Seattle                        240 Sales
       1700 Seattle                        270 Payroll
       1700 Seattle                        120 Treasury
       1700 Seattle                        110 Accounting
       1700 Seattle                        100 Finance
       1700 Seattle                         30 Purchasing
       1800 Toronto                         20 Marketing
       2000 Beijing
       2400 London                          40 Human Resources
       2700 Munich                          70 Public Relations
       3200 Mexico City
```

To specify an outer join using traditional Oracle SQL, simply place a plus sign within parentheses in the WHERE clause join condition next to a column from the table that you know has missing data. We know in the foregoing case that there are locations that are not assigned to a single department. From the results, we can see the locations that have no departments assigned to them.

To execute the same query using ISO SQL syntax, you use the LEFT OUTER JOIN or RIGHT OUTER JOIN clauses, which can be shortened to LEFT JOIN or RIGHT JOIN—for example:

```
SELECT location_id, city, department_id, department_name
FROM locations LEFT JOIN departments d
USING (location_id)
ORDER BY 1;
```

Now let's say you must execute a query in which either table, on either side of the join, could be missing one or more corresponding rows. One approach is to create a union of two outer join queries:

```
SELECT last_name, first_name, department_name
FROM employees e, departments d
WHERE e.manager_id = d.manager_id(+)
UNION
SELECT last_name, first_name, department_name
FROM employees e, departments d
```

```
WHERE e.manager_id(+) = d.manager_id
ORDER BY department_name, last_name, first_name;


LAST_NAME                 FIRST_NAME           DEPARTMENT_NAME
------------------------- -------------------- ------------------------
Gietz                     William              Accounting
                                               Administration
                                               Benefits
Cambrault                 Gerald               Executive
Chen                      John                 Finance
                                               Government Sales
                                               Human Resources
Pataballa                 Valli                IT
                                               Manufacturing
Fay                       Pat                  Marketing
                                               Payroll
                                               Public Relations
Tobias                    Sigal                Purchasing
Tucker                    Peter                Sales
                                               Shareholder Services
Sarchand                  Nandita              Shipping
                                               Treasury
Lee                       Linda
Morse                     Steve
```

From the foregoing results, we can see all employees that manage departments, all employees that do not manage departments, as well as those departments with no assigned manager.

In order to do the same query using ISO SQL syntax, use the `FULL OUTER JOIN` clause, which can be shortened to `FULL JOIN`:

```
SELECT last_name, first_name, department_name
FROM employees FULL JOIN departments
USING (manager_id);
```

How It Works

There are really three outer joins that can be done based on your circumstances. Table 8-2 describes all the possible join conditions. SQL statements using traditional syntax or ISO SQL syntax are both perfectly acceptable. However, it is generally easier to write, read, and maintain ISO SQL than traditional Oracle SQL.

One of the main advantages of the ISO syntax is that for multiple table joins, all the join conditions are specified in the `FROM` clause, and are therefore isolated and easy to see. In Oracle SQL, the join conditions are specified in the `WHERE` clause, along with any other filtering criteria needed for the query. If you inherited poorly structured SQL code, it is simply harder to read longer and more complex SQL statements that have join conditions and filtering criteria interspersed within a single `WHERE` clause.

One other type of join not already mentioned is the cross join, which is a Cartesian join. While this type of join is rarely useful, it can be occasionally beneficial. As a DBA, let's say you are gathering database size information for your enterprise of databases, and are placing the results in a single spreadsheet. You need to get database and host information for each query. You can execute the following query:

```
  SELECT d.name, i.host_name, round(sum(f.bytes)/1048576) megabytes
  FROM v$database d
  CROSS JOIN v$instance i
  CROSS JOIN v$datafile f
  GROUP BY d.name, i.host_name;

NAME      HOST_NAME                      MEGABYTES
--------- ------------------------------ ----------
ORCL      DREGS-PC                             2333
```

In this case, the `v$instance` and `v$database` views contain only a single row, so there is no harm in doing a Cartesian join. The foregoing join could also be written with traditional Oracle SQL:

```
SELECT d.name, i.host_name, round(sum(f.bytes)/1048576) megabytes
FROM v$database d, v$instance i, v$datafile f
GROUP BY d.name, i.host_name;
```

Table 8-2. *Oracle Join Conditions*

Join Type	Traditional Join Syntax	ISO Join Syntax	Description

Inner join	`WHERE` clause, with one clause specified for each join condition	`FROM` clause, along with:
		`NATURAL JOIN`
		`JOIN ... USING`
		`JOIN … ON`
			There are corresponding rows in each table matching the condition.

Left outer join	`WHERE` clause, the 3-character sequence of (+) placed next to column from table with missing data	`FROM` clause, along with:
		`LEFT OUTER JOIN`
		`LEFT JOIN`
			There may not be corresponding rows in the table on the right side of the join condition.

Right outer join	`WHERE` clause, the 3-character sequence of (+) placed next to table with missing data	`FROM` clause, along with:
		`RIGHT OUTER JOIN`
		`RIGHT JOIN`
			This means there may not be corresponding rows on table on the left side of the join condition.

Full outer join	Two `SELECT` statements with union condition specified, with one side of the outer join specified on first part of the union, and the other side of the outer join specified on second part of the union	`FROM` clause, along with:
		`FULL OUTER JOIN`
		`FULL JOIN`
			This means there may not always be corresponding rows in both tables. It cannot be specified natively with traditional Oracle SQL syntax, and must be constructed with a `UNION`, while ISO SQL has the syntax built in.

Cross join	`WHERE` clause; there are no join conditions between the joined tables.	`FROM` clause, along with: `CROSS JOIN`	This means a Cartesian join is indicated.

8-5. Constructing Simple Subqueries

Problem

You need to retrieve data from the database, but cannot get the data you need using a single query.

Solution

It is common that data needs from a relational database are complex enough that the data cannot be retrieved within a single SQL `SELECT` statement. Rather than having to run two or more queries serially, it is possible to construct several SQL `SELECT` statements and place them within a single query. These additional `SELECT` statements are called subqueries, sub-selects, or nested selects.

Let's say you want to get the name of the employee with the highest salary in your company so you can ask your boss for a raise. Since you don't know what the highest salary is, you first have to run a query to determine the following:

```
SELECT MAX(salary) FROM employees;

MAX(SALARY)
-----------
      24000
```

Then, knowing what the highest salary is, you could run a second query to get the employee(s) with that salary:

```
SELECT last_name, first_name
FROM employees
WHERE salary = 24000;

LAST_NAME                 FIRST_NAME
------------------------- --------------------
King                      Steven
```

It's very simple to combine the foregoing two queries, and construct a single SQL statement with a subquery to accomplish the same task:

```
SELECT last_name, first_name
FROM employees
WHERE salary =
(SELECT MAX(salary) FROM employees);

LAST_NAME                 FIRST_NAME
------------------------- --------------------
King                      Steven
```

How It Works

Within a SQL statement, a subquery can be placed within the `SELECT`, `WHERE`, or `HAVING` clauses. You can also place a query within the `FROM` clause, which is also called an inline view, which is addressed in a different recipe. There are several kinds of subqueries that can be constructed:

- Single-row or scalar subquery
- Multiple-row subquery
- Multiple-column subquery
- Correlated subquery (addressed in a different recipe)

The subquery itself is also called an *inner query*. Except for correlated subqueries, the inner query is executed first, and then the results of the inner query are passed to the outer query, which is then executed.

Single-Row Subqueries

Single-row subqueries return a single column of a single row. The example shown in the "Solution" section is a single-row subquery. Use caution and be certain that the subquery can return only a single value; otherwise you can get an error from your subquery:

```
SELECT last_name, first_name
FROM employees
WHERE salary =
(SELECT salary FROM employees WHERE department_id = 30);

(SELECT salary FROM employees WHERE department_id = 30)
 *
ERROR at line 4:
ORA-01427: single-row subquery returns more than one row
```

In the foregoing example, there are multiple employees in department 30, so the subquery would return all of the matching rows.

If you want to see how your salary stacks up against the average salaries of employees in your company, you can issue a subquery in the `SELECT` clause to accomplish this:

```
SELECT last_name, first_name, salary, ROUND((SELECT AVG(salary) FROM employees)) avg_sal
FROM employees
WHERE last_name = 'King';

LAST_NAME                 FIRST_NAME               SALARY    AVG_SAL
------------------------- -------------------- ---------- ----------
King                      Steven                    24000       6462
```

Let's say you want to know which departments overall had a higher salary than the average for your company. By placing the subquery in the `HAVING` clause, you can get the desired results:

```
column avg_sal format 99999.99

SELECT department_id, ROUND(avg(salary),2) avg_sal
```

```
FROM employees
GROUP BY department_id
HAVING avg(salary) > (SELECT AVG(salary) FROM employees)
ORDER BY 2;

DEPARTMENT_ID   AVG_SAL
------------- ---------
           40   6500.00
          100   8600.00
           80   8955.88
           20   9500.00
           70  10000.00
          110  10150.00
           90  19333.33

8 rows selected.
```

Multiple-Row Subqueries

If you know the desired subquery is going to return multiple rows, you can use the `IN`, `ANY`, `ALL`, and `SOME` operators. The `IN` operator is the same as having multiple `OR` conditions in a select statement. For example, in the following SQL statement, we are getting the `DEPARTMENT_NAME` for departments 20, 30, and 40.

```
SELECT department_id, department_name
FROM departments
WHERE department_id = 20
OR department_id = 30
OR department_id = 40;

DEPARTMENT_ID DEPARTMENT_NAME
------------- ------------------------------
           20 Marketing
           30 Purchasing
           40 Human Resources
```

Using the `IN` operator, we can simplify our SQL statement and achieve the same result:

```
SELECT department_id, department_name
FROM departments
WHERE department_id IN (20,30,40);
```

The `ANY` and `SOME` operators function identically. They are used to compare a value retrieved from the database to each value shown in the list of values in the query. They are used with the comparison operators =, !=, <>, <, <=, >, or >=. Use care with `ANY` or `SOME`, as it evaluates each value separately, without regard to the entire list of values. For example, using the same query to get the department name for departments 20, 30, or 40, if we modify this query to use `ANY` or `SOME`, we can see how Oracle evaluates each value in the `ANY` clause. Because we used the `ANY` clause, departments 10, 20, and 30 were included in the result, even though departments 20 and 30 were within our `ANY` clause. This is because each value is evaluated separately before the result set is returned.

```
SELECT department_id, department_name
FROM departments
WHERE department_id < ANY (20,30,40);

SELECT department_id, department_name
FROM departments
WHERE department_id < SOME (20,30,40);

DEPARTMENT_ID DEPARTMENT_NAME
------------- ------------------------------
           10 Administration
           20 Marketing
           30 Purchasing
```

The `ALL` operator essentially uses a logical `AND` operator to do the comparison of values shown in the query. While with the `ANY` operator, each value was compared individually to see if there was a match, the `ALL` operator needs to compare every value in the list before determining if there is a match. Using our department table as an example, see the following query. In this query, we are retrieving the department names from the table if the `DEPARTMENT_ID` value is less than or equal to *all* values in the list:

```
SELECT department_id, department_name
FROM departments
WHERE department_id <= ALL (20,30,40);

DEPARTMENT_ID DEPARTMENT_NAME
------------- ------------------------------
           10 Administration
           20 Marketing
```

Multiple-Column Subqueries

At times, you need to match data based on multiple columns. If placed within the `WHERE` clause, the column list needs to be placed within parentheses. As an example, if you want to get a list of the employees with the highest salary in their respective departments, you can write a multiple-column subquery such as the following:

```
SELECT last_name, first_name, department_id, salary
FROM employees
WHERE (department_id, salary) IN
(SELECT department_id, max(salary)
FROM employees
GROUP BY department_id)
ORDER BY department_id;

LAST_NAME                 FIRST_NAME           DEPARTMENT_ID     SALARY
------------------------- -------------------- ------------- ----------
Whalen                    Jennifer                        10       4400
Hartstein                 Michael                         20      13000
Raphaely                  Den                             30      11000
Mavris                    Susan                           40       6500
Fripp                     Adam                            50       8200
```

```
Hunold                    Alexander                  60       9000
Baer                      Hermann                    70      10000
Russell                   John                       80      14000
King                      Steven                     90      24000
Greenberg                 Nancy                     100      12000
Higgins                   Shelley                   110      12000

11 rows selected.
```

8-6. Constructing Correlated Subqueries

Problem

You are writing a subquery to retrieve data from a given set of tables from your database, but in order to retrieve the proper results, you really need to reference the outer query from inside the inner query.

Solution

The correlated subquery is a powerful component of the SQL language. The reason it is called "correlated" is that it allows you to reference the outer query from within the inner query. For example, we want to see all the jobs each current employee has ever held in the company:

```
SELECT employee_id, job_id
FROM job_history h
WHERE job_id in
(SELECT job_id FROM employees e
WHERE e.job_id = h.job_id)
ORDER BY 1;

EMPLOYEE_ID JOB_ID
----------- ----------
        101 AC_ACCOUNT
        101 AC_MGR
        102 IT_PROG
        114 ST_CLERK
        122 ST_CLERK
        176 SA_REP
        176 SA_MAN
        200 AD_ASST
        200 AC_ACCOUNT
        201 MK_REP

10 rows selected.
```

How It Works

Because you reference the outer query from inside the inner query, the process of executing a correlated subquery is essentially the opposite compared to a simple subquery. In a correlated subquery, the outer query is executed first, as the inner query needs the data from the outer query in order to be able to process the query and retrieve the results. The steps to execute a correlated subquery are as follows. These steps repeat for each row in the outer query:

1. Retrieve row from the outer query.
2. Execute the inner query.
3. The outer query compares the value returned from the inner query.
4. If there is a value match in step 3, the row is returned to the user.

Another type of correlated subquery is to use the `EXISTS` clause in a subquery. When you use `EXISTS`, a test is done to see if the inner query returns at least one row. This is the important test that occurs when using the `EXISTS` operator. As you can see from the following example, the column list of the `SELECT` clause within the inner query is irrelevant. Something is included there simply to have proper SQL syntax only. If we want to see which of our employees are also managers, we can use the `EXISTS` operator with a self-join back to the employees table to determine this information:

```
SELECT employee_id, last_name, first_name
FROM employees e
WHERE EXISTS
(SELECT 'ANY LITERAL WILL DO HERE'
FROM employees m
WHERE e.employee_id = manager_id);

EMPLOYEE_ID LAST_NAME                 FIRST_NAME
----------- ------------------------- --------------------
        100 King                      Steven
        101 Kochhar                   Neena
        102 De Haan                   Lex
        103 Hunold                    Alexander
        108 Greenberg                 Nancy
        114 Raphaely                  Den
        120 Weiss                     Matthew
        121 Fripp                     Adam
        122 Kaufling                  Payam
        123 Vollman                   Shanta
        124 Mourgos                   Kevin
        145 Russell                   John
        146 Partners                  Karen
        147 Errazuriz                 Alberto
        148 Cambrault                 Gerald
        149 Zlotkey                   Eleni
        201 Hartstein                 Michael
        205 Higgins                   Shelley

18 rows selected.
```

You can also use `NOT EXISTS` if you want to test the opposite condition within a query. For example, your CEO wants to determine the manager-to-employee ratio within your company. Using the query from the previous example, we can first use the `EXISTS` operator to determine the number of managers within the company:

```
SELECT count(*)
FROM employees e
WHERE EXISTS
(SELECT 'TESTING 1,2,3'
FROM employees m
WHERE e.employee_id = manager_id);

  COUNT(*)
----------
        18
```

If we convert `EXISTS` to `NOT EXISTS`, we can determine the number of non-managers within the company:

```
SELECT count(*)
FROM employees e
WHERE NOT EXISTS
(SELECT 'X'
FROM employees m
WHERE e.employee_id = manager_id);

 COUNT(*)
---------
       89
```

8-7. Comparing Two Tables to Finding Missing Rows

Problem

You need to compare data for a subset of columns between two tables. You need to find rows in one table that are missing from the other.

Solution

You can use the Oracle `MINUS` set operator to compare two sets of data, and show data missing from one of the tables. When using any of the Oracle set operators, the `SELECT` clauses must be identical in terms of number of columns, and the datatypes of each column.

As an example, you work for a cable television company, and you want to find out what channels are offered free of charge. To test this out, you could first simply get a list of the channels offered by your company:

```
SELECT channel_id FROM channels;

CHANNEL_ID
----------
         2
         3
         4
         5
         9
```

Then, you can run a query to find out which channels have costs associated with them by querying the `COSTS` table:

```
SELECT DISTINCT channel_id FROM costs
ORDER BY channel_id;

CHANNEL_ID
----------
         2
         3
         4
```

By quickly doing a visual examination of the results, the free channels are channels 5 and 9. By using a set operator, in this case, `MINUS`, you can get this result from a single query:

```
SELECT channel_id
FROM channels
MINUS
SELECT channel_id
FROM costs;

CHANNEL_ID
----------
         5
         9
```

How It Works

It is also very common to use set operators in queries to get more information about the missing data. For instance, you have gotten the free channel list, but you really need to get more information about those free channels, and would like to accomplish everything within a single query:

```
SELECT channel_id, channel_desc FROM channels
WHERE channel_id IN
(SELECT channel_id
FROM channels
MINUS
SELECT channel_id
FROM costs);
```

```
CHANNEL_ID CHANNEL_DESC
---------- --------------------
         5 Catalog
         9 Tele Sales
```

8-8. Comparing Two Tables to Finding Matching Rows

Problem

You need to compare data for a subset of columns between two tables. You need to see all matching rows from those tables.

Solution

You can use the Oracle `INTERSECT` set operator to compare two sets of data, and show the matching data between the two tables. Again, when using any of the Oracle set operators, the `SELECT` clauses must be identical in terms of number of columns, and the datatypes of each column.

Using the example of the free channels, we now want to see which channels are not free, and have costs associated with them. By using the `INTERSECT` set operator, we will see only the matching rows between the two tables:

```
SELECT channel_id
FROM channels
INTERSECT
SELECT channel_id
FROM costs;

CHANNEL_ID
----------
         2
         3
         4
```

How It Works

When using `INTERSECT`, think of it as the overlapping data between two tables, based on the column list in the `SELECT` statement.

8-9. Combining Results from Similar SELECT Statements

Problem

You need to combine the results between two similar `SELECT` statements, and would like to accomplish it within a single query.

Solution

You can use the Oracle set operators `UNION` or `UNION ALL` to combine results from two like queries. The difference between using `UNION` and `UNION ALL` is that `UNION` will automatically eliminate any duplicate rows, and each row of the result set will be unique. When using `UNION ALL`, it will show all matching rows, including duplicate rows. Using `UNION ALL` may yield better performance than `UNION`, because a sort to eliminate duplicates is avoided. If your application can eliminate duplicates during processing, it may be worth the performance gained from using `UNION ALL`.

In Oracle's sample schemas, we have the `SCOTT.EMP` table and the `HR.EMPLOYEES` table. If we want to see all the employees on both tables, we can use a `UNION` set operator to get the results:

```
SELECT empno, hiredate FROM scott.emp
UNION
SELECT employee_id, hire_date FROM hr.employees;

     EMPNO HIREDATE
---------- ---------
       100 17-JUN-87
       101 21-SEP-89
       102 13-JAN-93
       ...
      7902 03-DEC-81
      7934 23-JAN-82
      7997 15-AUG-11

122 rows selected.
```

How It Works

You are running two queries where you have a nearly identical column list, but let's say you have one additional column on one table. In this case, we have the `COMM` column on the `SCOTT.EMP` table, which is the commission amount an employee has earned. You don't have an equivalent column on the `HR.EMPLOYEES` table. By using `NULL` in the missing column, you can still use a set operator such as `UNION` as long as you account for any missing columns on either side of the operation:

```
SELECT empno, mgr, hiredate, sal, deptno, comm
FROM scott.emp
UNION
SELECT employee_id, manager_id, hire_date, salary, department_id, NULL
FROM hr.employees;
```

```
     EMPNO        MGR HIREDATE          SAL     DEPTNO       COMM
---------- ---------- --------- ---------- ---------- ----------
       100            17-JUN-87      24000         90
       101        100 21-SEP-89      17000         90
       102        100 13-JAN-93      17000         90
       ...
      7369       7902 17-DEC-80        800         20
      7499       7698 20-FEB-81       1600         30        300
      7521       7698 22-FEB-81       1250         30        500
      7566       7839 02-APR-81       2975         20
      7654       7698 28-SEP-81       1250         30       1400
```

After examining the `HR.EMPLOYEES` table, there is a column named `COMMISSION_PCT`. We can derive the actual commission based on this column, and add it to the previous query. Also, our manager has told us that he or she wants to see a value in the commission column for all employees, even if they earn no commission:

```
SELECT empno, mgr, hiredate, sal, deptno, nvl(comm,0)
FROM scott.emp
UNION
SELECT employee_id, manager_id, hire_date, salary, department_id,
nvl(salary*commission_pct/100,0)
FROM hr.employees;
```

```
     EMPNO        MGR HIREDATE          SAL     DEPTNO NVL(COMM,0)
---------- ---------- --------- ---------- ---------- -----------
       100            17-JUN-87      24000         90           0
       101        100 21-SEP-89      17000         90           0
       102        100 13-JAN-93      17000         90           0
       ...
       147        100 10-MAR-97      12000         80          36
       148        100 15-OCT-99      11000         80          33
       149        100 29-JAN-00      10500         80          21
       ...
      7499       7698 20-FEB-81       1600         30         300
      7521       7698 22-FEB-81       1250         30         500
      7566       7839 02-APR-81       2975         20           0
```

One point to stress again is that the datatypes for each column also must be the same. For example, we are doing a union between the `SCOTT.EMP` table and the `HR.DEPARTMENTS` table, and want to see a combined list of the department numbers, along with their locations. However, based on the datatype list, we cannot use an Oracle set operator such as `UNION` for this, as the location column for each table is different:

```
SQL> desc scott.dept
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------

 DEPTNO                                    NOT NULL NUMBER(2)
 DNAME                                              VARCHAR2(14)
 LOC                                                VARCHAR2(13)

SQL> desc hr.departments
```

```
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------

 DEPARTMENT_ID                             NOT NULL NUMBER(4)
 DEPARTMENT_NAME                           NOT NULL VARCHAR2(30)
 MANAGER_ID                                         NUMBER(6)
 LOCATION_ID                                        NUMBER(4)

SQL> l
  1  SELECT deptno, loc FROM scott.dept
  2  UNION
  3* select department_id, location_id from hr.departments
SQL> /
SELECT deptno, loc FROM scott.dept
               *
ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
```

8-10. Searching for a Range of Values

Problem

You need to retrieve data from your database based on a range of values for a given column.

Solution

The `BETWEEN` clause is commonly used to retrieve a range of values from a database. It is most commonly used with dates, timestamps, and numbers, but can also be used with alphanumeric data. It is an efficient way of retrieving data from the database when an exact set of values is not known for a column within the `WHERE` clause. For instance, if we wanted to see all employees that were hired between the year 2000 and through the year 2010, the query could be written as follows:

```
SELECT last_name, first_name, hire_date
FROM employees
WHERE hire_date BETWEEN '2000-01-01' and '2010-12-31'
ORDER BY hire_date;
```

When using the `BETWEEN` clause, it is an efficient way to find a range of values for a column, and works for a multitude of datatypes. If you want to get a range of values for a `NUMBER` datatype as in the `SALARY` column, a range can be given:

```
SELECT last_name, first_name, salary
FROM employees
WHERE salary BETWEEN 20000 and 30000
ORDER BY salary;
```

If you want to add to the foregoing query and get only those employees whose last names are in the first half of the alphabet, you can supply a range to satisfy this request. In order to guarantee all values, we filled out the possible values to the 25-character length of the `last_name` column:

```
SELECT last_name, first_name, salary
FROM employees
WHERE salary BETWEEN 20000 and 30000
AND last_name BETWEEN 'Aaaaaaaaaaaaaaaaaaaaaaa'
AND 'Mzzzzzzzzzzzzzzzzzzzzzzz'
ORDER BY salary;
```

How It Works

One common pitfall when using the `BETWEEN` clause is with the use of date-based columns, whether it be the `DATE` datatype, the `TIMESTAMP` datatype, or any other date-based datatype. If not constructed carefully, desired rows can be missed from the result set.

One way this occurs is that often queries on dates are done using a combination of year, month, and day. It is important to remember that even though the format of the date-based fields on an Oracle database usually defaults to a year, month, and day type of format, the element of time must always be accounted for, else rows can be missed from a query. In this first example, we have an employee, Sarah Bell, who was hired February 4, 1996:

```
SELECT hire_date FROM employees
WHERE email = 'SBELL';

HIRE_DATE
----------
1996-02-04
```

If we query the database, and don't consider the time element for any date column, we can omit critical rows from our result set. Therefore it is important to know whether the time portion of the column is included in the makeup of the data. In this case, there is indeed a time element present in the `hire_date` column:

```
SELECT last_name, first_name, hire_date
FROM employees
WHERE hire_date = '1996-02-04';

no rows selected
```

Sometimes, when rows are inserted into the database, the time portion of a date or timestamp can be truncated. However, when coding efficient SQL, it is important to always assume there is a time element present for any and all date-based columns. Based on that assumption, we can modify the foregoing query to consider the time element in the `hire_date` column:

```
SELECT last_name, first_name, to_char(hire_date,'yyyy-mm-dd:hh24:mi:ss') hire_date
FROM employees
WHERE hire_date
BETWEEN TO_DATE('1996-02-04:00:00:00','yyyy-mm-dd:hh24:mi:ss')
AND TO_DATE('1996-02-04:23:59:59','yyyy-mm-dd:hh24:mi:ss');

LAST_NAME                 FIRST_NAME           HIRE_DATE
------------------------- -------------------- -------------------
Bell                      Sarah                1996-02-04:12:30:46
```

Here is a similar case, where we are performing a SELECT to retrieve all data for a given month specified in the query. In this case, we are retrieving all employees who were hired in the month of September, 1997. If we omit the time element from the BETWEEN clause, we can actually omit data that meets the criteria for our query:

```
SELECT last_name, first_name, hire_date
FROM employees
WHERE hire_date
BETWEEN '1997-09-01' and '1997-09-30';

LAST_NAME                 FIRST_NAME           HIRE_DATE
------------------------- -------------------- ----------
Chen                      John                 1997-09-28


SELECT last_name, first_name, hire_date
FROM employees
WHERE hire_date
BETWEEN TO_DATE('1997-09-01:00:00:00','yyyy-mm-dd:hh24:mi:ss')
AND TO_DATE('1997-09-30:23:59:59','yyyy-mm-dd:hh24:mi:ss');

LAST_NAME                 FIRST_NAME           HIRE_DATE
------------------------- -------------------- ----------
Chen                      John                 1997-09-28
Sciarra                   Ismael               1997-09-30
```

If you are using a BETWEEN clause in your query, and there is an index on the column specified in the WHERE clause, the Oracle optimizer can use the index to retrieve the data. You would need to perform an explain plan to validate if this is the case, but using BETWEEN means an index can often be used if one is present, and can be an efficient manner of selecting data from the database using a range of values:

```
SELECT last_name, first_name, salary
FROM employees
WHERE last_name between 'Ba' and 'Bz'
ORDER BY salary;

-------------------------------------------------
| Id  | Operation                    | Name        |
-------------------------------------------------
|   0 | SELECT STATEMENT             |             |
|   1 |  SORT ORDER BY               |             |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |
|   3 |    INDEX RANGE SCAN          | EMP_NAME_IX |
-------------------------------------------------
```

8-11. Handling Null Values

Problem

You have null values in some of your database data, and need to understand the ramifications of dealing with null values in data. You also need to write queries to correctly deal with such nulls.

Solution

Null values have to be dealt with in a certain manner, depending on whether you are searching for null values in your data in the `SELECT` clause, or you are attempting to make a determination of what to do when a null value is found in the `WHERE` clause.

Handling Nulls in the SELECT Clause

Within the `SELECT` clause, if you are dealing with data within a column that contains null values, there are two Oracle-provided functions you can use within SQL to transform a null value into a more usable form. The two functions are `NVL` and `NVL2`.

■ **Note** Actually, there are more than just the two functions `NVL` and `NVL2`. However, those are widely used, and are a good place to begin.

With `NVL`, you simply pass in the column name, along with the value you want to give the output based on whether that value is null in the database. For instance, in our employees table, not all employees get a commission based on their jobs, and the value in that column for these employees is null:

```
SELECT ename , sal , comm
FROM emp
ORDER BY ename;

ENAME             SAL       COMM
---------- ---------- ----------
ADAMS            1100
ALLEN            1600        300
BLAKE            2850
KING             5000
MARTIN           1250       1400
```

If we simply want to see a zero in the commission column for employees not eligible for a commission, we can use the `NVL` function to accomplish this:

```
SELECT ename , sal , NVL(comm,0) comm
FROM emp
ORDER BY ename;

ENAME             SAL       COMM
---------- ---------- ----------
ADAMS            1100          0
ALLEN            1600        300
BLAKE            2850          0
KING             5000          0
MARTIN           1250       1400
```

If we decide to perform arithmetic on a null value, the result will always be null; therefore if we want to compute "Total Compensation" as salary plus commission, we must apply the `NVL` function to properly compute this with consideration of the null values. In the following example, we compute the sum of these columns, both with and without the `NVL` function. Without using `NVL`, we get an incorrect result, which can be seen in the `TOTAL_COMP_NO_NVL` output field:

```
SELECT ename , sal , nvl(comm,0) comm, sal+comm total_comp_no_nvl,
       sal+NVL(comm,0) total_comp_nvl
FROM emp
ORDER BY ename;

ENAME             SAL       COMM TOTAL_COMP_NO_NVL TOTAL_COMP_NVL
---------- ---------- ---------- ----------------- --------------
ADAMS            1100          0                              1100
ALLEN            1600        300              1900           1900
BLAKE            2850          0                              2850
KING             5000          0                              5000
MARTIN           1250       1400              2650           2650
```

The `NVL2` is similar to `NVL`, except that `NVL2` takes in three arguments—the value or column, the value to return if the column is not null, and finally the value to return if the column is null. For instance, if we use the same foregoing example when determining if an employee gets a commission, we simply want to assign a value to each employee stating whether he or she is a "commissioned" or "non-commissioned" employee. We can accomplish this with the `NVL2` function:

```
SELECT ename , sal ,
NVL2(comm,'Commissioned','Non-Commissioned') comm_status
FROM emp
ORDER BY ename;

ENAME             SAL COMM_STATUS
---------- ---------- ---------------
ADAMS            1100 Non-Commissioned
ALLEN            1600 Commissioned
BLAKE            2850 Non-Commissioned
KING             5000 Non-Commissioned
MARTIN           1250 Commissioned
```

Handling Nulls in the WHERE Clause

Within the `WHERE` clause, if you simply want to check a column to see if it contains a null value, use `IS NULL` or `IS NOT NULL` as the comparison operator—for example:

```
SELECT ename , sal
FROM emp
WHERE comm IS NULL
ORDER BY ename;

ENAME             SAL
---------- ----------
ADAMS            1100
BLAKE            2850
KING             5000


SELECT ename , sal
FROM emp
WHERE comm IS NOT NULL
ORDER BY ename;

ENAME             SAL
---------- ----------
ALLEN            1600
MARTIN           1250
```

You can also use the `NVL` or `NVL2` function in the `WHERE` clause just as it was used in the `SELECT` statement:

```
SELECT ename , sal
FROM emp
WHERE NVL(comm,0) = 0
ORDER BY ename;

ENAME             SAL
---------- ----------
ADAMS            1100
BLAKE            2850
KING             5000
```

How It Works

It is best to always explicitly handle the possibility of null values, so if a column of a table is nullable, assume nulls exist, else output results can be undesired or unpredictable. One quick check that can be made to determine if a column has null values is to compare a count of rows in the table (`COUNT *`) to a count of rows for that column (`COUNT <column_name>`). A count on a nullable column will count only those rows that do not have null values. Here's an example:

```
SELECT count(*) FROM emp;

  COUNT(*)
----------
        14

SELECT count(comm) FROM emp;

COUNT(COMM)
-----------
          4
```

This technique of comparing row count to a count of values in a column is a handy way to check if nulls exist in a column.

Another very useful function that can be used in the handling of null values is the `COALESCE` function. With `COALESCE`, you can pass in a series of values, and the function will return the first non-`NULL` value. If all values within `COALESCE` are `NULL`, a `NULL` value is returned. Here is a simple example:

```
SELECT coalesce(NULL,'ABC','DEF') FROM dual;

COA
---
ABC
```

Let's say you wanted to get the shipping address for your customers, and if none were present, you would then get the billing address. Using `COALESCE`, you could achieve this as shown in the following example:

```
SELECT COALESCE(
(SELECT shipping_address FROM customers
WHERE cust_id = 9342),
(SELECT billing_address FROM customers
WHERE cust_id = 9342))
FROM dual;
```

All arguments used in a statement with `COALESCE` must be with the same datatype, else you will receive an error, as shown here:

```
SELECT coalesce(NULL,123,'DEF') FROM dual;
                     *
ERROR at line 1:
ORA-00932: inconsistent datatypes: expected NUMBER got CHAR
```

8-12. Searching for Partial Column Values

Problem

You need to search for a string from a column in the database, but do not know the exact value of the column data.

Solution

When you are unsure of the data values in the columns you are filtering on in your `WHERE` clause, you can utilize the `LIKE` operator. Unlike the normal comparison operators such as the equal sign, the `BETWEEN` clause, or the `IN` clause, the `LIKE` operator allows you to search for matches based on a partial string of the column data. When you use the `LIKE` clause, you need to also use the "%" symbol or the "_" symbol within the data itself in order to search for the data you need. The percent sign is used to replace one to many characters. For example, if you want to see the list of employees that were hired in 1995, regardless of the exact date, the `LIKE` clause can be used to search for any matches within `hire_date` that contain the string `1995`. When using `LIKE` with a date or timestamp datatype, you need to ensure that the date format you are using is compatible with your search criteria in your `LIKE` statement. For instance, if the default date format for your database is `DD-MON-YY`, then the string `1995` is not compatible with that format and a match would never be found. In order to search in this manner, set your date format within your session before issuing your query:

```
alter session set nls_date_format = 'yyyy-mm-dd';

Session altered.

SELECT employee_id, last_name, first_name, hire_date
FROM employees
WHERE hire_date LIKE '%1995%'
ORDER BY hire_date;

EMPLOYEE_ID LAST_NAME                 FIRST_NAME           HIRE_DATE
----------- ------------------------- -------------------- ----------
        122 Kaufling                  Payam                1995-05-01
        115 Khoo                      Alexander            1995-05-18
        137 Ladwig                    Renske               1995-07-14
        141 Rajs                      Trenna               1995-10-17
```

An easy way to remedy having to worry about the date format of your session is to simply use the `TO_CHAR` function within the query. The advantage of this method is it is very easy to code, without having to worry about your session's date format. See the following example:

```
SELECT employee_id, last_name, first_name, hire_date
FROM employees
WHERE to_char(hire_date,'yyyy') = '1995'
ORDER BY hire_date;
```

The underscore symbol ("_") is used to replace exactly one character. Let's say you were looking for an employee that had a last name of "Olsen" or "Olson," but were unsure of the spelling. In a single query, you can use the underscore in conjunction with the `LIKE` clause to find all employees with that name variation in your database:

```
SELECT last_name, first_name, phone_number
FROM employees
WHERE last_name like 'Ols_n';

LAST_NAME                 FIRST_NAME           PHONE_NUMBER
------------------------- -------------------- --------------------
Olsen                     Christopher          011.44.1344.498718
Olson                     TJ                   650.124.8234
```

How It Works

The `LIKE` clause is extremely useful for finding data within your database when you are unsure of the exact column values stored within the data. There are performance ramifications that need to be considered when using the `LIKE` clause. The primary consideration is that when the `LIKE` clause is used, the chances of the optimizer using an index to aid in retrieving the data are reduced. Since an index is based on a complete value for a column, having to search for only a portion of the complete value of a column is problematic for the optimizer to be able to use an index.

Using our foregoing example of finding employees that started during the year 1995, here is the explain plan for that query:

```
---------------------------------------
| Id  | Operation         | Name      |
---------------------------------------
|   0 | SELECT STATEMENT  |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
---------------------------------------
```

Since an index is based on an entire value, the optimizer can recognize that an index can be used if the first part of the value is intact in the search criteria. By placing the percent signs on both sides of the value, it is the same as saying "contains." If we place the percent sign on only the trailing end of the value, it is the same as "starts with." Since the leading edge of the value is intact, the optimizer will be able to effectively compare the value based on the value in the `LIKE` clause with an existing index, and can therefore use such an index, if one is present on that column:

```
SELECT employee_id, last_name, first_name, hire_date
FROM employees
WHERE hire_date LIKE '1995%';

EMPLOYEE_ID LAST_NAME                 FIRST_NAME           HIRE_DATE
----------- ------------------------- -------------------- ----------
        115 Khoo                      Alexander            1995-05-18
        122 Kaufling                  Payam                1995-05-01
        137 Ladwig                    Renske               1995-07-14
        141 Rajs                      Trenna               1995-10-17

-------------------------------------------------
| Id  | Operation                   | Name        |
-------------------------------------------------
|   0 | SELECT STATEMENT            |             |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |
|   2 |   INDEX RANGE SCAN          | EMP_NAME_IX |
-------------------------------------------------
```

Sometimes it is very possible for an underscore to be part of the data that is being searched. In these cases, it is important to preface the underscore with the escape character. If you are a DBA, and are searching for a tablespace name in your database, which easily can contain the underscore character, make sure you consider that underscore is a wildcard symbol, and must be considered. See the following example:

```
SELECT tablespace_name FROM dba_tablespaces
WHERE tablespace_name like '%EE_DATA';

TABLESPACE_NAME
------------------------------
EMPLOYEE_DATA
EMPLOYEE1DATA
```

It is very possible that the underscore is searched as data, not as a substitution character for the `LIKE` clause. If you insert an escape character within the query, you can avoid getting undesired results. By inserting the escape character directly in front of the underscore, then the underscore will be considered as part of the data, rather than a substitution character:

```
SELECT tablespace_name FROM dba_tablespaces
WHERE tablespace_name LIKE '%EE^_DATA' ESCAPE '^';

TABLESPACE_NAME
------------------------------
EMPLOYEE_DATA
```

The benefit of the `LIKE` clause is the flexibility it gives you in finding data based on a partial value of the column data. The likely trade-off is performance. Queries using the `LIKE` clause are often much less likely to use an index. As an alternative to `LIKE`, the `BETWEEN` clause, although not as simple to code within your SQL statement, can generally be more likely to use an index. Sometimes, however, the `LIKE` clause can be perceived as a clause to avoid because of the performance ramifications, but if the leading percent sign is avoided, often the optimizer will use an index, if available.

In the `TO_CHAR` example of the "Solution" section, you will note that the `TO_CHAR` function is placed on the left side of the comparison operator. Generally, when this occurs, it means no index on the filtering column will be used (see Recipe 8-14 for more discussion on this topic). However, with certain Oracle functions and the manner in which they are translated, an index still may be used. The only way to be certain is to simply run an explain plan on your query. For our foregoing query using `TO_CHAR`, it still used an index even though the function was placed on the left side of the comparison operator:

```
SELECT employee_id, last_name, first_name, hire_date
FROM employees
WHERE to_char(hire_date,'yyyy') = '1995'
ORDER BY hire_date;


-------------------------------------------------
| Id  | Operation                   | Name         |
-------------------------------------------------
|   0 | SELECT STATEMENT            |              |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |
|   2 |   INDEX FULL SCAN           | EMPLOYEES_I1 |
-------------------------------------------------
```

8-13. Re-using SQL Statements Within the Shared Pool

Problem

You are getting an excessive amount of hard-parsing for your SQL statements, and want to lower the number of SQL statements that go through the hard parse process.

Solution

Implementing bind variables within an application can tremendously improve the efficiency and performance of queries. Essentially, bind variables are called substitution variables, and replace literals within a query. By placing bind variables within your SQL statements, the statements can be re-used in memory, and do not have to go through the entire expensive SQL parsing process.

Here is an example of a normal SQL statement, with literal values shown in the `WHERE` clause:

```
SELECT employee_id, last_name || ', ' || first_name employee_name
FROM employees
WHERE employee_id = 115;

EMPLOYEE_ID EMPLOYEE_NAME
----------- ------------------------
        115 Khoo, Alexander
```

There are a couple of ways to define bind variables within Oracle. First, you can simply use SQL Plus. To accomplish this within SQL Plus, you first need to define a variable, along with a datatype to the variable. Then, you can use the `exec` command, which actually will run a PL/SQL command to populate the variable with the desired value. Notice that when referencing a bind variable in SQL Plus, it is prefaced with a colon:

```
SQL> variable g_emp_id number
SQL> exec :g_emp_id := 115;

PL/SQL procedure successfully completed.
```

After you have defined a variable and assigned a value to it, you can simply substitute the variable name within your SQL statement. Again, since it is a bind variable, you need to preface it with a colon:

```
SELECT employee_id, last_name || ', ' || first_name employee_name
FROM employees
WHERE employee_id = :g_emp_id;

EMPLOYEE_ID EMPLOYEE_NAME
----------- ------------------------
        115 Khoo, Alexander
```

You can also assign variables within PL/SQL. The nice advantage of PL/SQL is that just by using variables in PL/SQL, they are automatically bind variables, so there is no special coding required. And, unlike SQL Plus, no colon is required when referencing a variable that was defined within the PL/SQL block:

```
SQL> set serveroutput on
  1  DECLARE
  2    v_emp_id employees.employee_id%TYPE := 200;
  3    v_last_name employees.last_name%TYPE;
  4    v_first_name employees.first_name%TYPE;
  5  BEGIN
  6    SELECT last_name, first_name
  7    INTO v_last_name, v_first_name
  8    FROM employees
  9    WHERE employee_id = v_emp_id;
 10  dbms_output.put_line('Employee Name = ' || v_last_name || ', ' || v_first_name);
 11* END;
SQL> /
Employee Name = Whalen, Jennifer
```

How It Works

When bind variables are used, their use increases the likelihood that a SQL statement can be re-used within the shared pool. Oracle uses a hashing algorithm to assign a value to every unique SQL statement. If literals are used within a SQL statement, the hash values between two otherwise identical statements will be different. By using the bind variables, the statements will have the same hash value within the shared pool, and part of the expensive parsing process can be avoided.

Re-use Is Efficient

Re-use is efficient because Oracle does not have to go through the entire parsing process for those SQL statements. If you do not use bind variables within your SQL statements, and instead use literals, the statements need to be completely parsed.

See Table 8-3 for a review of the steps taken to process a SQL statement. A statement that is "hard-parsed" must execute all of the steps. If a statement is "soft parsed," the optimizer generally does not execute the optimization and row source generation steps.

Table 8-3. *Steps to Execute a SQL Statement*

Step	Description
Syntax checking	Determines if SQL statement is syntactically correct
Semantic checking	Determines if objects referenced in SQL statement exist and user has proper privileges to those objects
Check shared pool	Oracle uses hashing algorithm to generate hash value for SQL statement and checks shared pool for existence of that statement in the shared pool.
Optimization	The Oracle optimizer chooses what it perceives as the best execution plan for the SQL statement based on gathered statistics.

Continued

Step	Description
Row source generation	This is an Oracle program that received the execution plan from the optimization step and generates a query plan. When you generate an explain plan for a statement, it shows the detailed query plan.
Execution	Each step of the query plan is executed, and the result set is returned to the user.

Hard-Parsing Can Be Avoided

By using bind variables, a hard parse can be avoided and can help the performance of SQL queries, as well as reduce the amount of memory thrashing that can occur in the shared pool. The `TKPROF` utility is one way to verify whether SQL statements are being re-used in the shared pool. Later, there are examples of PL/SQL code that use bind variables, and PL/SQL code that does not use bind variables.

By using the `TKPROF` utility, we can see how these statements are processed. In order to see this information with the `TKPROF` utility, we first must turn tracing on within our session:

```
alter session set sql_trace=true;
```

The trace file gets generated in the location specified by the `diagnostic_dest` or `user_dump_dest` parameter settings. The following PL/SQL block updates the employees table and gives all employees a 3% raise. Since all PL/SQL variables are treated as bind variables, we can see with the `TKPROF` output that the update statement was parsed only once, but executed 107 times:

```
BEGIN
FOR i IN 100..206
LOOP
UPDATE employees
SET salary=salary*1.03
WHERE employee_id = i;
END LOOP;
COMMIT;
END;
```

Here is an excerpt from the `TKPROF`-generated report, which summarizes information about the session on which tracing was enabled:

```
SQL ID : f7mtnudzhm2py
UPDATE EMPLOYEES SET SALARY=SALARY*1.03
WHERE
 EMPLOYEE_ID = :B1
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute    107      0.01       0.00          0        107        112         107
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      108      0.01       0.00          0        107        112         107
```

In the following example, we do the same thing using dynamic SQL with the `execute immediate` command:

```
BEGIN
FOR i IN 100..206
LOOP
execute immediate 'UPDATE employees SET salary=salary*1.03 WHERE employee_id = ' || i;
END LOOP;
COMMIT;
END;
```

Since the entire statement is assembled together prior to execution, the variable is converted to a literal before execution. We can see with the `TKPROF` output that the statement was parsed with each execution:

```
SQL ID : 67776qbqqz5wc
UPDATE employees SET salary=salary*:"SYS_B_0"
WHERE
 employee_id = :"SYS_B_1"


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse      107      0.00       0.00          0          0          0           0
Execute    107      0.00       0.04          0        107        112         107
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      214      0.00       0.05          0        107        112         107
```

Bind Variables Are Usable with EXECUTE IMMEDIATE

If we want to use the `execute immediate` command more efficiently, we can convert that `execute immediate` command to use a bind variable with the `USING` clause, and specify a bind variable within the `execute immediate` statement. The result shows that the statement was parsed only one time:

```
BEGIN
FOR i IN 100..206
LOOP
execute immediate 'UPDATE employees SET salary=salary*1.03 WHERE employee_id = :empno' USING
i;
END LOOP;
COMMIT;
END;
```

```
SQL ID : 4yO9bqzjngvq4
update employees set salary=salary*1.03
where
 employee_id = :empno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute    107      0.01       0.01          0        107        112         107
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      108      0.01       0.01          0        107        112         107
```

■ **Tip** Hard-parsing always occurs for DDL statements.

8-14. Avoiding Accidental Full Table Scans

Problem

You have queries that should be using indexes, but instead are doing full table scans. You want to avoid doing full table scans when the optimizer could be using an index to retrieve the data.

Solution

When constructing a SQL statement, a fundamental rule to try to always observe, if possible, is to avoid using functions on the left side of the comparison operator. A function essentially turns a column into a literal value, and therefore the Oracle optimizer does not recognize that converted value as a column any longer, but as a value instead.

Here, we're trying to get a list of all the employees that started since the year 1999. Because we placed a function on the left side of the comparison operator, the optimizer is forced to do a full table scan, even though the `HIRE_DATE` column is indexed:

```
SELECT employee_id, salary, hire_date
FROM employees
WHERE TO_CHAR(hire_date,'yyyy-mm-dd') >= '2000-01-01';

---------------------------------------
| Id  | Operation         | Name      |
---------------------------------------
|   0 | SELECT STATEMENT  |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
---------------------------------------
```

By moving the function to the right side of the comparison operator and leaving HIRE_DATE as a pristine column in the WHERE clause, the optimizer can now use the index on HIRE_DATE:

```
SELECT employee_id, salary, hire_date
FROM employees
WHERE hire_date >= TO_DATE('2000-01-01','yyyy-mm-dd');

-----------------------------------------------
| Id  | Operation                   | Name      |
-----------------------------------------------
|   0 | SELECT STATEMENT            |           |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES |
|   2 |   INDEX RANGE SCAN          | EMP_I5    |
-----------------------------------------------
```

How It Works

Functions are wonderful tools to convert a value or return the desired value based on what you need from the database, but they can be a performance killer if used incorrectly within a SQL statement. Make sure all functions are on the right side of the comparison operator, and the optimizer will be able to use any indexes on columns specified in the WHERE clause. This rule holds true for any function. In certain cases, it is possible Oracle will still use an index even if a function is on the left side of the comparison operator, but this is usually the exception. See Recipe 8-12 for an example of this.

Keep in mind that the datatype for a given column in the WHERE clause may change how the SQL statement needs to be modified to move the function to the right side of the comparison operator. With the following example, we had to change the comparison operator in order to effectively move the function:

```
SELECT last_name, first_name
FROM employees
WHERE SUBSTR(phone_number,1,3) = '515';

---------------------------------------
| Id  | Operation         | Name      |
---------------------------------------
|   0 | SELECT STATEMENT  |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
---------------------------------------
```

In order to effectively get all numbers in the 515 area code, we can use a BETWEEN clause and capture all possible values. We can also use a LIKE clause, as long as the wildcard character is on the trailing end of the condition. By using either of these methods, the optimizer changed the execution plan to use an index:

```
SELECT last_name, first_name
FROM employees
WHERE phone_number BETWEEN '515.000.0000' and '515.999.9999';
```

```
SELECT last_name, first_name
FROM employees
WHERE phone_number LIKE'515%';


------------------------------------------------
| Id  | Operation                   | Name      |
------------------------------------------------
|   0 | SELECT STATEMENT            |           |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES |
|   2 |   INDEX RANGE SCAN          | EMP_I6    |
------------------------------------------------
```

8-15. Creating Efficient Temporary Views

Problem

You need a table or a view of data that does not exist to construct a needed query, and do not have the authority to create such a table or view on your database.

Solution

At times, within a single SQL statement, you want to create a table "on the fly" that is used solely for your query, and will never be used again. In the `FROM` clause of your query, you normally place the name of your table or view on which to retrieve the data. In cases where a needed view of the data does not exist, you can create a temporary view of that data with what is called an "inline view," where you specify the characteristics of that view right in the `FROM` clause of your query:

```
SELECT last_name, first_name, department_name dept, salary
FROM employees e join
       ( SELECT department_id, max(salary) high_sal
         FROM employees
         GROUP BY department_id ) m
USING (department_id) join departments
USING (department_id)
WHERE e.salary = m.high_sal
ORDER BY SALARY desc;

LAST_NAME                 FIRST_NAME           DEPT                    SALARY
------------------------- -------------------- -------------------- ----------
King                      Steven               Executive                24000
Russell                   John                 Sales                    14000
Hartstein                 Michael              Marketing                13000
Greenberg                 Nancy                Finance                  12000
Higgins                   Shelley              Accounting               12000
Raphaely                  Den                  Purchasing               11000
Baer                      Hermann              Public Relations         10000
Hunold                    Alexander            IT                        9000
```

```
Fripp                Adam            Shipping              8200
Mavris               Susan           Human Resources       6500
Whalen               Jennifer        Administration        4400
```

In the foregoing query, we are getting the employees with the highest salary for each department. There isn't such a view in our database. Moreover, there isn't a way to directly join the employees table to the departments table to retrieve this data within a single query. Therefore, the inline view is created as part of the SQL statement, and holds only the key information we needed—it has the high salary and department information, which now can easily be joined to the employees table based on the employee with that matching salary.

How It Works

Inline views, as with many components of the SQL language, need to be used carefully. While extremely useful, if misused or overused, inline views can cause database performance issues, especially in terms of the use of the temporary tablespace. Since inline views are created and used only for the duration of a query, their results are held in the program global memory area, and if too large, the temporary tablespace. Before using an inline view, the following questions should be considered:

1. Most importantly, how often will the SQL containing the inline view be run? (If only once or rarely, then it might be best to simply execute the query and not worry about any potential performance impact).
2. How many rows will be contained in the inline view?
3. What will the row length be for the inline view?
4. How much memory is allocated for the `pga_aggregate_target` or `memory_target` setting?
5. How big is the temporary tablespace that is used by your Oracle user or schema?

If you have a simple ad hoc query you are doing, this kind of analysis may not be necessary. If you are creating a SQL statement that will run in a production environment, it is important to perform this analysis, as if all the temporary tablespace is consumed by an inline view, it affects not only the completion of that query, but also the successful completion of any processing for any user that may use that specific temporary tablespace. In many database environments, there is only a single temporary tablespace. Therefore, if one user process consumes all the temporary space with a single operation, this affects the operations for every user in the database.

Consider the following query:

```
WITH service_info AS
(SELECT
product_id,
geographic_id,
sum(qty) quantity
FROM services
GROUP BY
product_id,
geographic_id),
product_info AS
(SELECT product_id, product_group, product_desc
```

```
FROM products
WHERE source_sys = 'BILLING'),
billing_info AS
(SELECT journal_date, billing_date, product_id
FROM BILLING
WHERE journal_date = TO_DATE('2011-08-15', 'YYYY-MM-DD'))
SELECT
product_group,
product_desc,
journal_date,
billing_date,
sum(service_info.quantity)
FROM service_info JOIN product_info
ON service_info.product_id = product_info.product_id JOIN billing_info
ON  service_info.product_id = billing_info.product_id
WHERE
service_info.quantity > 0
GROUP BY
product_group,
product_desc,
journal_date,
billing_date;
```

In this query, there are three inline views created: the `SERVICE_INFO` view, the `PRODUCT_INFO` view, and the `BILLING_INFO` view. Each of these queries will be processed and the results stored in the program global area or the temporary tablespace before finally processing the true end-user query, which starts with the final `SELECT` statement shown in the query. While efficient in that the desired results can be done by executing a single query, the foregoing query, depending on the size of the data within the tables, can be tremendously inefficient to process, as storing potentially millions of rows in the temporary tablespace uses critical resources needed by an entire community of users that use the database. In examples such as these, it is generally more efficient at the database level to create tables that hold the data defined by the inline views—in this case, three separate tables. Then, the final query can be extracted from joining the three permanent tables to generate the results. While this may be more upfront work by the development team and the DBA, it could very well pay dividends if the query is run on a regular basis. Furthermore, as complexity increases with a SQL statement, ease of maintenance decreases. So, overall, it is more efficient, and usually more maintainable, to break a complex statement into chunks.

Inline views provide great benefit. However, do the proper analysis and investigation prior to implementing the use of such a view in a production environment.

■ **Caution** Large inline views can easily consume a large amount of temporary tablespace✎

8-16. Avoiding the NOT Clause

Problem

You have queries that use the NOT clause that are not performing adequately, and wish to modify them to improve performance.

Solution

Just as often as we query our database for equality conditions, we will query our database for non-equality conditions. It is the nature of retrieving data from a database and the nature of the SQL language to allow users to do this.

There are performance drawbacks in using the NOT clause within your SQL statements, as they trigger full table scans. Here's an example query from a previous recipe:

```
SELECT last_name, first_name, salary, email
FROM employees_big
WHERE department_id NOT IN(20,30)
AND commission_pct > 0;

-------------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time  |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |   697K|    21M|  4480   (1)| 00:00:54 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES_BIG |   697K|    21M|  4480   (1)| 00:00:54 |
-------------------------------------------------------------------------------------
```

Even though we have an index on the department_id column, by using the NOT clause we cause Oracle to bypass the use of that index in order to properly search and ensure all rows were not those in department 20 or 30. Note the overall cost of 4480 that Oracle assigned to this query.

It is possible to enable the use of the index by rewriting the query. For instance, let's issue a subquery to get a list of all the department_id values that are not 20 or 30, and then pass that list to the parent query. By doing this, we are moving the NOT clause to the much smaller departments table, so the table scan on that table will be fast. Those values get passed the parent query, and the parent query can use an index because it no longer needs the NOT clause.

Here's the new query, and the resulting execution plan.

```
SELECT last_name, first_name, salary, email
FROM employees_big
WHERE department_id IN
(SELECT department_id FROM departments
WHERE department_id NOT IN (20,30))
AND commission_pct > 0;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |    33 |  1188 |     3   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |            |    33 |  1188 |     3   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES  |    34 |  1088 |     3   (0)| 00:00:01 |
|*  3 |   INDEX UNIQUE SCAN| DEPT_ID_PK |     1 |     4 |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Note now that after our change, the query now uses an index, and the overall cost that Oracle assigned dropped from 4480 to 3.

How It Works

You can effectively use the `NOT` clause several ways:

- Comparison operators ('<>', '!=', '^=')
- `NOT IN`
- `NOT LIKE`

By using `NOT`, each of the following queries has the same basic effect in that it will negate the use of any possible index on the columns to which `NOT` applies:

```
SELECT last_name, first_name, salary, email
FROM employees
WHERE department_id != 20
AND commission_pct > 0;

SELECT last_name, first_name, salary, email
FROM employees
WHERE department_id NOT IN(20,30)
AND commission_pct > 0;

SELECT last_name, first_name, salary, email
FROM employees
WHERE hire_date NOT LIKE '2%'
AND commission_pct > 0;

--------------------------------------
| Id  | Operation         | Name      |
--------------------------------------
|   0 | SELECT STATEMENT  |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
--------------------------------------
```

At times, a full table scan is simply required. Even if an index is present, if you need to read more than a certain percentage of rows of a table, the Oracle optimizer may perform a full table scan regardless of whether an index is present. Still, if you simply try to avoid using `NOT` where possible, you may be able to improve performance on your queries.

All this said, you can try to use `NOT EXISTS` as an alternative that may improve performance in these conditions. Using the foregoing query and modifying it to use `NOT EXISTS`, you can still use an index to improve performance of the query:

```
SELECT last_name, first_name, salary, email
FROM employees
WHERE NOT EXISTS
(SELECT department_id FROM departments
WHERE department_id in(20,30))
AND commission_pct > 0;

-----------------------------------------
| Id  | Operation           | Name       |
-----------------------------------------
|   0 | SELECT STATEMENT    |            |
|   1 |  FILTER             |            |
|   2 |   TABLE ACCESS FULL | EMPLOYEES  |
|   3 |   INLIST ITERATOR   |            |
|   4 |    INDEX UNIQUE SCAN| DEPT_ID_PK |
-----------------------------------------
```

8-17. Controlling Transaction Sizes

Problem

You are performing a series of DML activities, and want to better manage the units of work and the recoverability of your transactions.

Solution

With the use of savepoints, you can split up transactions more easily into logical chunks, and can manage them more effectively upon failure. With the use of savepoints, you can roll back a series of DML statements to an incremental savepoint you have created. Within your SQL session, simply create a savepoint at an appropriate place during your processing that allows you to more easily isolate a "logical unit of work." The following is an example showing how to create a savepoint:

```
SQL> savepoint A;

Savepoint created.
```

If you have an online bookstore, for instance, and you have a customer placing an online order, when he or she submits an order, a logical unit of work for this transaction would be as follows:

- Adding a row to the orders table
- Adding one to many rows to the orderitems table

When processing this online order, you will want to commit all the information for the order and all items for an order as one transaction. This represents multiple database DML statements, but needs to be processed one at a time to preserve the integrity of a customer order; therefore it can be regarded as one "logical unit of work." By using savepoints, you can more easily process multiple DML statements as logical units of work. When a savepoint is created, it is essentially creating an alias based on a system change number (SCN). After creating a savepoint, you then have the luxury to roll back a transaction to that SCN based on the savepoint you created.

How It Works

Let's say your company has established two new departments, as well as employees for those departments. You need to insert rows in the corresponding `DEPT` and `EMP` tables, but you need to do this in one transaction per department. In case of an error, you can roll back to the point the last logical transaction completed. First, we can see a current picture of the `DEPT` table:

```
SELECT * FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

We first insert the information for the first department into the `DEPT` and `EMP` tables, and then create a savepoint:

```
INSERT INTO dept VALUES (50,'PAYROLL','LOS ANGELES');

1 row created.

INSERT INTO emp VALUES (7997,'EVANS','ACCTNT',7566,'2011-08-15',900,0,50);

1 row created.

savepoint A;

Savepoint created.
```

We then start processing information for the second department. Let's say in the middle of the transaction, an unknown error occurs between the insert into the `DEPT` table and the insert into the `EMP` table. In this case, we know this transaction of inserting the information into the recruiting department must be rolled back. At the same time, we wish to commit the transaction to the payroll department. Using the savepoint we created, we can commit a portion of the transaction, while rolling back the portion of the transaction we do not want to keep:

```
INSERT INTO dept VALUES (60,'RECRUITING','DENVER');

1 row created.

ROLLBACK to savepoint A;
```

```
Rollback complete.

COMMIT;

Commit complete.
```

Because of the savepoint, our rollback rolled back the incomplete transaction only for department 60, and the subsequent commit wrote the complete transaction for department 50 to the database in both the DEPT and EMP tables:

```
SELECT * FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 PAYROLL        LOS ANGELES

SELECT * FROM emp
WHERE empno = 7997;

     EMPNO ENAME      JOB              MGR HIREDATE    SAL       COMM  DEPTNO
---------- ---------- --------- ---------- ---------- ---- ---------- -------
      7997 EVANS      ACCTNT          7566 2011-08-15  900          0      50
```

There are many similar mechanisms or coding techniques you can use in programming languages such as PL/SQL. The SAVEPOINT command in the SQL language is a simple way to manage transactions without having to code more complex programming structures.

CHAPTER 9

Manually Tuning SQL

It has been said many times in books, articles, and other publications that over 90% of all performance problems on a database are due to poorly written SQL. Often, database administrators are given the task of "fixing the database" when queries are not performing adequately. The database administrator is often guilty before proven innocent—and often has the task of proving that a performance problem is not the database itself, but rather, simply, SQL statements that are not written efficiently. The goal, of course, is to have SQL statements written efficiently the first time. This chapter's focus is to help monitor and analyze existing queries to help show why they may be underperforming, as well as show some steps to improve queries.

If you have SQL code that you are maintaining or that needs help to improve performance, some of the questions that need to be asked first include the following:

- Has the query run before successfully?
- Was the query performance acceptable in the past?
- Are there any metrics on how long the query has run when successful?
- How much data is typically returned from the query?
- When was the last time statistics were gathered on the objects referenced in the query?

Once these questions are answered, it helps to direct the focus to where the problem may lie. You then may want to run an explain plan for the query to see if the execution plan is reasonable at first glance. The skill of reading an explain plan takes time and improves with experience. Sometimes, especially if there are views on top of the objects being queried, an explain plan can be lengthy and intimidating. Therefore, it's important to simply know what to look for first, and then dig as you go.

At times, poorly running SQL can expose database configuration issues, but normally, poorly performing SQL queries occur due to poorly written SQL statements. Again, as a database administrator or database developer, the best approach is to take time up front whenever possible to tune the SQL statements prior to ever running in a production environment. Often, a query's elapsed time is a benchmark for efficiency, which is an easy trap in which to fall. Over time, database characteristics change, more historical data may be stored for an application, and a query that performed well on initial install simply doesn't scale as an application matures. Therefore, it's important to take the time to do it right the first time, which is easy to say, but tough to accomplish when balancing client requirements, budgets, and project timelines.

9-1. Displaying an Execution Plan for a Query

Problem

You want to quickly retrieve an execution plan from within SQL Plus for a query.

Solution

From within SQL Plus, you can use the AUTOTRACE feature to quickly retrieve the execution plan for a query. This SQL Plus utility is very handy at getting the execution plan, along with getting statistics for the query's execution plan. In the most basic form, to enable AUTOTRACE within your session, execute the following command within SQL Plus:

```
SQL> set autotrace on
```

Then, you can run a query using AUTOTRACE, which will show the execution plan and query execution statistics for your query:

```
SELECT last_name, first_name
FROM employees NATURAL JOIN departments
WHERE employee_id = 101;

LAST_NAME                 FIRST_NAME
------------------------- --------------------
Kochhar                   Neena

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     1 |    33 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |               |     1 |    33 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    26 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |    11 |    77 |     1   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN          | DEPT_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        490  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

How It Works

There are several options to choose from when using AUTOTRACE, and the basic factors are as follows:

1. Do you want to execute the query?
2. Do you want to see the execution plan for the query?
3. Do you want to see the execution statistics for the query?

As you can see from Table 9-1, you can abbreviate each command, if so desired. The portions of the words in brackets are optional.

Table 9-1. *Options of AUTOTRACE Within SQL Plus*

AUTOTRACE Option	Execution Plan Shown	Statistics Shown	Query Executed
AUTOT[RACE] OFF	No	No	Yes
AUTOT[RACE] ON	Yes	Yes	Yes
AUTOT[RACE] ON EXP[LAIN]	Yes	No	Yes
AUTOT[RACE] ON STAT[ISTICS]	No	Yes	Yes
AUTOT[RACE] TRACE[ONLY]	Yes	Yes	Yes, but query output is suppressed.
AUTOT[RACE] TRACE[ONLY] EXP[LAIN]	Yes	No	No

The most common use for AUTOTRACE is to get the execution plan for the query, without running the query. By doing this, you can quickly see whether you have a reasonable execution plan, and can do this without having to execute the query:

```
SQL> set autot trace exp

SELECT l.location_id, city, department_id, department_name
  FROM locations l, departments d
  WHERE l.location_id = d.location_id(+)
  ORDER BY 1;
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |    27 |   837 |     8  (25)| 00:00:01 |
|   1 |  SORT ORDER BY      |             |    27 |   837 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN OUTER   |             |    27 |   837 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| LOCATIONS   |    23 |   276 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| DEPARTMENTS |    27 |   513 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

For the foregoing query, if you wanted to see only the execution statistics for the query, and did not want to see all the query output, you would do the following:

```
SQL> set autot trace stat
SQL> /

43 rows selected.


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         14  consistent gets
          0  physical reads
          0  redo size
       1862  bytes sent via SQL*Net to client
        438  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
         43  rows processed
```

Once you are done using `AUTOTRACE` for a given session and want to turn it off and run other queries without using `AUTOTRACE`, run the following command from within your SQL Plus session:

```
SQL> set autot off
```

The default for each SQL Plus session is `AUTOTRACE OFF`, but if you want to check to see what your current `AUTOTRACE` setting is for a given session, you can do that by executing the following command:

```
SQL> show autot
autotrace OFF
```

9-2. Customizing Execution Plan Output

Problem

You want to configure the explain plan output for your query based on your specific needs.

Solution

The Oracle-provided PL/SQL package DBMS_XPLAN has extensive functionality to get explain plan information for a given query. There are many functions within the DBMS_XPLAN package. The DISPLAY function can be used to quickly get the execution plan for a query, and also to customize the information that is presented to meet your specific needs. The following is an example that invokes the basic display functionality:

```
explain plan for
SELECT last_name, first_name
FROM employees JOIN departments USING(department_id)
WHERE employee_id = 101;

Explained.

SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
Plan hash value: 1833546154

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    22 |     1   (0)| 00:00:01|
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    22 |     1   (0)| 00:00:01|
|*  2 |   INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01|
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("EMPLOYEES"."DEPARTMENT_ID" IS NOT NULL)
   2 - access("EMPLOYEES"."EMPLOYEE_ID"=101)
```

The DBMS_XPLAN.DISPLAY procedure is very flexible in configuring how you would like to see output. If you wanted to see only the most basic execution plan output, using the foregoing query, you could configure the DBMS_XPLAN.DISPLAY function to get that output:

```
SELECT * FROM table(dbms_xplan.display(null,null,'BASIC'));

-----------------------------------------------------
| Id  | Operation                    | Name          |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |               |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |
|   2 |   INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |
-----------------------------------------------------
```

How It Works

The DBMS_XPLAN.DISPLAY function has a lot of built-in functionality to provide customized output based on your needs. The function provides four basic levels of output detail:

- BASIC
- TYPICAL (default)
- SERIAL
- ALL

Table 9-2 shows the format options that are included within each level of detail option.

Table 9-2. *DBMS_XPLAN.DISPLAY Options*

Format Option	BASIC	TYPICAL	SERIAL	ALL	Description
Basic (ID, Operation, Object Name	X	X	X	X	
ALIAS (Section)				X	Information on object aliases and query block information
BYTES (Column)		X	X	X	Estimated bytes
COST (Column)		X	X	X	Displays optimizer cost
NOTE (Section)		X	X	X	Shows NOTE section of the explain plan
PARALLEL (Detail within plan)		X		X	Show parallelism information related to the explain plan
PARTITION (Columns)		X	X	X	Displays partition pruning information
PREDICATE (Section)		X	X	X	Shows PREDICATE section of the explain plan
PROJECTION (Section)				X	Shows PROJECTION section of the explain plan
REMOTE (Detail within plan)				X	Shows information for distributed queries
ROWS (Column)		X	X	X	Shows estimated number of rows

If you simply want the default output format, there is no need to pass in any special format options:

```
SELECT * FROM table(dbms_xplan.display);
```

If you want to get all available output for a query, use the ALL level of detail format output option:

```
SELECT * FROM table(dbms_xplan.display(null,null,'ALL'));


-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    22 |     1   (0)| 00:00:01|
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    22 |     1   (0)| 00:00:01|
|*  2 |   INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01|
-----------------------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$38D4D5F3 / EMPLOYEES@SEL$1
   2 - SEL$38D4D5F3 / EMPLOYEES@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("EMPLOYEES"."DEPARTMENT_ID" IS NOT NULL)
   2 - access("EMPLOYEES"."EMPLOYEE_ID"=101)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "EMPLOYEES"."FIRST_NAME"[VARCHAR2,20], "EMPLOYEES"."LAST_NAME"[VARCHAR2,25]
   2 - "EMPLOYEES".ROWID[ROWID,10]

Note
-----
   - rule based optimizer used (consider using cbo)
```

One of the very nice features of the DBMS_XPLAN.DISPLAY function is after deciding the base level of detail you need, you can add individual options to be displayed in addition to the base output for that level of detail. For instance, if you want just the most basic output information, but also want to know cost information, you can format the DBMS_XPLAN.DISPLAY as follows:

```
SELECT * FROM table(dbms_xplan.display(null,null,'BASIC +COST'));
```

```
-------------------------------------------------------------------------------
| Id  | Operation                   | Name                       | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                            |     1   (0)|
|   1 |  RESULT CACHE               | 0fnzzb94z0dj2b5vzkmq4f4xcu |            |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES                  |     1   (0)|
|   3 |    INDEX UNIQUE SCAN        | EMP_EMP_ID_PK              |     0   (0)|
-------------------------------------------------------------------------------
```

You can also do the reverse, that is, subtract information you do not want to see. If you wanted to see the output using the `TYPICAL` level of output, but did not want to see the `ROWS` or `BYTES` information, you could issue the following query to display that level of output:

```
SELECT * FROM table(dbms_xplan.display(null,null,'TYPICAL -BYTES -ROWS'));

-------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

9-3. Graphically Displaying an Execution Plan

Problem

You want to quickly view an execution plan without having to run SQL statements to retrieve the execution plan. You would like to use a GUI to view the plan, so that you can just click your way to it.

Solution

From within Enterprise Manager, you can quickly find the execution plan for a query. In order to use this functionality, you will have to have Enterprise Manager configured within your environment. This can be either Database Control, which manages a single database, or Grid Control, which manages an enterprise of databases. In order to see the execution plan for a given query, you will need to navigate to the Top Sessions screen of Enterprise Manager. (Refer to the Oracle Enterprise Manager documentation for your specific release.) Once on the Top Sessions screen, you can drill down into session specific information. First, find your session. Then, click the SQL ID shown under Current SQL. From there, you can click Plan, and the execution plan will appear, such as the one shown in Figure 9-1.

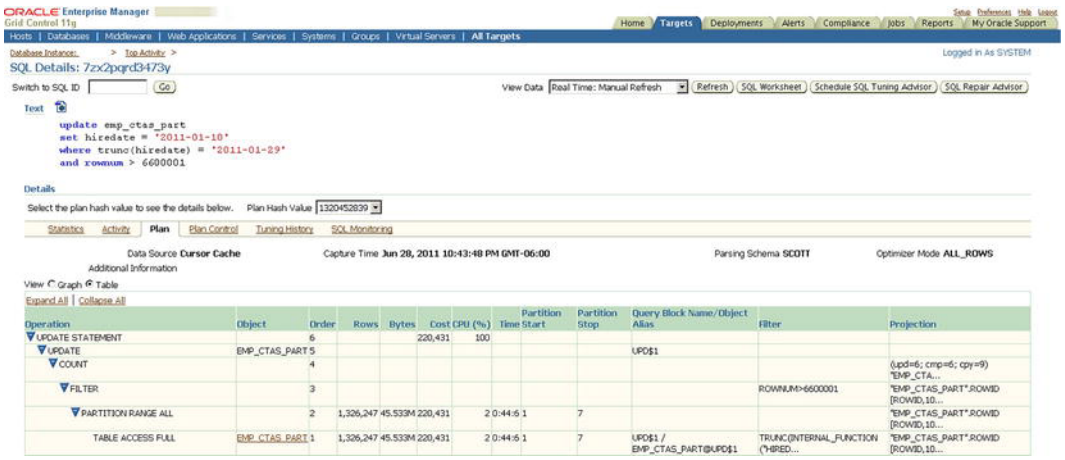

Figure 9-1. *Sample execution plan output from within Enterprise Manager*

How It Works

Using Enterprise Manager makes it very easy to find the execution plan for currently running SQL operations within your database. If a particular SQL statement isn't performing as expected, this method is one of the fastest ways to determine the execution plan for a running query or other SQL operation. In order to use this feature, you must be licensed for the Tuning Pack of Enterprise Manager.

9-4. Reading an Execution Plan

Problem

You have run an explain plan for a given SQL statement, and want to understand how to read the plan.

Solution

The execution plan for a SQL operation tells you step-by-step exactly how the Oracle optimizer will execute your SQL operation. Using `AUTOTRACE`, let's get an explain plan for the following query:

```
set autotrace trace explain

SELECT ename, dname
FROM emp JOIN dept USING (deptno);

-----------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |    14 |   308 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |         |    14 |   308 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     4 |    52 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | PK_DEPT |     4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |         |    14 |   126 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | EMP     |    14 |   126 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
       filter("EMP"."DEPTNO"="DEPT"."DEPTNO")

Note
-----
   - automatic DOP: Computed Degree of Parallelism is 1 because of parallel threshold
```

Once you have an explain plan to interpret, you can tell which steps are executed first because the innermost or most indented steps are executed first, and are executed from the inside out, in top-down order. In the foregoing query, we are joining the `EMP` and `DEPT` tables. Here are the steps of how the query is processed based on the execution plan:

1. The `PK_DEPT` index is scanned (ID 3).
2. All `EMP` table rows are scanned (ID 5).
3. Rows are retrieved from the `DEPT` table based on the matching entries in the `PK_DEPT` index (ID 2).
4. Resulting data from the `EMP` table is sorted (ID 4).
5. Data from the `EMP` and `DEPT` tables are then joined via a `MERGE JOIN` (ID 1).
6. The resulting data from the query is returned to the user (ID 0).

How It Works

When first looking at an explain plan and wanting to quickly get an idea of the steps in which the query will be executed, do the following:

1. Look for the most indented rows in the plan (the right-most rows). These will be executed first.
2. If multiple rows are at the same level of indentation, they will be executed in top-down fashion in the plan, with the highest rows in the plan first moving downward in the plan.
3. Look at the next most indented row or rows and continue working your way outward.
4. The top of the explain plan corresponds with the least indented or left-most part of the plan, and usually is where the results are returned to the user.

Once you have an explain plan for a query, and can understand the sequence of how the query will be processed, you then can move on and perform some analysis to determine if the explain plan you are looking at is efficient. When looking at your explain plan, answer these questions and consider these factors when determining if you have an efficient plan:

- What is the access path for the query (is the query performing a full table scan or is the query using an index)?
- What is the join method for the query (if a join condition is present)?
- Look at the columns within the filtering criteria found within the `WHERE` clause of the query, and determine if they are indexed.
- Get the volume or number of rows for each table in the query. Are the tables small, medium-sized, or large? This may help you determine the most appropriate join method. See Table 9-3 for a synopsis of the types of join methods.
- When were statistics last gathered for the objects involved in the query?
- Look at the `COST` column of the explain plan to get a starting cost.

By looking at our original explain plan, we determined that the `EMP` table is larger in size, and also that there is no index present on the `DEPTNO` column, which is used within a join condition between the `DEPT` and `EMP` tables. By placing an index on the `DEPTNO` column on the `EMP` table and gathering statistics on the `EMP` table, the plan now uses an index:

```
-------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |    14 |   280 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |         |    14 |   280 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP     |    14 |    98 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | EMP_I2  |    14 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |         |     4 |    52 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | DEPT    |     4 |    52 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Table 9-3. *Join Methods*

Method	Description
Hash	Most appropriate if at least one table involved in the query is large
Nested loop	Appropriate for smaller tables
Sort merge	Appropriate for pre-sorted data
Cartesian	Signifies either no join condition or a missing join condition; usually signifies an unwanted condition and query needs to be scrutinized to ensure there is a join condition for each and every table in the query

For information on parallel execution plans, see Chapter 15.

■ **Tip** One of the most common reasons for a sub-optimal explain plan is the lack of current statistics on one or more objects involved in a query.

9-5. Monitoring Long-Running SQL Statements

Problem

You have a SQL statement that runs a long time, and you want to be able to monitor the progress of the statement and find out when it will finish.

Solution

By viewing information for a long-running query in the `V$SESSION_LONGOPS` data dictionary view, you can gauge about when a query will finish. Let's say you are running the following query, with join conditions, against a large table:

```
SELECT last_name, first_name FROM employees_big
WHERE last_name = 'EVANS';
```

With a simple query against the `V$SESSION_LONGOPS` view, you can quickly get an idea of how long the query will execute, and when it will finish:

```
SELECT username, target, sofar blocks_read, totalwork total_blocks,
round(time_remaining/60) minutes
FROM v$session_longops
WHERE sofar <> totalwork
and username = 'HR';

USERNAME     TARGET               BLOCKS_READ TOTAL_BLOCKS    MINUTES
------------ -------------------- ----------- ------------ ----------
HR           HR.EMPLOYEES_BIG           81101      2353488         10
```

As the query progresses, you can see the BLOCKS_READ column increase, as well as the MINUTES column decrease. It is usually necessary to place the WHERE clause to eliminate rows that have been completed, which is why in the foregoing query it asked for rows where the SOFAR column did not equal TOTALWORK.

How It Works

In order to be able to monitor a query within the V$SESSION_LONGOPS view, the following requirements apply:

- The query must run for six seconds or greater.
- The table being accessed must be greater than 10,000 database blocks.
- TIMED_STATISTICS must be set or SQL_TRACE must be turned on.
- The objects within the query must have been analyzed via DBMS_STATS or ANALYZE.

This view can contain information on SELECT statements, DML statements such as UPDATE, as well as DDL statements such as CREATE INDEX. Some common operations that find themselves in the V$SESSION_LONGOPS view include table scans, index scans, join operations, parallel operations, RMAN backup operations, sort operations, and Data Pump operations.

9-6. Identifying Resource-Consuming SQL Statements That Are Currently Executing

Problem

You have contention issues within your database, and want to identify the SQL statement consuming the most system resources.

■ **Note** Recipe 9-9 shows how to examine the historical record to find resource-consuming SQL statements that have executed in the past, but that are not currently executing.

Solution

Look at the V$SQLSTATS view, which gives information about currently or recently run SQL statements. If you wanted to get the top five recent SQL statements that performed the highest disk I/O, you could issue the following query:

```
SELECT sql_text, disk_reads FROM
  (SELECT sql_text, buffer_gets, disk_reads, sorts,
   cpu_time/1000000 cpu, rows_processed, elapsed_time
   FROM v$sqlstats
   ORDER BY disk_reads DESC)
WHERE rownum <= 5;
```

If you wanted to see the top five SQL statements by CPU time, sorts, loads, invalidations, or any other column, simply replace the disk_reads column in the foregoing query with your desired column. The SQL_TEXT column can make the results look messy, so another alternative is to substitute the SQL_TEXT column with SQL_ID, and then, based on the statistics shown, you can run a query to simply get the SQL_TEXT based on a given SQL_ID.

How It Works

The V$SQLSTATS view is meant to help more quickly find information on resource-consuming SQL statements. V$SQLSTATS has the same information as the V$SQL and V$SQLAREA views, but V$SQLSTATS has only a subset of columns of the other views. However, data is held within the V$SQLSTATS longer than either V$SQL or V$SQLAREA.

Sometimes, there are SQL statements that are related to the database background processing of keeping the database running, and you may not want to see those statements, but only the ones related to your application. If you join V$SQLSTATS to V$SQL, you can see information for particular users. See the following example:

```
SELECT schema, sql_text, disk_reads, round(cpu,2) FROM
  (SELECT s.parsing_schema_name schema, t.sql_id, t.sql_text, t.disk_reads,
  t.sorts, t.cpu_time/1000000 cpu, t.rows_processed, t.elapsed_time
   FROM v$sqlstats t join v$sql s on(t.sql_id = s.sql_id)
  WHERE parsing_schema_name = 'SCOTT'
  ORDER BY disk_reads DESC)
WHERE rownum <= 5;
```

Keep in mind that V$SQL represents SQL held in the shared pool, and is aged out faster than the data in V$SQLSTATS, so this query will not return data for SQL that has been already aged out of the shared pool.

9-7. Seeing Execution Statistics for Currently Running SQL

Problem

You want to view execution statistics for SQL statements that are currently running.

Solution

You can use the V$SQL_MONITOR view to see real-time statistics of currently running SQL, and see the resource consumption used for a given query based on such statistics as CPU usage, buffer gets, disk reads, and elapsed time of the query. Let's first find a current executing query within our database:

```
SELECT sid, sql_text FROM v$sql_monitor
WHERE status = 'EXECUTING';

       SID SQL_TEXT
---------- --------------------------------------------------------------------
       100 select department_name, city, avg(salary)
           from employees_big join departments using(department_id)
           join locations using (location_id)
           group by department_name, city
           having avg(salary) > 2000
           order by 2,1
```

For the foregoing executing query found in V$SQL_MONITOR, we can see the resource utilization for that statement as it executes:

```
SELECT sid, buffer_gets, disk_reads, round(cpu_time/1000000,1) cpu_seconds
FROM v$sql_monitor
WHERE SID=100
AND status = 'EXECUTING';

       SID BUFFER_GETS DISK_READS CPU_SECONDS
---------- ----------- ---------- -----------
       100      149372       4732        39.1
```

The V$SQL_MONITOR view contains currently running SQL statements, as well as recently run SQL statements. If you wanted to see the top five most CPU-consuming queries in your database, you could issue the following query:

```
SELECT * FROM (
  SELECT sid, buffer_gets, disk_reads, round(cpu_time/1000000,1) cpu_seconds
  FROM v$sql_monitor
  ORDER BY cpu_time desc)
WHERE rownum <= 5;

       SID BUFFER_GETS DISK_READS CPU_SECONDS
---------- ----------- ---------- -----------
        20     1332665      30580       350.5
       105      795330      13651       269.7
        20      259324       5449        71.6
        20      259330       5485        71.3
       100      259236       8188        67.9
```

How It Works

SQL statements are monitored in V$SQL_MONITOR under the following conditions:

- Automatically for any parallelized statements
- Automatically for any DML or DDL statements
- Automatically if a particular SQL statement has consumed at least five seconds of CPU or I/O time
- Monitored for any SQL statement that has monitoring set at the statement level

To turn monitoring on at the statement level, a hint can be used. See the following example:

```
SELECT /*+ monitor */ ename, dname
FROM emppart JOIN dept USING (deptno);
```

If, for some reason, you do not want certain statements monitored, you can use the NOMONITOR hint in the statement to prevent monitoring from occurring for a given statement.

Statistics in V$SQL_MONITOR are updated near real-time, that is, every second. Any currently executing SQL statement that is being monitored can be found in V$SQL_MONITOR. Completed queries can be found there for at least one minute after execution ends, and can exist there longer, depending on the space requirements needed for newly executed queries. One key advantage of the V$SQL_MONITOR view is it has detailed statistics for each and every execution of a given query, unlike V$SQL, where results are cumulative for several executions of a SQL statement. In order to drill down, then, to a given execution of a SQL statement, you need three columns from V$SQL_MONITOR:

1. SQL_ID
2. SQL_EXEC_START
3. SQL_EXEC_ID

If we wanted to see all executions for a given query (based on the SQL_ID column), we can get that information by querying on the three necessary columns to drill to a given execution of a SQL query:

```
SELECT * FROM (
  SELECT sql_id, to_char(sql_exec_start,'yyyy-mm-dd:hh24:mi:ss') sql_exec_start,
         sql_exec_id, sum(buffer_gets) buffer_gets,
         sum(disk_reads) disk_reads, round(sum(cpu_time/1000000),1) cpu_secs
  FROM v$sql_monitor
  WHERE sql_id = 'fcg00hyh7qbpz'
  GROUP BY sql_id, sql_exec_start, sql_exec_id
  ORDER BY 6 desc)
WHERE rownum <= 5;

SQL_ID        SQL_EXEC_START      SQL_EXEC_ID BUFFER_GETS DISK_READS CPU_SECS
------------- ------------------- ----------- ----------- ---------- --------
fcg00hyh7qbpz 2011-05-21:12:28:10    16777222      259324       5449     71.6
fcg00hyh7qbpz 2011-05-21:12:29:24    16777223      259330       5485     71.3
fcg00hyh7qbpz 2011-05-21:12:26:08    16777220      213823       4502     58.4
fcg00hyh7qbpz 2011-05-21:12:27:09    16777221      211752       4579     58.1
fcg00hyh7qbpz 2011-05-21:12:25:37    16777219      107973       2414     29.4
```

Keep in mind that if a statement is running in parallel, one row will appear for each parallel thread for the query, including one for the query coordinator. However, they will share the same SQL_ID, SQL_EXEC_START, and SQL_EXEC_ID values. In this case, you could perform an aggregation on a particular statistic, if desired. See the following example for a parallelized query, along with parallel slave information denoted by the PX_SERVER# column:

```
SELECT sql_id, sql_exec_start, sql_exec_id, px_server# px#, disk_reads,
       cpu_time/1000000 cpu_secs, buffer_gets
FROM v$sql_monitor
WHERE status = 'EXECUTING'
ORDER BY px_server#;

SQL_ID        SQL_EXEC_S SQL_EXEC_ID PX# DISK_READS CPU_SECS BUFFER_GETS
------------- ---------- ----------- --- ---------- -------- -----------
0gzf8010xdasr 2011-05-21    16777216   1       4306     38.0      136303
0gzf8010xdasr 2011-05-21    16777216   2       4625     40.6      146497
0gzf8010xdasr 2011-05-21    16777216   3       4774     41.6      149717
0gzf8010xdasr 2011-05-21    16777216   4       4200     37.6      132167
0gzf8010xdasr 2011-05-21    16777216              6     92.2          53
```

Then, to perform a simple aggregation for a given query, in this case, our parallelized query, the aggregation is done on the three key columns that make up a single execution of a given SQL statement:

```
SELECT sql_id,sql_exec_start, sql_exec_id, sum(buffer_gets) buffer_gets,
       sum(disk_reads) disk_reads, round(sum(cpu_time/1000000),1) cpu_seconds
FROM v$sql_monitor
WHERE sql_id = '0gzf8010xdasr'
GROUP BY sql_id, sql_exec_start, sql_exec_id;

SQL_ID        SQL_EXEC_S SQL_EXEC_ID BUFFER_GETS DISK_READS CPU_SECONDS
------------- ---------- ----------- ----------- ---------- -----------
0gzf8010xdasr 2011-05-21    16777216      642403      20351       283.7
```

If you wanted to perform an aggregation for one SQL statement, regardless of the number of times is has been executed, simply run the aggregate query only on the SQL_ID column, as shown here:

```
SELECT sql_id, sum(buffer_gets) buffer_gets,
       sum(disk_reads) disk_reads, round(sum(cpu_time/1000000),1) cpu_seconds
FROM v$sql_monitor
WHERE sql_id = '0gzf8010xdasr'
GROUP BY sql_id;
```

■ **Note** Initialization parameter STATISTICS_LEVEL must be set to TYPICAL or ALL, and CONTROL_MANAGEMENT_PACK_ACCESS must be set to DIAGNOSTIC+TUNING for SQL monitoring to occur.

9-8. Monitoring Progress of a SQL Execution Plan

Problem

You want to see the progress a query is making from within the execution plan used.

Solution

There are a couple of ways to get information to see where a query is executing in terms of the execution plan. First, by querying the V$SQL_PLAN_MONITOR view, you can get information for all queries that are in progress, as well as recent queries that are complete. If we are joining two tables to get employee and department information, our query would look like this:

```
SELECT ename, dname
FROM emppart JOIN dept USING (deptno);

-------------------------------------------
| Id  | Operation               | Name     |
-------------------------------------------
|   0 | SELECT STATEMENT        |          |
|   1 |  PX COORDINATOR         |          |
|   2 |   PX SEND QC (RANDOM)   | :TQ10001 |
|   3 |    HASH JOIN            |          |
|   4 |     BUFFER SORT         |          |
|   5 |      PX RECEIVE         |          |
|   6 |       PX SEND BROADCAST | :TQ10000 |
|   7 |        TABLE ACCESS FULL| DEPT     |
|   8 |     PX BLOCK ITERATOR   |          |
|   9 |      TABLE ACCESS FULL  | EMPPART  |
-------------------------------------------
```

To see information for the foregoing query while it is currently running, you can issue a query like the one shown here (some rows have been removed for conciseness):

```
column operation format a25
column plan_line_id format 9999 heading 'LINE'
column plan_options format a10 heading 'OPTIONS'
column status format a10
column output_rows heading 'ROWS'
break on sid on sql_id on status

SELECT sid, sql_id, status, plan_line_id,
plan_operation || ' ' || plan_options operation, output_rows
FROM v$sql_plan_monitor
WHERE status not like '%DONE%'
ORDER BY 1,4;
```

```
       SID SQL_ID        STATUS      LINE OPERATION                      ROWS
---------- ------------- ---------- ----- ------------------------ ----------
        18 36bdwxutr5n75 EXECUTING      0 SELECT STATEMENT            3929326
                                        1 PX COORDINATOR              3929326
        27 36bdwxutr5n75 EXECUTING      0 SELECT STATEMENT                  0
                                        2 PX SEND QC (RANDOM)         1752552
                                        3 HASH JOIN                   1752552
                                        8 PX BLOCK ITERATOR           1752552
                                        9 TABLE ACCESS FULL           1752552
       101 36bdwxutr5n75 EXECUTING      0 SELECT STATEMENT                  0
                                        2 PX SEND QC (RANDOM)         2148232
                                        3 HASH JOIN                   2148232
                                        8 PX BLOCK ITERATOR           2148232
                                        9 TABLE ACCESS FULL           2148232
```

In this particular example, the EMPPART table has a parallel degree of 2, and we can see that for SIDs 27 and 101, these are the parallel slaves that are getting the data. As these processes pass data back to the query coordinator and then back to the user, we can see that when we look at SID 18. If we simply run subsequent queries against the V$SQL_PLAN_MONITOR view, we can see the progress of the query as it is executing. In the foregoing example, we simply see the output row values increasing as the query progresses.

Another method of seeing the progress of a query via the execution plan is by using the DBMS_SQLTUNE.REPORT_SQL_MONITOR function. If we use the same query against the EMPPART and DEPT tables used in the previous example, we can run the REPORT_SQL_MONITOR function to get a graphical look at the progress. See the following example of how to generate the file that would produce the HTML file that could be, in turn, used to view our progress. Figure 9-2 shows portions of the resulting report.

```
set pages 9999
set long 1000000
SELECT DBMS_SQLTUNE.REPORT_SQL_MONITOR(sql_id=> '36bdwxutr5n75',type=>'HTML') FROM dual;
```

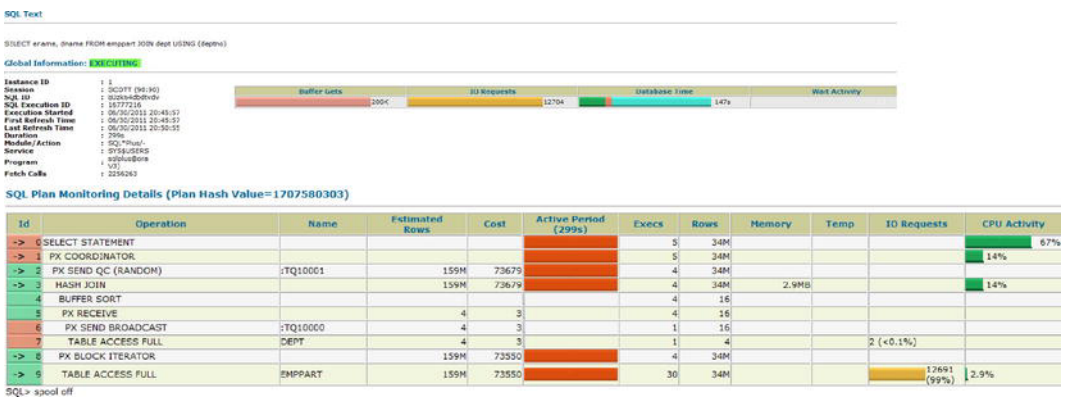

Figure 9-2. *Sample HTML report from* DBMS_SQLTUNE.REPORT_SQL_MONITOR

How It Works

The V$SQL_PLAN_MONITOR is populated from the V$SQL_MONITOR view (see Recipe 9-7). Both of these views are new as of Oracle 11g, and are updated every second that a statement executes. The V$SQL_MONITOR view is populated each time a SQL statement is monitored.

The DBMS_SQLTUNE.REPORT_SQL_MONITOR function can be invoked in several ways. The level of detail, as well as the type of detail you wish to see in the report, can be changed based on the parameters passed into the function. The output can be viewed in several formats, including plain text, HTML, and XML. The default output format is plain text. As an example, let's say we wanted to see the output for our join against the EMPPART and DEPT tables. In this instance, we want the output in text format. We want the detail aggregated, and we want to see just the most basic level of detail. Our query would then be run as follows:

```
SELECT DBMS_SQLTUNE.REPORT_SQL_MONITOR
(sql_id=>'36bdwxutr5n75',event_detail=>'NO',report_level=>'BASIC') FROM dual;

SQL Monitoring Report

SQL Text
------------------------------
select ename, dname from emppart join dept using (deptno)

Global Information
------------------------------
 Status              :  EXECUTING
 Instance ID         :  1
 Session             :  SCOTT (27:229)
 SQL ID              :  36bdwxutr5n75
 SQL Execution ID    :  16777225
 Execution Started   :  05/15/2011 14:56:16
 First Refresh Time  :  05/15/2011 14:56:16
 Last Refresh Time   :  05/15/2011 15:09:47
 Duration            :  812s
 Module/Action       :  SQL*Plus/-
 Service             :  SYS$USERS
 Program             :  sqlplus@ora
 Fetch Calls         :  6131367

Global Stats
=========================================================================================
| Elapsed |   Cpu   |    IO    | Concurrency |  Other   | Fetch | Buffer | Read | Read  |
| Time(s) | Time(s) | Waits(s) |  Waits(s)   | Waits(s) | Calls |  Gets  | Reqs | Bytes |
=========================================================================================
|     398 |     235 |     6.45 |        0.04 |      156 |    6M |   556K | 17629|   4GB |
=========================================================================================
```

Refer to the Oracle PL/SQL Packages and Types Reference for a complete list of all the parameters that can be used to execute the REPORT_SQL_MONITOR function. It is a very robust function, and there are a myriad of permutations to report on, based on your specific need.

9-9. Identifying Resource-Consuming SQL Statements That Have Executed in the Past

Problem

You want to view information on previously run SQL statements to aid in identifying resource-intensive operations.

■ **Note** Recipe 9-6 shows how to identify *currently executing* statements that are resource-intensive.

Solution

The DBA_HIST_SQLSTAT and DBA_HIST_SQLTEXT views are two of the views that can be used to get historical information on SQL statements and their resource consumption statistics. For example, to get historical information on what SQL statements are incurring the most disk reads, you can issue the following query against DBA_HIST_SQLSTAT:

```
SELECT * FROM (
  SELECT sql_id, sum(disk_reads_delta) disk_reads_delta,
                 sum(disk_reads_total) disk_reads_total,
                 sum(executions_delta) execs_delta,
                 sum(executions_total) execs_total
  FROM dba_hist_sqlstat
  GROUP BY sql_id
  ORDER BY 2 desc)
WHERE rownum <= 5;

SQL_ID        DISK_READS_DELTA DISK_READS_TOTAL EXECS_DELTA EXECS_TOTAL
------------- ---------------- ---------------- ----------- -----------
36bdwxutr5n75          6306401         10933153          13          24
0bx1z9rbm10a1          1590538          1590538           2           2
0gzf8010xdasr           970292          1848743           1           3
1gtkxf53fk7bp           969785           969785           7           7
4h81qj5nspx6s           869588           869588           2           2
```

Since the actual text of the SQL isn't stored in DBA_HIST_SQLSTAT, you can then look at the associated DBA_HIST_SQLTEXT view to get the SQL text for the query with the highest number of disk reads:

```
SELECT sql_text FROM dba_hist_sqltext
WHERE sql_id = '36bdwxutr5n75';

SQL_TEXT
----------------------------------------
select ename, dname
from emppart join dept using (deptno)
```

How It Works

There are many useful statistics to get from the DBA_HIST_SQLSTAT view regarding historical SQL statements, including the following:

- CPU utilization
- Elapsed time of execution
- Number of executions
- Total disk reads and writes
- Buffer get information
- Parallel server information
- Rows processed
- Parse calls
- Invalidations

Furthermore, this information is separated by two views of the data. There is a set of "Total" information in one set of columns, and there is a "Delta" set of information in another set of columns. The "Total" set of columns is calculated based on instance startup. The "Delta" columns are based on the values seen in the BEGIN_INTERVAL_TIME and END_INTERVAL_TIME columns of the DBA_HIST_SNAPSHOT view.

If you want to see explain plan information for historical SQL statements, there is an associated view available to retrieve that information for a given query. You can access the DBA_HIST_SQL_PLAN view to get the explain plan information for historical SQL statements. See the following example:

```
SELECT id, operation || ' ' || options operation, object_name, cost, bytes
FROM dba_hist_sql_plan
WHERE sql_id = '0gzf8010xdasr'
ORDER BY 1;

        ID OPERATION                 OBJECT_NAME        COST      BYTES
---------- ------------------------- ------------ ---------- ----------
         0 SELECT STATEMENT                            73679
         1 PX COORDINATOR
         2 PX SEND QC (RANDOM)       :TQ10001          73679 3506438144
         3 HASH JOIN                                   73679 3506438144
         4 BUFFER SORT
         5 PX RECEIVE                                      3         52
         6 PX SEND BROADCAST         :TQ10000              3         52
         7 TABLE ACCESS FULL         DEPT                  3         52
         8 PX BLOCK ITERATOR                           73550 1434451968
         9 TABLE ACCESS FULL         EMPPART           73550 1434451968

10 rows selected.
```

9-10. Comparing SQL Performance After a System Change

Problem

You are making a system change, and want to see the impact that change will have on performance of a SQL statement.

Solution

By using the Oracle SQL Performance Analyzer, and specifically the DBMS_SQLPA package, you can quantify the performance impact a system change will have on one or more SQL statements. A system change can be an initialization parameter change, a database upgrade, or any other change to your environment that could affect SQL statement performance.

Let's say you are going to be performing a database upgrade, and want to see the impact the upgrade is going to have on a series of SQL statements run within your database. Using the DBMS_SQLPA package, the basic steps to get the information needed to perform the analysis generally are as follows:

1. Create an analysis task based on a single or series of SQL statements.
2. Run an analysis for those statements based on your current configuration.
3. Perform the given change to your environment (like a database upgrade).
4. Run an analysis for those statements based on the new configuration.
5. Run a "before and after" comparison to determine what impact the change has on the performance of your SQL statement(s).
6. Generate a report to view the output of the comparison results.

Using the foregoing steps, see the following example for a single query. First, we need to create an analysis task. For the database upgrade example, this would be done on an appropriate test database that is Oracle 11g. In this case, within SQL Plus, we will do the analysis for one specific SQL statement:

```
variable g_task varchar2(100);

EXEC :g_task := DBMS_SQLPA.CREATE_ANALYSIS_TASK(sql_text => 'select ename, dname from emppart
join dept using(deptno)');
```

In order to properly simulate this scenario, on our Oracle 11g database, we then set the optimizer_features_enable parameter back to Oracle 10g. We then run an analysis for our query using the "before" conditions—in this case, with a previous version of the optimizer:

```
alter session set optimizer_features_enable='10.2.0.4';

EXEC DBMS_SQLPA.EXECUTE_ANALYSIS_TASK(task_name=>:g_task,execution_type=>'test
execute',execution_name=>'before_change');
```

After completing the before analysis, we set the optimizer to the current version of our database, which, for this example, represents the version to which we are upgrading our database:

```
alter session set optimizer_features_enable='11.2.0.1';

EXEC DBMS_SQLPA.EXECUTE_ANALYSIS_TASK(task_name=>:g_task,execution_type=>'test
execute',execution_name=>'after_change');
```

Now that we have created our analysis task based on a given SQL statement, and have run "before" and "after" analysis tasks for that statement based on the changed conditions, we can now run an analysis task to compare the results of the two executions of our query. There are several metrics that can be compared. In this case, we are comparing "buffer gets":

```
EXEC DBMS_SQLPA.EXECUTE_ANALYSIS_TASK(task_name=>:g_task,execution_type=>'COMPARE
PERFORMANCE',execution_name=>'compare change',execution_params =>
dbms_advisor.arglist('comparison_metric','buffer_gets'));
```

Finally, we can now use the `REPORT_ANALYSIS_TASK` function of the `DBMS_SQLPA` package in order to view the results. In the following example, we want to see output only if the execution plan has changed. The output can be in several formats, the most popular being HTML and plain text. For our example, we produced text output:

```
set long 100000 longchunksize 100000 linesize 200 head off feedback off echo off
spool compare_report.txt

SELECT DBMS_SQLPA.REPORT_ANALYSIS_TASK(:g_task, 'TEXT', 'CHANGED_PLANS', 'ALL')
FROM DUAL;

General Information
-------------------------------------------------------------------------------------------
 Task Information:                             Workload Information:
-------------------------------------------  ------------------------------------------
  Task Name    : TASK_1383
  Task Owner   : SCOTT
  Description  :

Execution Information:
-------------------------------------------------------------------------------------------
  Execution Name  : compare change         Started             : 05/28/2011 17:28:07
  Execution Type  : COMPARE PERFORMANCE    Last Updated        : 05/28/2011 17:28:08
  Description     :                        Global Time Limit   : UNLIMITED
  Scope           : COMPREHENSIVE          Per-SQL Time Limit  : UNUSED
  Status          : COMPLETED              Number of Errors    : 0

Analysis Information:
-------------------------------------------------------------------------------------------
 Comparison Metric: BUFFER_GETS
 ------------------
 Workload Impact Threshold: 1%
 --------------------------
 SQL Impact Threshold: 1%
 ----------------------
```

```
Before Change Execution:                    After Change Execution:
 ----------------------------------------   ----------------------------------------
  Execution Name      : before_change         Execution Name      : after_change
  Execution Type      : TEST EXECUTE          Execution Type      : TEST EXECUTE
  Description         :                       Description         :
  Scope               : COMPREHENSIVE         Scope               : COMPREHENSIVE
  Status              : COMPLETED             Status              : COMPLETED
  Started             : 05/28/2011 17:19:47   Started             : 05/28/2011 17:23:43
  Last Updated        : 05/28/2011 17:23:37   Last Updated        : 05/28/2011 17:28:07
  Global Time Limit   : UNLIMITED             Global Time Limit   : UNLIMITED
  Per-SQL Time Limit  : UNUSED                Per-SQL Time Limit  : UNUSED
  Number of Errors    : 0                     Number of Errors    : 0
-------------------------------------------------------------------------------------

Execution Statistics:
-------------------------------------------------------------------------------------
|              | Impact on | Value     | Value     | Impact   | % Workload | % Workload |
| Stat Name    | Workload  | Before    | After     | on SQL   |   Before   |   After    |
-------------------------------------------------------------------------------
| elapsed_time |   -12.24% |  230.819  |  259.072  | -12.24%  |       100% |100%        |
| parse_time   |    -4100% |         0 |      .041 |   -4.1%  |         0% |100%        |
| cpu_time     |    -1.62% |  198.948  |  202.177  |  -1.62%  |       100% |100%        |
| buffer_gets  |        0% | 16882239  | 16882239  |      0%  |       100% |100%        |
| cost         |        0% | 16812553  | 16812553  |      0%  |       100% |100%        |
| reads        |   -34.9%  |    77791  |   104939  |  -34.9%  |       100% |100%        |
| writes       |        0% |         0 |         0 |      0%  |         0% |  0%        |
| rows         |         % | 16777222  | 16777222  |       %  |          % |   %        |
-------------------------------------------------------------------------------

Findings (1):
------------------------------
 1. The structure of the SQL execution plan has changed.

Execution Plan Before Change:
------------------------------
-----------------------------------------------------------------------------------------
| Id  | Operation                      | Name    | Rows     | Bytes     | Cost    | Time |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |         | 16777222 | 671088880 | 16793338 |      |
|   1 |   NESTED LOOPS                 |         | 16777222 | 671088880 | 16793338 |      |
|   2 |    PARTITION RANGE ALL         |         | 16777222 | 335544440 |    16116 |      |
|   3 |     TABLE ACCESS FULL          | EMPPART | 16777222 | 335544440 |    16116 |      |
|   4 |    TABLE ACCESS BY INDEX ROWID | DEPT    |        1 |        20 |        1 |      |
| * 5 |     INDEX UNIQUE SCAN          | PK_DEPT |        1 |           |          |      |
-----------------------------------------------------------------------------------------
```

```
Execution Plan After Change:
----------------------------
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows     | Bytes     | Cost   | Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         | 16777222 | 352321662 | 16812553| 6:02:31|
|   1 |   NESTED LOOPS                |         |          |           |        |        |
|   2 |    NESTED LOOPS               |         | 16777222 | 352321662 | 16812553|56:02:31|
|   3 |     PARTITION RANGE ALL       |         | 16777222 | 150994998 |    29013|00:05:49|
|   4 |      TABLE ACCESS FULL        | EMPPART | 16777222 | 150994998 |    29013|00:05:49|
| * 5 |     INDEX UNIQUE SCAN         | PK_DEPT |        1 |           |        0|00:00:01|
|   6 |    TABLE ACCESS BY INDEX ROWID| DEPT    |        1 |        12 |        1|00:00:01|
-------------------------------------------------------------------------------------------------
```

In the foregoing example output, we can see the before and after execution statistics, as well as the before and after execution plans. We can also see an estimated workload impact and SQL impact percentages, which are very useful in order to see, at a quick glance, if there is a large impact by making the system change being made. In this example, we can see there would be a 1% change, and by looking at the execution plans, we see a negligible difference. So, for the foregoing query, the database upgrade will essentially have minimal or no impact on the performance of our query. If you see an impact percentage of 10% or greater, it may mean more analysis and tuning need to occur to proactively tune the query or the system prior to making the change to your production environment. In order to get an accurate comparison, it is also recommended to export production statistics and import them into your test environment prior to performing the analysis using `DBMS_SQLPA`.

■ **Tip** In SQL Plus, remember to `SET LONG` and `SET LONG CHUNKSIZE` in order for output to be displayed properly.

How It Works

The SQL Performance Analyzer and the `DBMS_SQLPA` package can be used to analyze a SQL workload, which can be defined as any of the following:

- A SQL statement
- A SQL ID stored in cache
- A SQL tuning set (see Chapter 11 for information on SQL tuning sets)
- A SQL ID based on a snapshot from the Automatic Workload Repository (see Chapter 4 for more information)

In normal circumstances, it is easiest to gather information on a series of SQL statements, rather than one single statement. Getting information via Automatic Workload Repository (AWR) snapshots or via SQL tuning sets is the easiest way to get information for a series of statements. The AWR snapshots will contain information based on a specific time period, while SQL tuning sets will contain information on a specifically targeted set of SQL statements. Some of the possible key reasons to consider doing a "before and after" performance analysis include the following:

- Initialization parameter changes
- Database upgrades
- Hardware changes
- Operating system changes
- Application schema object additions or changes
- The implementation of SQL baselines or profiles

There is an abundance of information available for comparison. When reporting on the information gathered in your analysis, it may be beneficial to show only the output for SQL statements affected adversely by the system change. For instance, you may want to narrow down the information shown from the `REPORT_ANALYSIS_TASK` function to show information only on SQL statements such as the following:

- Those statements that show regressed performance
- Those statements with a changed execution plan
- Those statements that show errors in the SQL statements

It may be beneficial to flush the `shared_pool` and/or the `buffer_cache` prior to gathering information on each of your tasks, which will aid in getting the best possible information for comparison. Information on analysis tasks is stored in the data dictionary. You can reference any of the data dictionary views prefaced with "`DBA_ADVISOR`" to get information on performance analysis tasks you have created, executions performed, as well as execution statistics, execution plans, and report information. Refer to the Oracle PL/SQL Packages and Types Reference for your version of the database for a complete explanation of the `DBMS_SQLPA` package, which is new as of version Oracle 11g.

■ **Note** The `ADVISOR` system privilege is needed to perform the analysis tasks using `DBMS_SQLPA`.

CHAPTER 10

Tracing SQL Execution

Tracing session activity is at the heart of most SQL performance tuning exercises. Oracle provides a rich set of tools to trace SQL activity. This chapter introduces the Oracle SQL trace facility and shows you how to set up SQL tracing in your environment. Oracle provides numerous "events" that help you perform various types of traces.

Although there are several tracing methods available, Oracle now recommends that you use the `DBMS_MONITOR` package for most types of tracing. The chapter contains several recipes that explain how to use this package to generate traces. In addition, we show how to trace sessions by setting various Oracle events, the setting of which is often requested by Oracle Support. You'll learn how to trace a single SQL statement, a session as well as an entire instance, as well as how to trace parallel queries. There are recipes that show how to trace another user's session and how to use a trigger to start a session trace. You'll also learn how to trace the Oracle optimizer's execution path.

Oracle provides the `TKPROF` utility as well as the freely downloadable profiler named Oracle Trace Analyzer. This chapter shows how to use both of these profilers to analyze the raw trace files you generate.

10-1. Preparing Your Environment

Problem

You want to make sure your database is set up correctly for tracing SQL sessions.

Solution

You must do three things before you can start tracing SQL statements:

1. Enable timed statistics collection.
2. Specify a destination for the trace dump file.
3. Adjust the trace dump file size.

You can enable the collection of timed statistics by setting the `timed_statistics` parameter to `true`. Check the current value of this parameter first:

```
SQL> sho parameter statistics

NAME                                 TYPE        VALUE
------------------------------------ ----------- -----------
…
statistics_level                     string      TYPICAL
timed_statistics                     boolean     TRUE
SQL>
```

If the value of the `timed_statistics` parameter is `false`, you set it to `true` with the following statement.

```
SQL> alter system set timed_statistics=true scope=both;

System altered.

SQL>
```

You can also set this parameter at the session level with the following statement:

```
SQL> alter session set timed_statistics=true
```

You can find the location of the trace directory (which was referred to as the user dump directory in pre-Oracle Database 11g releases) with the following command:

```
SQL> select name,value from v$diag_info
  2* where name='Diag Trace'
SQL> /

NAME                                              VALUE
----------------------------    ----------------------------------------

Diag Trace                      c:\app\ora\diag\rdbms\orcl1\orcl1\trace


SQL>
```

In Oracle Database 11g, the default value of the `max_dump_file_size` parameter is `unlimited`, as you can verify by issuing the following command:

```
SQL> sho parameter dump

NAME                                 TYPE        VALUE
------------------------------------ ----------- ----------
…
max_dump_file_size                   string      unlimited
```

An unlimited dump file size means that the file can grow as large as the operating system permits

How It Works

Before you can trace any SQL sessions, ensure that you've set the `timed_statistics` initialization parameter to `true`. If the value for this parameter is `false`, SQL tracing is disabled. Setting the `timed_statistics` parameter to `true` enables the database to collect statistics such as the CPU and

elapsed times and store them in various dynamic performance tables. The default value of this parameter, starting with the Oracle 11.1.0.7.0 release, depends on the value of the initialization parameter `statistics_level`. If you set the `statistics_level` parameter to `basic`, the default value of the `timed_statistics` parameter is `false`. If you set `statistics_level` to the value `typical` or `all`, the default value of the `timed_statistics` parameter is `true`. The `timed_statistics` parameter is dynamic, meaning you don't have to restart the database to turn it on—you can turn this parameter on for the entire database without a significant overhead. You can also turn the parameter on only for an individual session.

When you trace a SQL session, Oracle generates a trace file that contains diagnostic data that's very useful in troubleshooting SQL performance issues. Starting with Oracle Database 11g, the database stores all diagnostic files under a dedicated diagnostic directory that you specify through the `diagnostic_dest` initialization parameter. The structure of the diagnostic directory is as follows:

```
<diagnostic_dest>/diag/rdbms/<dbname>/<instance>
```

The diagnostic directory is called the ADR Home. If your database name is `prod1` and the instance name is `prod1` as well, then the ADR home directory will be the following:

```
<diagnostic_dest>/diag/rdbms/prod1/prod1
```

The ADR home directory contains trace files in the `<ADR Home>/trace` subdirectory. Trace files usually have the extension `.trc`. You'll notice that several trace files have a corresponding trace map file with the `.trm` extension. The `.trm` files contain structural information about trace files, which the database uses for searching and navigation. You can view the diagnostic directory setting for a database with the following command:

```
SQL> sho parameter diagnostic_dest

NAME                                 TYPE        VALUE
------------------------------------ ----------- -----------
diagnostic_dest                      string      C:\APP\ORA
SQL>
```

The `V$DIAG_INFO` view shows the location of the various diagnostic directories, including the trace directory, which is listed in this view under the name Diag Trace. Although the new database diagnosability infrastructure in Oracle Database 11g ignores the `user_dump_dest` initialization parameter, the parameter still exists, and points to the same directory as the `$ADR_BASE\diag\rdbms\<database>\<instance>\trace` directory, as the following command shows:

```
SQL> show parameter user_dump_dest

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
user_dump_dest                       string      c:\app\ora\diag\rdbms\orcl1\or
                                                 cl1\trace
SQL>
```

In Oracle Database 11g, you don't have to set the `max_dump_file_size` parameter to specify the maximum size of a trace file.

10-2. Tracing a Specific SQL Statement

Problem

You want to trace a specific SQL statement, in order to find out where the database is spending its time during the execution of the statement.

Solution

In an Oracle 11.1or higher release, you can use the enhanced SQL tracing interface to trace one or more SQL statements. Here are the steps to tracing a set of SQL statements.

1. Issue the `alter session set events` statement, as shown here, to set up the trace.

```
SQL> alter session set events 'sql_trace level 12';

Session altered.
SQL>
```

2. Execute the SQL statements.

```
SQL> select count(*) from sales;
```

3. Set tracing off.

```
SQL> alter session set events 'sql_trace off';

Session altered.

SQL>
```

You can choose to trace specific SQL statements by specifying the SQL ID of a statement in the `alter session set events` statement. Here are the steps:

1. Find the SQL ID of the SQL statement by issuing this statement:

```
 SQL> select sql_id,sql_text
      from v$sql
      where sql_text='select sum(quantity_sold) from sales';

SQL_ID                        SQL_TEXT
----------------          ----------------------------------
fb2yuOp1kgvhr             select sum(quantity_sold) from sales

SQL>
```

2. Set tracing on for the specific SQL statement whose SQL ID you've retrieved.

```
SQL> alter session set events 'sql_trace [sql:fb2yuOp1kgvhr] level 12';

Session altered.

SQL>
```

3. Execute the SQL statement.

```
SQL> select sum(quantity_sold) from sales;

SUM(QUANTITY_SOLD)
------------------
            918843
```

4. Turn off tracing.

```
SQL> alter session set events 'sql_trace[sql:fb2yuOp1kgvhr] off';

Session altered.

SQL>
```

You can trace multiple SQL statements by separating the SQL IDs with the pipe (|) character, as shown here:

```
SQL> alter session set events 'sql_trace [sql: fb2yuOp1kgvhr|4v433su9vvzsw]';
```

You can trace a specific SQL statement running in a different session by issuing an `alter system set events` statement:

```
SQL> alter system set events 'sql_trace[sql:fb2yuOp1kgvhr] level 12';

System altered.
SQL>
```

You can get the SQL ID for the statement by querying the `V$SQL` view as shown earlier in this recipe, or you can get it through the Oracle Enterprise Manager. Once the user in the other session completes executing the SQL statement, turn off tracing with the following command:

```
SQL> alter system set events 'sql_trace[sql:fb2yuOp1kgvhr] off';

System altered.

SQL>
```

How It Works

In Oracle Database 11g, you can set the Oracle event `SQL_TRACE` to trace the execution of one or more SQL statements. You can issue either an `alter session` or an `alter system` statement for tracing a specific SQL statement. Here's the syntax of the command:

```
alter session/system set events 'sql_trace [sql:<sql_id>|<sql_id>] … event specification';
```

Even if you execute multiple SQL statements before you turn the tracing off, the trace file will show just the information pertaining to the SQL_ID or SQL_IDs you specify.

10.3. Enabling Tracing in Your Own Session

Problem

You want to trace your own session.

Solution

Normal users can use the `DBMS_SESSION` package to trace their sessions, as shown in this example:

```
SQL>execute dbms_session.session_trace_enable(waits=>true, binds=> false);
```

To disable tracing, the user must execute the `session_trace_disable` procedure, as shown here:

```
SQL> execute dbms_session.session_trace_disable();
```

How It Works

The `DBMS_MONITOR` package, which Oracle recommends for all tracing, is executable only by a user with the DBA role. If you don't have the DBA role, you can use the `dbms_session.session_trace_enable` procedure to trace your own session.

10-4. Finding the Trace Files

Problem

You'd like to find a way to easily identify your trace files.

Solution

Issue the following statement to set an identifier for your trace files, before you start generating the trace:

```
SQL> alter session set tracefile_identifier='MyTune1';
```

To view the most recent trace files the database has created, in Oracle Database 11.1 and newer releases, you can query the Automatic Diagnostic Repository (ADR) by executing the following command (see Chapter 5 for details on the `adrci` utility):

```
adrci> show tracefile -t
08-MAY-11 19:01:48  diag\rdbms\orcl1\orcl1\trace\orcl1_p000_8652_MyTune1.trc
08-MAY-11 19:01:48  diag\rdbms\orcl1\orcl1\trace\orcl1_p001_6424_MyTune1.trc
08-MAY-11 19:01:48  diag\rdbms\orcl1\orcl1\trace\orcl1_p002_5980_MyTune1.trc
adrci>
```

To find out the path to your current session's trace file, issue the following command:

```
SQL>  select value from v$diag_info
      where name = 'Default Trace File';

VALUE
-------------------------------------------------------------------------

c:\app\ora\diag\rdbms\orcl1\orcl1\trace\orcl1_ora_11248_My_Tune1.trc

SQL>
```

To find all trace files for the current instance, issue the following query:

```
SQL> select value from v$diag_info where name = 'Diag Trace'
```

How It Works

Often, it's hard to find the exact trace file you're looking for, because there may be a bunch of other trace files in the trace directory, all with similar-looking file names. A best practice during SQL tracing is to associate your trace files with a unique identifier. Setting an identifier for the trace files you're going to generate makes it easy to identify the SQL trace files from among the many trace files the database generates in the trace directory.

You can confirm the value of the trace identifier with the following command:

```
SQL> sho parameter tracefile_identifier
NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------------
tracefile_identifier                 string      MyTune1
SQL>
```

The column `TRACEID` in the `V$PROCESS` view shows the current value of the `tracefile_identifier` parameter as well. The trace file identifier you set becomes part of the trace file name, making it easy to pick the correct file name for a trace from among a large number of trace files in the trace directory. You can modify the value of the `tracefile_identifier` parameter multiple times for a session. The trace file names for a process will contain information to indicate that they all belong to the same process.

Once you set the `tracefile_identifier` parameter, the trace files will have the following format, where `sid` is the Oracle SID, `pid` is the process ID, and `traceid` is the value you've set for the `tracefile_identifier` initialization parameter.

```
sid_ora_pid_traceid.trc
```

10-5. Examining a Raw SQL Trace File

Problem

You want to examine a raw SQL trace file.

Solution

Open the trace file in a text editor to inspect the tracing information. Here are portions of a raw SQL trace generated by executing the `dbms_monitor.session_trace_enable` procedure:

```
PARSING IN CURSOR #3 len=490 dep=1 uid=85 oct=3 lid=85 tim=269523043683 hv=672110367
ad='7ff18986250' sqlid='bqasjasn0z5sz'

PARSE #3:c=0,e=647,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,plh=0,tim=269523043680
EXEC #3:c=0,e=1749,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,plh=3969568374,tim=269523045613
WAIT #3: nam='Disk file operations I/O' ela= 15833 FileOperation=2 fileno=4 filetype=2 obj#=-1
tim=269523061555
FETCH #3:c=0,e=19196,p=0,cr=46,cu=0,mis=0,r=1,dep=1,og=1,plh=3969568374,tim=269523064866
STAT #3 id=3 cnt=12 pid=2 pos=1 obj=0 op='HASH GROUP BY (cr=46 pr=0 pw=0 time=11 us cost=4
size=5317 card=409)'
STAT #3 id=4 cnt=3424 pid=3 pos=1 obj=89079 op='TABLE ACCESS  FULL DEPT (cr=16 pr=0 pw=0
time=246 us cost=3 size=4251 card=327)'
```

As you can see from this excerpt of the raw trace file, you can glean useful information, such as parse misses, waits, and the execution plan of the SQL statement.

How It Works

The usual practice after getting a session trace file is to analyze it using a tool such as `TKPROF`. However, you can examine a trace file by visually reading the trace output. The raw trace files capture information for each of the following three steps of SQL statement processing:

Parse: During this stage, the database converts the SQL statement into an execution plan, and checks for authorization and the existence of tables and other objects.

Execute: The database executes the SQL statement during this phase. For a `SELECT` statement, the execute phase identifies the rows the database must retrieve. The database modifies the data for DML statements such as insert, update, and delete.

Fetch: This step applies only for a `SELECT` statement. During this phase, the database retrieves the selected rows.

A SQL trace file will contain detailed statistics for each of the three phases of execution, in addition to wait event information. You usually format the raw trace files with a utility such as `TKPROF`. However, there are times when a raw trace file can show you useful information very quickly, by a simple scroll through the file. A locking situation is a good example where you can visually inspect a raw trace file. `TKPROF` doesn't provide you details about latches and locks (enqueues). If you suspect that a query was waiting on a lock, digging deep into a raw trace file shows you exactly where and why a query was

waiting. In the `WAIT` line, the elapsed time (`ela`) shows the amount of time waited (in microseconds). In our example, elapsed wait time for "Disk file operations I/O" is 15,833 microseconds. Since 1 second=1,000,000 microseconds, this is not a significant wait time. The raw trace file clearly shows if an I/O wait event, as is true in this case, or another type of wait event held up the query. If the query was waiting on a lock, you'll see something similar to the following: `WAIT #2: nam='enqueue ela-300….`

We've purposefully kept the discussion short in this recipe, because tools such as `TKPROF` and the Oracle Trace Analyzer provide you sophisticated diagnostic information by profiling the raw trace files.

10-6. Analyzing Oracle Trace Files

Problem

You want to know how to analyze an Oracle trace file.

Solution

There are multiple ways to interpret a SQL trace file. Here are the different approaches:

- Read the raw SQL trace file in a text editor.
- Use the Oracle-provided `TKPROF` (Trace Kernel Profiler) utility.
- Use Oracle Trace Analyzer, a free product you can download from Oracle Support.
- Use third-party tools.

How It Works

Getting a SQL trace is often the easy part—analyzing it is, of course, more of a task than collecting the trace. Sometimes, if you're particularly adept at it, you can certainly directly view the source trace file itself, but in most cases, you need a tool to interpret and profile the huge amount of data that a trace file can contain. Note that the `TKPROF` or other profiling tools show the elapsed times for various phases of query execution, but not the information for locks and latches. If you're trying to find out if any locks are slowing down a query, look at the raw trace files to see if there are any enqueue waits in the `WAIT` lines of the raw file.

You can easily read certain trace files such as the trace file for event 10053, since the file doesn't contain any SQL execution statistics (such as parsing, executing, and fetching statistics), and no wait event analysis—it mostly consists of a trace of the execution path used by the cost-based optimizer (CBO). However, for any SQL execution trace files, such as those you generate with the event 10046, a visual inspection of the trace file, while technically possible, is not only time-consuming, but the raw data is not in a summary form and key events are often described in obscure ways. Therefore, using a profiler such as the `TKPROF` utility is really your best option.

The `TKPROF` utility is an Oracle-supplied profiling tool that most Oracle DBAs use on a routine basis. Recipes 10-7 and 10-8 show how to use `TKPROF`.

Oracle's Trace Analyzer is free (you have to download it from Oracle Support), easy to install and use, and produces clear reports with plenty of useful diagnostic information. You do have to install the tool first, but it takes only a few minutes to complete the installation. Thereafter, you just pass the name of the trace file to a script to generate the formatted output. Recipe 10-9 shows how to install and use the Oracle Trace Analyzer.

There are also third-party profiling tools that offer features not found in the `TKPROF` utility. Some of these tools generate pretty HTML trace reports and some include charts as well to help you visually inspect the details of the execution of the SQL statement that you've traced. Note that in order to use some of these products, you'll have to upload your trace files for analysis. If your trace files contain sensitive data or security information, this may not work for you.

10-7. Formatting Trace Files with TKPROF

Problem

You've traced a session, and you want to use `TKPROF` to format the trace file.

Solution

You run the `TKPROF` utility from the command line. Here's an example of a typical `tkprof` command for formatting a trace file.

```
$ tkprof user_sql_001.trc user1.prf explain=hr/hr table=hr.temp_plan_table_a sys=no
  sort=exeela,prsela,fchela
```

In the example shown here, the `tkprof` command takes the `user_sql_001.trc` trace file as input and generates an output file named `user1.prf.` The "How it Works" section of this recipe explains key optional arguments of the `TKPROF` utility.

How It Works

`TKPROF` is a utility that lets you format any extended trace files that you generate with the event 10046 or through the `DBMS_MONITOR` package. You can use this tool to generate reports for analyzing results of the various types of SQL tracing explained in this chapter. You can run `TKPROF` on a single trace file or a set of trace files that you've concatenated with the `trcsess` utility. `TKPROF` shows details of various aspects of SQL statement execution, such as the following:

- SQL statement text
- SQL trace statistics
- Number of library cache misses during the parse and execute phases
- Execution plans for all SQL statements
- Recursive SQL calls

You can view a list of all the arguments you can specify issuing the `tkprof` command without any arguments, as shown here:

```
$ tkprof
Usage: tkprof tracefile outputfile [explain= ] [table= ]
              [print= ] [insert= ] [sys= ] [sort= ]
…
```

Here's a brief explanation of the important arguments you can specify with the `tkprof` command:

`filename1`: Specifies the name of the trace file

`filename2`: Specifies the formatted output file

`waits`: Specifies whether the output file should record a summary of the wait events; default is *yes*.

`sort`: By default, `TKPROF` lists the SQL statements in the trace file in the order they were executed. You can specify various options with the `sort` argument to control the order in which `TKPROF` lists the various SQL statements.

- `prscpu`: CPU time spent parsing
- `prsela`: Elapsed time spent parsing
- `execpu`: CPU time spent executing
- `exeela`: Elapsed time spent executing
- `fchela`: Elapsed time spent fetching

`print`: By default `TKPROF` will list all traced SQL statements. By specifying a value for the `print` option, you can limit the number of SQL statements listed in the output file.

`sys`: By default `TKPROF` lists all SQL statements issued by the user SYS, as well as recursive statements. Specify the value `no` for the `sys` argument to make `TKPROF` omit these statements.

`explain`: Writes execution plans to the output file; `TKPROF` connects to the database and issues explain plan statements using the username and password you provide with this parameter.

`table`: By default, `TKPROF` uses a table named `PLAN_TABLE` in the schema of the user specified by the explain parameter, to store the execution plans. You can specify an alternate table with the table parameter.

`width`: This is an integer that determines the output line widths of some types of output, such as the explain plan information.

10-8. Analyzing TKPROF Output

Problem

You've formatted a trace file with `TKPROF`, and you now want to analyze the `TKPROF` output file.

Solution

Invoke the TKPROF utility with the tkprof command as shown here:

```
c:\>tkprof orcl1_ora_6448_mytrace1.trc ora6448.prf explain=hr/hr sys=no
sort=prsela,exeela,fchela

TKPROF: Release 11.2.0.1.0 - Development on Sat May 14 11:36:35 2011

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.

c:\app\ora\diag\rdbms\orcl1\orcl1\trace>
```

In this example, orcl1_ora_6448_mytrace1.trc is the trace file you want to format. The ora6448.prf file is the TKPROF output file. The "How it Works" section that follows shows how to interpret a TKPROF output file.

How It Works

In our example, there's only a single SQL statement. Thus, the sort parameters (prsecla, exeela, fchela) don't really matter, because they come into play only when TKPROF needs to list multiple SQL statements. Here's a brief description of the key sections in a TKPROF output file.

Header

The header section shows the trace file name, the sort options, and a description of the terms used in the output file.

```
Trace file: orcl1_ora_6448_mytrace1.trc
Sort options: prsela  exeela  fchela
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************
```

Execution Statistics

TKPPROF lists execution statistics for each SQL statement in the trace file. TKPROF lists the execution statistics for the three steps that are part of SQL statement processing: parse, execute, and fetch.

```
call   count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.03          0         64          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch     5461      0.29       0.40          0       1299          0       81901
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     5463      0.31       0.43          0       1363          0       81901
```

The following is what the SQL execution statistics in the table stand for:

`count`: The number of times the database parsed, executed, or fetched this statement

`cpu`: The CPU time used for the parse/execute/fetch phases

`elapsed`: Total elapsed time (in seconds) for the parse/execute/fetch phases

`disk`: Number of physical block reads for the parse/execute/fetch phases

`query`: Number of data blocks read with logical reads from the buffer cache in consistent mode for the parse/fetch/execute phases (for a `select` statement)

`current`: Number of data blocks read and retrieved with logical reads from the buffer cache in current mode (for `insert`, `update`, `delete`, and `merge` statements)

`rows`: Number of fetched rows for a `select` statement or the number of rows inserted, deleted, or updated, respectively, for an `insert`, `delete`, `update`, or `merge` statement

Row Source Operations

The next section of the report shows the number of misses in the library cache, the current optimizer mode, and the row source operations for the query. Row source operations show the number of rows that the database processes for each operation such as joins or full table scans.

```
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 85  (HR)

Rows     Row Source Operation
-------  ---------------------------------------------------
  81901  HASH JOIN  (cr=1299 pr=0 pw=0 time=3682295 us cost=22 size=41029632 card=217088)
   1728   TABLE ACCESS FULL DEPT (cr=16 pr=0 pw=0 time=246 us cost=6 size=96768 card=1728)
   1291   TABLE ACCESS FULL EMP (cr=1283 pr=0 pw=0 time=51213 us cost=14 size=455392
card=3424)
```

The "Misses in library cache during parse" indicates the number of hard parses during the parse and execute database calls. In the `Row Source Operation` column, the output includes several statistics for each row source operation. These statistics quantify the various types of work performed during a row source operation. The following is what the different statistics stand for (you may not see all of these for every query):

`cr`: Blocks retrieved through a logical read in the consistent mode

`pr`: Number of blocks read through a physical disk read

`pw`: Number of blocks written with a physical write to a disk

`time`: Total time (in milliseconds) spent processing the operation

`cost`: Estimated cost of the operation

`size`: Estimated amount of data (bytes) returned by the operation

`card`: Estimated number of rows returned by the operation

The Execution Plan

If you specified the `explain` parameter when issuing the `tkprof` command, you'll find an execution table showing the execution plan for each SQL statement.

```
Rows     Execution Plan
-------  ---------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
  81901   HASH JOIN
   1728    TABLE ACCESS (FULL) OF 'DEPT' (TABLE)
   1291    TABLE ACCESS (FULL) OF 'EMP' (TABLE)
```

In our example, the execution plan shows that there were two full table scans, and a hash join following it.

Wait Events

You'll see the wait events section only if you've specified `waits=>true` in your trace command. The wait events table summarizes waits during the trace period:

```
Elapsed times include waiting on following events:
  Event waited on                          Times   Max. Wait  Total Waited
  ---------------------------------------  Waited  ----------  ------------
    SQL*Net message from client               5461      112.95        462.81
    db file sequential read                      1        0.05          0.05

********************************************************************************
```

In this example, the `SQL*Net message to client` waits account for most of the waits, but these are idle waits. If you see wait events such as the db file sequential read event or the db file scattered read event with a significant number of waits (and/or total wait time), you need to investigate those wait events further.

Note that the `TKPROF` output doesn't show you information about any bind variables. It also doesn't show any waits due to enqueue locks.

10-9. Analyzing Trace Files with Oracle Trace Analyzer

Problem

You want to use the Oracle Trace Analyzer to analyze trace files.

Solution

The Oracle Trace Analyzer, also known as TRCANLZR or TRCA, is a SQL trace profiling tool that's an alternative to the `TKPROF` utility. You must download the TRCA from Oracle Support. Once you download TRCA, unzip the files and install TRCA by executing the `/trca/install/trcreate.sql` script.

Once you install TRCA, you must log in as a user with the SYSDBA privilege to execute the `tacreate.sql` script. The `tacreate.sql` generates the formatted output files for any traces you've generated. The script asks you for information relating to the location of the trace files, the output file, and the tablespace where you want TRCA to store its data.

Here are the steps for installing and running TRCA.

1. Installing TRCA is straightforward, so we just show you a summary of the installation here:

```
SQL> @tacreate.sql
Uninstalling TRCA, please wait
TADOBJ completed.
SQL>
SQL> WHENEVER SQLERROR EXIT SQL.SQLCODE;
SQL> REM If this DROP USER command fails that means a session is connected wi
this user.
SQL> DROP USER trcanlzr CASCADE;
SQL> WHENEVER SQLERROR CONTINUE;
SQL>
SQL> SET ECHO OFF;
TADUSR completed.
TADROP completed.

Creating TRCA$ INPUT/BDUMP/STAGE Server Directories
...

TACREATE completed. Installation completed successfully.
SQL>
```

2. Set up tracing.

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

System altered.

SQL>
```

3. Execute the SQL statement you want to trace.

```
SQL> select …
```

4. Turn off tracing.

```
SQL> alter session set events '10046 trace name context off';
System altered.

SQL>
```

5. Run the `/trca/run/trcanlzr` script (`START trcanlzr.sql`) to profile the trace you've just generated. You must pass the trace file name as input to this script:

```
c:\trace\trca\trca\run>sqlplus hr/hr
SQL> START trcanlzr.sql orcl1_ora_7460_mytrace7.trc
Parameter 1:
Trace Filename or control_file.txt (required)
Value passed to trcanlzr.sql:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TRACE_FILENAME: orcl1_ora_7460_mytrace7.trc
Analyzing orcl1_ora_7460_mytrace7.trc
... analyzing trace(s) ...
Trace Analyzer completed.
Review first trcanlzr_error.log file for possible fatal errors.
TKPROF: Release 11.2.0.1.0 - Development on Sat May 14 15:59:13 2011
  …
   233387  05/14/2011 15:59   trca_e21106.html
   115885  05/14/2011 15:59   trca_e21106.txt
File trca_e21106.zip has been created
TRCANLZR completed.
SQL>
c:\trace\trca\trca\run>
```

You can now view the profiled trace data in text or HTML format—TRCA provides both of these in the ZIP file that it creates when it completes profiling the trace file. TRCA places the ZIP file in the directory from which you run the `/trca/run/trcanlzr.sql` script.

How It Works

Oracle Support Center of Expertise (CoE) provides the TRCA diagnostic tool. Although many DBAs are aware of the TRCA, few use it on regular basis. Some of us have used it in response to a request by Oracle Support. As you learned in the "Solution" section, the TRCA tool accepts a SQL trace generated by you and outputs a diagnostic report in both text and HTML formats. The TRCA tool also provides you a `TKPROF` report (it executes the `tkprof` command as part of its diagnostic data collection). Since TRCA provides a rich set of diagnostic information, consider using it instead of `TKPROF`.

Apart from the data normally collected by the `TKPROF` utility, TRCA also identifies expensive SQL statements and gathers their explain plans. It also shows the optimizer statistics and configuration parameters that have a bearing on the performance of the SQL statements in the trace.

■ **Tip** Use TRCA instead of `TKPROF` for analyzing your trace files—it provides you a wealth of diagnostic information, besides giving you a `TKPROF` output file as part of the bargain.

In our example, we used TRCA to format a trace file on the same system where we generated the trace. However, if you can't install TRCA in a production system, not to worry. TRCA can also analyze production traces using a different system. The details are in the `trca_instructions` HTML document, which is part of the TRCA download ZIP file.

Here are the major sections of a TRCA report, and as you can see already, the report offers a richer set of diagnostic information than that offered by `TKPROF`.

Summary: Provides a breakdown of elapsed time, response time broken down into CPU and non-idle wait time, and other response time-related information

Non-Recursive Time and Totals: Provides a breakdown of response time and elapsed time during the parse, execute, and fetch steps; the report also contains a table that provides total and average waits for each idle and non-idle wait event.

Top SQL: Provides detailed information about SQL statements that account for the most response time, elapsed time, and CPU time, as shown in the following extract from the report:

```
There are 2 SQL statements with "Response Time Accounted-for" larger than threshold of
10.0% of the "Total Response Time Accounted-for".
These combined 2 SQL statements are responsible for a total of 99.3% of the "Total
Response Time Accounted-for".

There are 3 SQL statements with "Elapsed Time" larger than threshold of 10.0% of the
"Total Elapsed Time".
These combined 3 SQL statements are responsible for a total of 75.5% of the "Total Elapsed
Time".

There is only one SQL statement with "CPU Time" larger than threshold of 10.0% of the
"Total CPU Time".
```

Individual SQL: This is a highly useful section, as it lists all SQL statements and shows their elapsed time, response time, and CPU time. It provides the hash values and SQL IDs of each statement.

SQL Self - Time, Totals, Waits, Binds and Row Source Plan: Shows parse, execute, and fetch statistics for each statement, similar to the `TKPROF` utility; it also shows the wait event breakdown (average and total times) for each statement. There's also a very nice explain plan for each statement, which shows the time and the cost of each execution step.

Tables and Indexes: Shows the number of rows, partitioning status, the sample size, and the last time the object was analyzed; for indexes, it additionally shows the clustering factor and the number of keys.

Summary: Shows I/O related wait (such as the db file sequential read event) information including average and total waits, for tables and indexes

Hot I/O Blocks: Shows the list of blocks with the largest wait time or times waited

Non-default Initialization Parameters: Lists all non-default initialization parameters

As this brief review of TRCA shows, it's a far superior tool than `TKPROF`. Besides, if you happen to love `TKPROF` reports, it includes them as well in its ZIP file. So, what are you waiting for? Download the TRCA and benefit from its rich diagnostic profiling of problem SQL statements.

10-10. Tracing a Parallel Query

Problem

You'd like to trace a parallel query.

Solution

You can get an event 10046 trace for a parallel query in the same way as you would for any other query. The only difference is that the 10046 event will generate as many trace files as the number of parallel query servers. Here's an example:

```
SQL>alter session set tracefile_identifier='MyTrace1';

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

Session altered.

SQL> select /*+ full(sales) parallel (sales 6) */ count(quantity_sold) from
     sales;

COUNT(QUANTITY_SOLD)
--------------------
              918843

SQL> alter session set events '10046 trace name context off';

Session altered.

SQL>
```

You'll now see a total of seven trace files with the trace file identifier `MyTrace1` in the trace directory. Depending on what you're looking for, you can analyze each of the trace files separately, or consolidate them into one big trace file with the `trcsess` utility before analyzing it with `TKPROF` or another profiler such as the Oracle Trace Analyzer. You'll also find several files with the suffix `.trm` in the trace directory—you can ignore these files, as they are for use by the database.

How It Works

The only real difference between getting an extended trace for a single query and one for a parallel query is that you'll have multiple trace files, one for each parallel query server. When a user executes a parallel query, Oracle creates multiple parallel query processes to process the query, with each process getting its own session. That's the reason Oracle creates multiple trace files for a parallel query.

Once you turn off the trace, go to the trace directory and execute the following command to find all the trace files for the parallel query:

```
$ find . -name '*MyTrace1*'
```

The `find` command lists all the trace files for your parallel query (ignore the files ending with `.trm` in the trace directory). You can move the trace files to another directory and use the `trcsess` utility to consolidate those files, as shown here:

```
$ trcsess output=MyTrace1.trc clientid='px_test1' orcl1_ora_8432_mytrace1.trc
orcl1_ora_8432_mytrace2.trc
```

You're now ready to use the `TKPROF` utility to profile the parallel query.

When you issue a parallel query, the parallel execution coordinator/query coordinator (QC) controls the execution of the query. The parallel execution servers/slaves (QS) do the actual work. The parallel execution server *set* is the set of all the query servers that execute an operation. The query coordinator and each of the execution servers generate their own trace files. For example, if one of the slave processes waits for a resource, the database records the resulting wait events in that slave process's trace file, but not in the query coordinator's trace file.

Note that if you're using a 10.2 or an older release, the trace files for the user process will be created in the user dump directory and the background processes (slaves) will generate trace files in the background dump directory. In 11.1 and newer databases, the trace files for both background and user processes are in the same directory (trace). You can find the trace file names by issuing the `show tracefile -t` command after invoking ADRCI, or by querying the `V$DIAG_INFO` view from SQL*Plus.

10-11. Tracing Specific Parallel Query Processes

Problem

You want to trace one or more specific parallel query processes.

Solution

Identify the parallel query processes you want to trace with the following command.

```
SQL> select inst_id,p.server_name,
     p.status as p_status,
     p.pid as p_pid,
     p.sid as p_sid
     from  gv$px_process p
     order by p.server_name;
```

Let's say you decide to trace the processes p002 and p003. Issue the following `alter system set events` command to trace just these two parallel processes.

```
SQL> alter system set events 'sql_trace  {process: pname = p002 | p003}';
```

Once you're done tracing, turn off the trace by issuing the following command:

```
SQL> alter system set events 'sql_trace  {process: pname = p002 | p003} off';
```

How It Works

Tracing parallel processes is always tricky. One of the improvements made to the tracing infrastructure in the Oracle Database 11g release is the capability to trace a specific statement or a set of statements. This capability comes in handy when there are a large number of SQL statements being executed by a session and you're sure about the identity of the SQL statement whose execution you want to trace.

10-12. Tracing Parallel Queries in a RAC System

Problem

You're tracing a parallel query in a RAC environment, but aren't sure in which instance the trace files are located.

Solution

Finding the trace files for the server (or thread or slave) processes is sometimes difficult in a RAC environment, because you aren't sure on which node or node(s) the database has created the trace files. Here are the steps to follow to make it easier to find the trace files on the different nodes.

1. Set the `px_trace` with an `alter session` command, to help identify the trace files, as shown here:

```
SQL> alter session set tracefile_identifier='10046';
SQL> alter session set  "_px_trace" = low , messaging;
SQL> alter session set events '10046 trace name context forever,level 12';
```

2. Execute your parallel query.

```
SQL> alter table bigsales (parallel 4);
SQL> select count(*) from bigsales;
```

3. Turn all tracing off.

```
SQL> alter session set events '10046 trace name context off';
SQL> alter session set "_px_trace" = none;
```

Specifying `px_trace` will cause the query coordinator's trace file to include information about the slave processes that are part of the query, and the instance each slave process belongs to. You can then retrieve the trace files from the instances listed in the query coordinator's trace file.

How It Works

The `_px_trace` (px trace) parameter is an undocumented, internal Oracle parameter that has existed since the 9.2 release. Once you run the trace commands as shown in the "Solution" section of this recipe, the trace file for the query coordinator (QC) process will show within it the name of each of the slave processes and the instances the processes have run on—for example:

```
Acquired 4 slaves on 1 instances avg height=4 in 1 set q serial:2049
        P000 inst 1 spid 7512
        P001 inst 1 spid 4088
        P002 inst 1 spid 7340
        P003 inst 1 spid 9256
```

In this case, you know that Instance 1 is where you must look to get the trace files for the slave processes P000, P001, P002, and P003. On Instance 1, in the ADR trace subdirectory, look for file names that contain the words P000 (or P001/P002/P003), to identify the correct trace files.

10-13. Consolidating Multiple Trace Files

Problem

You have generated multiple trace files for a session in order to tune performance, and you want to consolidate those files into a single trace file.

Solution

Use the `trcsess` command to merge multiple trace files into a single trace file. Here's a simple example:

```
c:\trace> trcsess output=combined.trc session=196.614 orcl1_ora_8432_mytrace1.trc
orcl1_ora_8432_mytrace2.trc
C:\trace>
```

The `trcsess` command shown here combines two trace files generated for a session into a single trace file. The session parameter identifies the session with a session identifier, consisting of the session index and session serial number, which you can get from the `V$SESSION` view.

How It Works

The `trcsess` utility is part of the Oracle database and helps by letting you consolidate multiple trace files during performance tuning and debugging exercises. Here's the syntax of the `trcsess` command:

```
trcsess [output=output_file_name]
           [session=session_id]
           [client_id=cleint_id]
           [service=service_name]
           [action=action_name]
           [module=module_name]
           [trace_files]
```

You must specify one of these five options when issuing the `trcess` command: `session`, `client_id`, `service`, `action`, and `module`. For example, if you issue the command in the following manner, the command includes all the trace files in the current directory for a session and combines them into a single file:

```
$ trcsess output=main.trc session=196.614
```

In our example, we specified the name of the consolidated trace file with the `output` option. If you don't specify the output option, `trcsess` prints the output to standard out. Once you use `trcsess` to combine the output of multiple trace files into one consolidated file, you can use the `TKPROF` utility to analyze the file, just as you'd do in the case of a single trace file.

10-14. Finding the Correct Session for Tracing

Problem

You want to initiate a session trace for a user from your own session, and you would like to find out the correct session to trace.

Solution

You must have the `SID` and the serial number for the user whose session you want to trace. You can find these from the `V$SESSION` view, of course, once you know the user's name. However, you must get several other details about the user's session to identify the correct session, since the user may have multiple sessions open. Use the following query to get the user's information:

```
SQL> select a.sid, a.serial#, b.spid, b.pid,
     a.username, a.osuser, a.machine
     from
     v$session a,
     v$process b
     where a.username IS NOT NULL
     and a.paddr=b.addr;
```

The query provides several attributes such as `USERNAME`, `OSUSER`, and `MACHINE`, which help you unambiguously select the correct session.

How It Works

You can't always rely on the first set of `SID` and serial number you manage to find for the user whose session you want to trace. Together, the `SID` and serial number uniquely identify a session. However, you may find multiple `SID` and serial number combinations for the same user, because your database may be using common user logins. Therefore, querying the `V$SESSION` view for other information such as `OSUSER` and `MACHINE` besides the `SID` and serial number helps to identify the correct user session.

`V$SESSION` view columns such as `COMMAND`, `SERVER`, `LOGON_TIME`, `PROGRAM`, and `LAST_CALL_ET` help identify the correct session to trace. If you still can't find the correct session, you may want to join the `V$SESSION` and `V$SQLAREA` views to identify the correct session.

10-15. Tracing a SQL Session

Problem

You want to turn on SQL tracing for a session to diagnose a performance problem.

Solution

There are multiple ways to trace a session, but the Oracle-recommended approach is to use the `DBMS_MONITOR` package to access the SQL tracing facility. To trace a session, first identify the session with the following command, assuming you know either the username or the SID for the session:

```
SQL> select sid, serial#, username from v$session;
```

Once you get the `SID` and `SERIAL#` from the previous query, invoke the `session_trace_enable` procedure of the `DBMS_MONITOR` package, as shown here:

```
SQL> execute dbms_monitor.session_trace_enable(session_id=>138,serial_num=>242,
waits=>true,binds=>false);
PL/SQL procedure successfully completed.
SQL>
```

■ **Caution** SQL tracing does impose an overhead on the database—you need to be very selective in tracing sessions in a production environment, as a trace can fill up a disk or affect CPU usage adversely.

In this example, we chose to trace the wait information as well, but it's optional. Once you execute this command, have the user execute the SQL statements that you're testing (in a dev or test environment). In a production environment, wait for a long enough period to make sure you've captured the execution of the SQL statements, before turning the tracing off. Invoke the `session_trace_disable` procedure to disable the SQL tracing for the session, as shown here:

```
SQL> execute dbms_monitor.session_trace_disable();
PL/SQL procedure successfully completed.
SQL>
```

Once you complete tracing the session activity, you can get the trace file for the session from the trace directory and use the `TKPROF` utility (or a different profiler) to get a report. The trace file will have the suffix `mytrace1`, the value you set as the trace file identifier.

To trace the current user session, use the following pair of commands:

```
SQL> execute dbms_monitor.session_trace_enable();
SQL> execute dbms_monitor.session_trace_disable();
```

How It Works

Tracing an entire session is expensive in terms of resource usage and you must do so only when you haven't identified a poorly performing SQL statement already. A session trace gathers the following types of information.

- Physical and logical reads for each statement that's running in the session
- CPU and elapsed times
- Number of rows processed by each statement
- Misses in the library cache
- Number of commits and rollbacks
- Row operations that show the actual execution plan for each statement
- Wait events for each SQL statement

■ **Note** You need the DBA role to execute procedures and functions in the `DBMS_MONITOR` package.

You can specify the following parameters for the `session_trace_enable` procedure:

`session_id`: Identifies the session you want to trace (`SID`); if you omit this, your own session will be traced.

`serial_num`: Serial number for the session

`waits`: Set it to true if you want to capture wait information (default = false).

`binds`: Set it to true to capture bind information (default=false).

`plan_stat`: Determines the frequency with which the row source statistics (execution plan and execution statistics) are dumped

All the parameters for the `session_trace_enable` procedure are self-evident, except the `plan_stat` parameter. You can set the following values for this parameter:

`never`: The trace file won't contain any information about row source operations.

`first_execution` (same as setting the `plan_stat` parameter to the value `null`): Row source information is written once, after the first execution of a statement.

`all_executions`: Execution plan and execution statistics are written for each execution of the cursor, instead of only when the cursor is closed.

Since an execution plan for a statement can change during the course of a program run, you may want to set the `plan_stat` parameter to the value `all_executions` if you want to capture all possible execution plans for a statement.

10-16. Tracing a Session by Process ID

Problem

You want to identify and trace a session using an operating system process ID.

Solution

Execute the `alter session` (or `alter system`) `set events` command to trace a session by its operating system process ID, which is shown by the `SPID` column in the `V$PROCESS` view. The general format of this command is as follows:

```
alter session set events 'sql_trace {process:pid}'
```

Here are the steps to tracing a session by its OS PID.

1. Get the OS process ID by querying the `V$PROCESS` view.

   ```
   SQL> select spid,pname from v$process;
   ```

2. Once you identify the SPID of the user, issue the following statement to start the trace for that session:

   ```
   SQL> alter session set events 'sql_trace {process:2714}';
   Session altered.
   SQL>
   ```

3. Turn off tracing the following way:

   ```
   SQL> alter session set events 'sql_trace {process:2714} off';
   Session altered.
   SQL>
   ```

   You can also execute the `set events` command in the following manner, to concatenate two processes:

   ```
   SQL> alter system set events 'sql_trace {process:2714|2936}';
   System altered.
   SQL> alter system set events 'sql_trace {process:2714|2936} off';
   System altered.
   SQL>
   ```

When you concatenate two processes, the database generates two separate trace files, one for each process, as shown here:

```
orcl1_ora_2714.trc
orcl1_ora_2936.trc
```

How It Works

In Oracle Database 11g, the `alter session set events` command has been enhanced to allow you to trace a process by specifying the process ID (PID), process name (PNAME), or the Oracle Process ID

(ORAPID). You can also use an `alter system` command with the same general syntax as well. Here's the syntax of the command:

```
alter session set events 'sql_trace {process : pid = <pid>, pname = <pname>, orapid =
<orapid>} rest of event specification'
```

The `V$PROCESS` view contains information about all currently active processes. In the `V$PROCESS` view, the following columns help you identify the three process-related values:

```
PID: the Oracle process identifier
SPID: the Operating System process identifier
PNAME: name of the process
```

In this recipe, we showed how to generate a trace file using the OS process identifier (`SPID` column in the `V$PROCESS` view). You can use the general syntax shown here to generate a trace using the PID or the process name.

10-17. Tracing Multiple Sessions

Problem

You want to trace multiple SQL sessions that belong to a single user.

Solution

You can trace multiple sessions that belong to a user by using the `client_id_trace_enable` procedure from the `DBMS_MONITOR` package. Before you can execute the `dbms_monitor.client_id_trace_enable` procedure, you must set the `client_identifier` for the session by using the `DBMS_SESSION` package, as shown here:

```
SQL> execute dbms_session.set_identifier('SH')
```

Once you set the client identifier as shown here, the `client_identifier` column in the `V$SESSION` view is populated. You can confirm the value of the `client_identifier` column by executing the following statement:

```
SQL> select sid, serial#,username from v$session where client_identifier='SH';
```

Now you can execute the `dbms_monitor.client_id_trace_enable` procedure:

```
SQL> execute dbms_monitor.client_id_trace_enable(client_id=>'SH', waits=>true, binds=>false);
```

You can disable the trace with the following command:

```
SQL> execute dbms_monitor.client_id_trace_disable(client_id=>'SH');
```

How It Works

Setting the `client_identifier` column lets you enable the tracing of multiple sessions, when several users may be connecting as the same Oracle user, especially in applications that use connection pools. The `client_id_trace_enable` procedure collects statistics for all sessions with a specific client ID.

Note that the client_id that you must specify doesn't have to belong to a currently active session. By default, the waits and binds parameters are set to false and you can set the tracing of both waits and binds by adding those parameters when you execute the client_id_trace_enable procedure:

```
 SQL> exec dbms_monitor.client_id_trace_enable('SH',true,true);

PL/SQL procedure successfully completed.
```

You can query the DBA_ENABLED_TRACES view to find the status of a trace that you executed with a client identifier. In this view, the column TRACE_TYPE shows the value CLIENT_ID and the PRIMARY_ID shows the value of the client identifier.

```
SQL> select trace_type, primary_id,waits,binds from dba_enabled_traces;

TRACE_TYPE       PRIMARY_ID        WAITS     BINDS
-------------    ---------------   --------  --------------
CLIENT_ID        SH                TRUE      TRUE
```

10-18. Tracing an Instance or a Database

Problem

You want to trace the execution of all SQL statements in the entire instance or database.

Solution

Use the dbms_monitor.database_trace_enable procedure to trace a specific instance or an entire database. Issue the following pair of commands to start and stop tracing for an *individual instance*.

```
SQL> execute dbms_monitor.database_trace_enable(instance_name=>'instance1');
SQL> execute dbms_monitor.database_trace_disable(instance_name=>'instance1');
```

You can optionally specify the waits and binds attributes. The following commands enable and disable SQL tracing at the *database level*:

```
SQL> execute dbms_monitor.database_trace_enable();
SQL> execute dbms_monitor.database_trace_disable();
```

You can also set the sql_trace initialization parameter to true to turn on and turn off SQL tracing, but this parameter is deprecated. Oracle recommends that you use the dbms_monitor (or the dbms_session) package for SQL tracing.

How It Works

Obviously, instance-level and database-level SQL tracing is going to impose a serious overhead, and may well turn out to be another source of performance problems! You normally don't ever have to do this—use the session-level tracing instead to identify performance problems. If you must trace an entire instance, because you don't know from which session a query may be executed, turn off tracing as soon as possible, to reduce the overhead.

10-19. Generating an Event 10046 Trace for a Session

Problem

You want to get an Oracle event 10046 trace for a session.

Solution

You can get an Oracle event 10046 trace, also called an extended trace, by following these steps:

1. Set your trace file identifier.

```
SQL> alter session set tracefile_identifier='10046';
```

2. Issue the following statement to start the trace.

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

3. Execute the SQL statement(s) that you want to trace.

```
SQL> select sum(amount_sold) from sales;
```

4. Turn tracing off with the following command:

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

You'll find the trace dump file in the trace directory that's specified by the `diagnostic_dest` parameter (`$DIAG_HOME/rdbms/db/inst/trace`). You can analyze this trace file with `TKPROF` or another utility such as the Oracle Trace Analyzer.

How It Works

Here's what the various keywords in the syntax for setting a 10046 trace mean:

`set events`: Sets a specific Oracle event, in this case, the event 10046

`10046`: Specifies when an action should be taken

`trace`: The database must take this action when the event (10046) occurs.

`name`: Indicates the type of dump or trace

`context`: Specifies that Oracle should generate a context-specific trace; if you replace `context` with `errorstack`, the database will not trace the SQL statement. It dumps the error stack when it hits the 10046 event.

`forever`: Specifying the keyword `forever` tells the database to invoke the action (trace) every time the event (10046) is invoked, until you disable the 10046 trace. If you omit the keyword `forever`, the action is invoked just once, following which the event is automatically disabled.

`level 12`: Specifies the trace level—in this case, it captures both bind and wait information.

While the Oracle event 10046 has existed for several years, this trace is identical to the trace you can generate with the `session_trace_enable` procedure of the `DBMS_MONITOR` package:

```
SQL> execute dbms_monitor.session_trace_enable(session_id=>99,
serial_num=>88,waits=>true,binds=>true);
```

Both the 10046 event and the `dbms_monitor.session_trace_enable` procedure shown here generate identical tracing information, called extended tracing because the trace includes wait and bind variable data.

If you aren't using the new diagnostic infrastructure (ADR) introduced in Oracle Database 11g, make sure you set the dump file size to the value `unlimited`, as the 10046 trace often produces very large trace files, and the database will truncate the dump file if there isn't enough space in the trace dump directory.

The level of tracing you specify for the 10046 trace determines which types of information is gathered by the trace. The default level is 1, which collects basic information. Level 4 allows you to capture the bind variable values, which are shown as `:bi`, `:b2`, and so on. You can see the actual values that Oracle substitutes for each of the bind variables. Level 8 provides all the information from a Level 1 trace plus details about all the wait events during the course of the execution of the SQL query. A Level 12 trace is a combination of the Level 4 and Level 8 traces, meaning it'll include both bind variable and wait information. Level 16 is new in Oracle Database 11g, and provides STAT line dumps for each execution of the query. Note that this is the same as setting the value `all_executions` for the `plan_level` parameter when you trace with the `dbms_monitor.session_trace_enable` procedure.

■ **Tip** If the session doesn't close cleanly before you disable tracing, your trace file may not include important trace information.

While getting a 10046 trace and analyzing it does provide valuable information regarding the usage of bind variables and the wait events, you need to be careful about when to trace sessions. If the instance as a whole is poorly performing, your tracing might even make performance worse due to overhead imposed by running the trace. In addition, it takes time to complete the trace, run it through `TKPROF` or some other profiler, and go through the dozens of SQL statements in the report. Here is a general strategy that has worked for us in our own work:

If you're diagnosing general performance problems, a good first step would be to get an AWR report, ideally taking several snapshots spaced 1–15 minutes apart. Often, you can identify the problem from a review of the AWR report. The report will highlight things such as inefficient SQL statements, contention of various types, memory issues, latching, and full table scans that are affecting performance. You can get all this information without running a 10046 trace.

Run a 10046 trace when a user reports a problem and you can't identify a problem through the AWR reports. If you've clearly identified a process that is performing poorly, you can trace the relevant session. You can also run this trace in a development environment to help developers understand the query execution details and tune their queries.

10-20. Generating an Event 10046 Trace for an Instance

Problem

You want to trace a problem SQL query, but you can't identify the session in advance. You would like to trace all SQL statements executed by the instance.

Solution

You can turn on tracing at the instance level with the following `alter system` command, after connecting to the instance you want to trace.

```
SQL> alter system set events '10046 trace name context forever,level 12';
```

The previous command enables the tracing of all sessions that start after you issue the command—it won't trace sessions that are already connected.

You disable the trace by issuing the following command:

```
SQL> alter system set events '10046 trace name context off';
```

This command disables tracing for all sessions.

How It Works

Instance-wide tracing helps in cases where you know a problem query is running, but there's no way to identify the session ahead of time. Make sure that you enable instance-wide tracing only when you have no other alternative, and turn it off as soon as you capture the necessary diagnostic information. Any instance-wide tracing is going to not only generate very large trace files in a busy environment, but also contribute significantly to the system workload.

10-21. Setting a Trace in a Running Session

Problem

You want to set a trace in a session, but the session has already started.

■ **Note** A user who phones to ask for help with a long-running query is a good example of a case in which you might want to initiate a trace in a currently executing session. Some business-intelligence queries, for example, run for dozens of minutes, even hours, so there is time to initiate a trace mid-query and diagnose a performance problem.

Solution

You can set a trace in a running session using the operating system process ID (PID), with the help of the `oradebug` utility. Once you identify the PID of the session you want to trace, issue the following commands to trace the session.

```
SQL> connect / as sysdba
SQL> oradebug setospid <SPID>
SQL> oradebug unlimit
SQL> oradebug event 10046 trace name context forever,level 12
SQL> oradebug event 10046 trace name context off
```

In the example shown here, we specified Level 12, but as with the 10046 trace you set with the `alter session` command, you can specify the lower tracing levels 4 or 8.

How It Works

The `oradebug` utility comes in handy when you can't access the session you want to trace, or when the session has already started before you can set tracing. `oradebug` lets you attach to the session and start the SQL tracing. If you aren't sure about the operating system PID (or SPID) associated with an Oracle session, you can find it with the following query.

```
SQL> select p.PID,p.SPID,s.SID
  2  from v$process p,v$session s
  3  where s.paddr = p.addr
  4* and s.sid = &SESSION_ID
```

`oradebug` is only a facility that allows you to set tracing—it's not a tracing procedure by itself. The results of the 10046 trace you obtain with the `oradebug` command are identical to those you obtain with a normal event 10046 trace command.

In the example shown in the "Solution" section, we use the OS PID of the Oracle users. You can also specify the Oracle Process Identifier (PID) to trace a session, instead of the OS PID.

```
SQL> connect / as sysdba
SQL> oradebug setorapid 9834
SQL> oradebug unlimit
SQL> oradebug event 10046 trace name context forever,level 12
```

In an Oracle RAC environment, as is the case with all other types of Oracle tracing, make sure you connect to the correct instance before starting the trace. As an alternative to using `oradebug`, you can use the `dbms_system.set_sql_trace_in_session` procedure to set a trace in a running session. Note that `DBMS_SYSTEM` is an older package, and the recommended way to trace sessions starting with the Oracle Database 10g release is to use the `DBMS_MONITOR` package.

10-22. Enabling Tracing in a Session After a Login

Problem

You want to trace a user's session, but that session starts executing queries immediately after it logs in.

Solution

If a session immediately begins executing a query after it logs in, it doesn't give you enough time to get the session information and start tracing the session. In cases like this, you can create a *logon trigger* that automatically starts tracing the session once the session starts. Here is one way to create a logon trigger to set up a trace for sessions created by a specific user:

```
SQL> create or replace trigger trace_my_user
  2    after logon on database
  3    begin
  4     if user='SH' then
  5     dbms_monitor.session_trace_enable(null,null,true,true);
  6     end if;
  8*   end;
SQL> /

Trigger created.

SQL>
```

Before creating the logon trigger, make sure that you grant the user the `alter session` privileges, as shown here:

```
SQL> grant alter session to sh,hr;

Grant succeeded.
SQL>
```

How It Works

Often, you find it hard to trace session activity because the session already starts executing statements before you can set up the trace. This is especially so in a RAC environment, where it is harder for the DBA to identify the instance and quickly set up tracing for a running session. A logon trigger is the perfect solution for such cases. Note that in a RAC environment, the database generates the trace files in the trace directory of the instance to which a user connected.

A logon trigger for tracing sessions is useful for tracing SQL statements issued by a specific user, by setting the trace as soon as the user logs in. From that point on, the database traces all SQL statements issued by that user. Make sure you disable the tracing and drop the logon trigger once you complete tracing the SQL statements you are interested in. Remember to revoke the `alter session` privilege from the user as well.

10-23. Tracing the Optimizer's Execution Path

Problem

You want to trace the cost-based optimizer (CBO) to examine the execution path for a SQL statement.

Solution

You can trace the optimizer's execution path by setting the Oracle event 10053. Here are the steps.

1. Set the trace identifier for the trace file.

```
SQL> alter session set tracefile_identifier='10053_trace1'
Session altered.
SQL>
```

2. Issue the `alter session set events` statement to start the trace.

```
SQL> alter session set events '10053 trace name context forever,level 1';
Session altered.
SQL>
```

3. Execute the SQL statement whose execution path you want to trace.

```
SQL> select * from users
  2  where user_id=88 and
  3  account_status='OPEN'
  4  and username='SH';
…
SQL>
```

4. Turn the tracing off.

```
SQL> alter session set events '10053 trace name context off';
Session altered.
SQL>
```

You can examine the raw trace file directly to learn how the optimizer went about its business in selecting the execution plan for the SQL statement.

How It Works

An event 10053 trace gives you insight into the way the optimizer does its job in selecting the optimal execution plan for a SQL statement. For example, you may wonder why the optimizer didn't use an index in a specific case—the event 10053 trace shows you the logic used by the optimizer in skipping that index. The optimizer considers the available statistics for all objects in the query and evaluates various join orders and access paths. The event 10053 trace also reveals all the evaluations performed by the optimizer and how it arrived at the best join order and the best access path to use in executing a query.

You can set either Level 1 or Level 2 for the event 10053 trace. Level 2 captures the following types of information:

- Column statistics
- Single access paths
- Table joins considered by the optimizer
- Join costs
- Join methods considered by the optimizer

A Level 1 trace includes all the foregoing, plus a listing of all the default initialization parameters used by the optimizer. You'll also find detailed index statistics used by the optimizer in determining the best execution plan. The trace file captures the amazing array of statistics considered by the cost optimizer, and explains how the CBO creates the execution plan. Here are some of the important things you'll find in the CBO trace file.

- List of all internal optimizer-related initialization parameters
- Peeked values of the binds in the SQL statement
- Final query after optimizer transformations
- System statistics (`CPUSPEEDNW`, `IOTFRSPEED`, `IOSEEKTIM`, `MBRC`)
- Access path analysis for all objects in the query
- Join order evaluation

Unlike a raw 10046 event trace file, a 10053 event trace file is quite easy (and interesting) to read. Here are key excerpts from our trace file. The trace file shows the cost-based query transformations applied by the optimizer:

```
OBYE:   Considering Order-by Elimination from view SEL$1 (#0)
OBYE:     OBYE performed.
```

In this case, the optimizer eliminated the `order by` clause in our SQL statement. After performing all its transformations, the optimizer arrives at the "final query after transformations," which is shown here:

```
select channel_id,count(*)
from sh.sales
group by channel_id
```

Next, the output file shows the access path analysis for each of the tables in your query.

```
Access path analysis for SALES
***************************************
SINGLE TABLE ACCESS PATH
  Single Table Cardinality Estimation for SALES[SALES]
  Table: SALES  Alias: SALES
    Card: Original: 918843.000000  Rounded: 918843  Computed: 918843.00  Non Adjusted:
918843.00
  Access Path: TableScan
    Cost:  495.47  Resp: 495.47  Degree: 0
      Cost_io: 481.00  Cost_cpu: 205554857
      Resp_io: 481.00  Resp_cpu: 205554857
  Access Path: index (index (FFS))
    Index: SALES_CHANNEL_BIX
    resc_io: 42.30  resc_cpu: 312277
    ix_sel: 0.000000  ix_sel_with_filters: 1.000000
  Access Path: index (FFS)
    Cost:  42.32  Resp: 42.32  Degree: 1
      Cost_io: 42.30  Cost_cpu: 312277
      Resp_io: 42.30  Resp_cpu: 312277
  ****** trying bitmap/domain indexes ******
  Access Path: index (FullScan)
```

```
    Index: SALES_CHANNEL_BIX
    resc_io: 75.00  resc_cpu: 552508
    ix_sel: 1.000000  ix_sel_with_filters: 1.000000
    Cost: 75.04  Resp: 75.04  Degree: 0
  Access Path: index (FullScan)
    Index: SALES_CHANNEL_BIX
    resc_io: 75.00  resc_cpu: 552508
    ix_sel: 1.000000  ix_sel_with_filters: 1.000000
    Cost: 75.04  Resp: 75.04  Degree: 0
  Bitmap nodes:
    Used SALES_CHANNEL_BIX
      Cost = 75.038890, sel = 1.000000
  Access path: Bitmap index - accepted
    Cost: 75.038890 Cost_io: 75.000000 Cost_cpu: 552508.000000 Sel: 1.000000
    Believed to be index-only
  ****** finished trying bitmap/domain indexes ******
******** Begin index join costing ********
******** End index join costing ********
  Best:: AccessPath: IndexFFS
  Index: SALES_CHANNEL_BIX
         Cost: 42.32  Degree: 1  Resp: 42.32  Card: 918843.00  Bytes: 0
```

In this case, the optimizer evaluates various access paths and shows the optimal access path as an Index Fast Full Scan (IndexFFS).

The optimizer then considers various permutations of join orders and estimates the cost for each join order it considers:

```
Considering cardinality-based initial join order.
Join order[1]:  SALES[SALES]#0
GROUP BY sort
GROUP BY adjustment factor: 1.000000
      Total IO sort cost: 0      Total CPU sort cost: 834280255
      Best so far:  Table#: 0  cost: 101.0459  card: 918843.0000  bytes: 2756529
Number of join permutations tried: 1
GROUP BY adjustment factor: 1.000000
GROUP BY cardinality:  4.000000, TABLE cardinality:  918843.000000
     Total IO sort cost: 0      Total CPU sort cost: 834280255
  Best join order: 1
  Cost: 101.0459  Degree: 1  Card: 918843.0000  Bytes: 2756529
```

As our brief review of the 10053 trace output shows, you can get answers to puzzling questions such as why exactly the optimizer chose a certain join order or an access path, and why it ignored an index. The answers are all there!

10-24. Generating Automatic Oracle Error Traces

Problem

You want to create an automatic error dump file when a specific Oracle error occurs.

Solution

You can create error dumps to diagnose various problems in the database, by specifying the error number in a `hanganalyze` or `systemstate` command. For example, diagnosing the causes for deadlocks is often tricky. You can ask the database to dump a trace file when it hits the `ORA-00060: Deadlock detected` error. To do this, specify the event number 60 with the `hanganalyze` or the `systemstate` command:

```
SQL> alter session set events '60 trace name hanganalyze level 4';

Session altered.

SQL> alter session set events '60 trace name systemstate level 266';

Session altered.

SQL>
```

Both of these commands will trigger the automatic dumping of diagnostic data when the database next encounters the `ORA-00060` error. You can use the same technique in an Oracle RAC database. For example, you can issue the following command to generate automatic `hanganalyze` dumps:

```
SQL>alter session set events '60 trace name hanganalyze_global level 4';
```

This `alter session` statement invokes the `hanganalyze` command in any instance in which the database encounters the `ORA-00060` error.

How It Works

Setting event numbers for an error will ensure that when the specified error occurs the next time, Oracle automatically dumps the error information for you. This comes in very handy when you're diagnosing an error that occurs occasionally and getting a current `systemstate` dump or a `hanganalyze` dump is unhelpful. Some events such as deadlocks have a text alias, in which case you can specify the alias instead of the error number. For the `ORA-00060` error, the text alias is `deadlock`, and so you can issue the following command for tracing the error:

```
SQL> alter session set events 'deadlock trace name systemstate level 266';

Session altered.

SQL>
```

Similarly, you can use text aliases wherever they're available, for other error events.

10-25. Tracing a Background Process

Problem

You want to trace a background process.

Solution

If you aren't sure of the correct name of the background process you want to trace, you can list the exact names of all background processes by issuing the following command:

```
SQL> select name,description from v$bgprocess;
```

Suppose you want to trace the `dbw0` process. Issue the following commands to start and stop the trace.

```
SQL> alter system set events 'sql_trace {process:pname=dbw0}';
System altered.
SQL> alter system set events 'sql_trace {process:pname=dbw0} off';
System altered.
SQL>
```

You can trace two background processes at the same time by specifying the pipe (|) character to separate the process names:

```
SQL> alter system set events 'sql_trace {process:pname=dbw0|dbw1}';
System altered.

SQL> alter system set events 'sql_trace {process:pname=dbw0|dbw1} off';
System altered.
SQL>
```

How It Works

The Oracle Database 11g release offers several improvements to the SQL tracing facility. One of them is the new capability that allows you to issue an `alter system set events` command to trace a process (or a set of processes) or a specific SQL statement (or a set of statements).This recipe shows how to trace a background process by specifying the process name.

10-26. Enabling Oracle Listener Tracing

Problem

You want to trace the Oracle listener for diagnostic purposes.

Solution

To generate a trace for the Oracle listener, add the following two lines in the `listener.ora` file, which is by default located in the `$ORACLE_HOME/network/admin` directory.

```
trace_level_listener=support
trace_timestamp_listener=true
```

You can optionally specify a file name for the listener trace by adding the line `trace_file_listener=<file_name>`. Reload the listener with the `lsnrctl reload` command, and then check the status of the listener with the `lsnrctl status` command. You should see the trace file listed in the output:

```
C:\>lsnrctl reload

The command completed successfully

C:\>lsnrctl status
…

Listener Parameter File   C:\app\ora\product\11.2.0\dbhome_1\network\admin\liste
ner.ora
Listener Log File         c:\app\ora\diag\tnslsnr\MIROPC61\listener\alert\log.xm
l
Listener Trace File       c:\app\ora\diag\tnslsnr\MIROPC61\listener\trace\ora_86
40_9960.trc
…
  Instance "orcl1", status READY, has 1 handler(s) for this service...
Service "orcl1XDB.miro.local" has 1 instance(s).
  Instance "orcl1", status READY, has 1 handler(s) for this service...
The command completed successfully

C:>
```

The output shows the listener trace file you've just configured.

■ **Note** This and the next recipe don't have anything to with SQL tracing, which is the focus of this chapter. However, we thought we'd add these two recipes because they're useful, and there's no better chapter to put them in!

How It Works

You can specify various levels for listener tracing. You can specify Level 4 (`user`) for user trace information and Level 10 (`admin`) for administrative trace information. Oracle Support may request Level 16 (`support`) for troubleshooting Oracle listener issues.

If you can't reload or restart the listener, you can configure the tracing dynamically by issuing the following commands:

```
C:>lsnrctl

LSNRCTL> set current_listener listener
Current Listener is listener
LSNRCTL> set trc_level 16
```

```
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC
listener parameter "trc_level" set to support
The command completed successfully
LSNRCTL>
```

You can turn off listener tracing by issuing the command `set trc_level off`, which is the default value for this parameter:

```
LSNRCTL> set trc_level off
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC1521)))
LISTENER parameter "trc_level" set to off
The command completed successfully
LSNRCTL>
```

You'll notice that the `lsnrctl status` command doesn't show the listener trace file any longer. Note that the trace file for the listener will not be in the `$ORACLE_HOME\rdbms\network\admin` directory. Rather, the database stores the trace file in the `diag` directory of the database diagnosability infrastructure (ADR), under the `tnslsnr` directory, as shown here:

```
Listener Trace File      c:\app\ora\diag\tnslsnr\myhost\listener\trace\ora_86
                            40_9960.trc
```

10-27. Setting Archive Tracing for Data Guard

Problem

You want to trace the creation and transmission of the archive logs in a Data Guard environment.

Solution

You can trace the archive logs on either the primary or the standby database by setting the `log_archive_trace` initialization parameter:

```
log_archive_trace=trace_level(integer)
```

For example, if you want to set tracing at Level 15 on the primary server, you'd set this parameter as follows:

```
SQL> alter system set log_archive_trace=15
```

By default the `log_archive_trace` parameter is set to zero, meaning archive log tracing is disabled.

How It Works

Although archive log tracing is disabled when you leave the `log_archive_trace` parameter at its default level of zero, the database will still record error conditions in the trace files and the alert log. When you set the `log_archive_trace` parameter to a non-zero value, Oracle writes the appropriate trace output generated by the archive log process to an audit trail. The audit trail is the same trace directory as that for the SQL trace files—the *trace* directory in the ADR (this is the same as the user dump directory specified by the `user_dump_dest` initialization parameter).

On the primary database, the `log_archive_trace` parameter controls the output of the `ARC`*`n`* (archiver), `FAL` (fetch archived log), and the `LGWR` (log writer) background processes. On the standby databases, it traces the work of the `ARCn`, `RFS` (remote file server), and the `FAL` processes.

You can specify any of 15 levels of archive log tracing. Here's what the important tracing levels mean:

Level 1: Tracks the archiving of log files

Level 2: Tracks archive log status by destination

Level 4: Tracks archive operational phase

Level 8: Tracks archive log destination activity

Level 128: Tracks LGWR redo shipping network activity

Level 4096: Tracks real-time apply activity

Level 8192: Tracks redo apply activity (media recovery or physical standby)

When you specify a higher level of tracing, the trace will include information from all lower tracing levels. For example, if you specify Level 15, the trace file will include trace information from Levels 1, 2, 4, and 8.

CHAPTER 11

Automated SQL Tuning

Prior to Oracle Database 11g, accurately identifying poorly performing SQL queries and recommending solutions was mainly the purview of veteran SQL tuners. Typically one had to know how to identify high-resource SQL statements and bottlenecks, generate and interpret execution plans, extract data from the dynamic performance views, understand wait events and statistics, and then collate this knowledge to produce good SQL queries. As you'll see in this chapter, the Oracle SQL tuning paradigm has shifted a bit.

With the advent of automated SQL tuning features, anybody from novice to expert can generate and recommend solutions for SQL performance problems. This opens the door for new ways to address problematic SQL. For example, imagine your boss coming to you each morning with tuning recommendations and asking what the plan is to implement enhancements. This is different.

The automated SQL tuning feature is not a panacea for SQL performance angst. If you are an expert SQL tuner, there's no need to fear your skills are obsolete or your job is lost. There will always be a need to verify recommendations and successfully implement solutions. A human is still required to review the automated SQL tuning output and confirm the worthiness of fixes.

Still, there's been a change in the way SQL performance problems can be identified and solutions can be recommended. Some old-school folks may disagree and argue that you can't allow just anybody to generate SQL tuning advice. Regardless, Oracle has made these automated tools accessible and usable by the general population (for a fee). Therefore you need to understand the underpinnings of these features and how to use them.

This chapter focuses on the following automated SQL tuning tools:

- Automatic SQL Tuning
- SQL tuning sets (STS)
- SQL Tuning Advisor
- Automatic Database Diagnostic Monitor (ADDM)

Starting with Oracle Database 11g, *Automatic SQL Tuning* is a preset background database job that by default runs every day. This task examines high resource-consuming statements in the Automatic Workload Repository (AWR). It then invokes the SQL Tuning Advisor and generates tuning advice (if any) for each statement analyzed. As part of automated SQL tuning, you can configure characteristics such as the automatic acceptance of some recommendations such as SQL profiles (see Chapter 12 for details on SQL profiles).

A *SQL tuning set* (STS) is a database object that contains one or more SQL statements *and* the associated execution statistics. You can populate a SQL tuning set from multiple sources, such as SQL recorded in the AWR and SQL in memory, or you can provide specific SQL statements. It's critical that you be familiar with SQL tuning sets. This feature is used as an input to several of Oracle's performance tuning and management tools, such as the SQL Tuning Advisor, SQL Plan Management, SQL Access Advisor, and SQL Performance Advisor.

The *SQL Tuning Advisor* is central to Oracle's Automatic SQL Tuning feature. This tool runs automatically on a periodic basis and generates tuning advice for high resource-consuming SQL statements found in the AWR. You can also run the SQL Tuning Advisor manually and provide as input specific snapshot periods in the AWR, high resource-consuming SQL in memory, or user-provided SQL statements. This tool can be invoked via the `DBMS_SQLTUNE` package, SQL Developer, or Enterprise Manager.

The *Automatic Database Diagnostic Monitor* (ADDM) analyzes information in the AWR and provides recommendations on database performance issues including high resource-consuming SQL statements. The main goal of ADDM is to help you reduce the overall time (the DB time metric) spent by the database processing user requests. This tool can be invoked from an Oracle-provided SQL script, the `DBMS_ADDM` package, or Enterprise Manager.

All of the prior listed tools require an extra license from Oracle. You may not have a license to run these tools. Even if you don't have one, we still recommend that you know how these tools function. For example, you might have a manager asking if these automated tools are worth the cost, or you might be working with a developer who is investigating the use of these tools in a test environment. As a SQL tuning guru, you need to be familiar with these tools, as you will sooner or later encounter these automated features.

Before investigating the recipes in this chapter, please take a long look at Figure 11-1. This diagram demonstrates how the various automated tools interact and in what scenarios you would use a particular feature. Refer back to this diagram as you work through the recipes in this chapter. Particularly notice that you can easily use SQL statements found in the AWR or SQL currently in memory as input for various Oracle tuning tools. This allows you to systematically identify and use high-resource SQL statements as the target for various performance tuning activities.

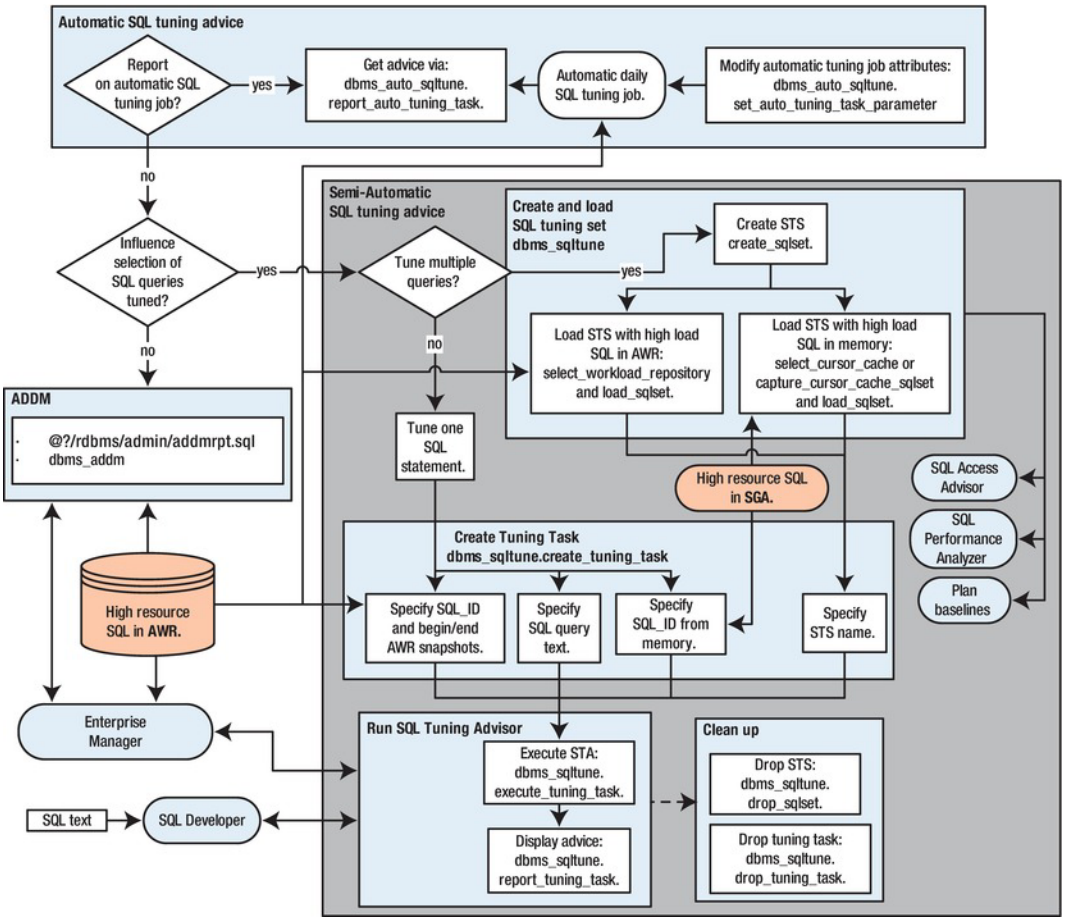

Figure 11-1. *Oracle's automatic SQL tuning tools*

The first several recipes in this chapter deal with the Automatic SQL Tuning feature. You'll be shown how to determine if and when the automated job is running and how to modify its characteristics. The middle section of this chapter focuses on how to create and manage SQL tuning sets. SQL tuning sets are used widely as input to various Oracle performance tuning tools. Lastly, the chapter shows you how to manually run the SQL Tuning Advisor and ADDM to generate performance recommendations for SQL statements.

■ **Note** In the examples in this chapter, we focus on showing you how to use features via SQL and built-in PL/SQL packages. While we do show some screenshots from Enterprise Manager, we don't focus on the graphical tool usage. You should be able to use the SQL and PL/SQL regardless of whether Enterprise Manager is installed. Furthermore, the manual approach allows you to understand each piece of the process and will help you to diagnose issues when problems arise.

11-1. Displaying Automatic SQL Tuning Job Details

Problem

You have an Oracle Database 11g environment and want to determine if the Automatic SQL Tuning job is enabled and regularly running. If the job is enabled, you want to display other aspects, such as when it starts and how long it executes.

Solution

Use the following query to determine if any Automatic SQL Tuning jobs are enabled:

```
SELECT client_name, status, consumer_group, window_group
FROM dba_autotask_client
ORDER BY client_name;
```

The following output shows that there are three enabled automatic jobs running, one of which is the SQL Tuning Advisor:

```
CLIENT_NAME                    STATUS   CONSUMER_GROUP            WINDOW_GROUP
------------------------------ -------- ------------------------- --------------------
sql tuning advisor             ENABLED  ORA$AUTOTASK_SQL_GROUP    ORA$AT_WGRP_SQ
auto space advisor             ENABLED  ORA$AUTOTASK_SPACE_GROUP  ORA$AT_WGRP_SA
auto optimizer stats collection ENABLED  ORA$AUTOTASK_STATS_GROUP  ORA$AT_WGRP_OS
```

Run the following query to view the last several times the Automatic SQL Tuning Advisor job has run:

```
SELECT task_name, status, TO_CHAR(execution_end,'DD-MON-YY HH24:MI')
FROM dba_advisor_executions
WHERE task_name='SYS_AUTO_SQL_TUNING_TASK'
ORDER BY execution_end;
```

Here is some sample output:

```
TASK_NAME                      STATUS     TO_CHAR(EXECUTION_END
------------------------------ ---------- --------------------
SYS_AUTO_SQL_TUNING_TASK       COMPLETED   30-APR-11 06:00
SYS_AUTO_SQL_TUNING_TASK       COMPLETED   01-MAY-11 06:02
```

How It Works

When you create a database in Oracle Database 11g or higher, Oracle automatically implements three automatic maintenance jobs:

- Automatic SQL Tuning Advisor
- Automatic Segment Advisor
- Automatic Optimizer Statistics Collection

These tasks are automatically configured to run in maintenance windows. A maintenance window is a specified time and duration for the task to run. You can view the maintenance window details with this query:

```
SELECT window_name,TO_CHAR(window_next_time,'DD-MON-YY HH24:MI:SS')
,sql_tune_advisor, optimizer_stats, segment_advisor
FROM dba_autotask_window_clients;
```

Here's a snippet of the output for this example:

```
WINDOW_NAME      TO_CHAR(WINDOW_NEXT_TIME SQL_TUNE OPTIMIZE SEGMENT_
---------------- ------------------------ -------- -------- --------
THURSDAY_WINDOW  28-APR-11 22:00:00       ENABLED  ENABLED  ENABLED
FRIDAY_WINDOW    29-APR-11 22:00:00       ENABLED  ENABLED  ENABLED
SATURDAY_WINDOW  30-APR-11 06:00:00       ENABLED  ENABLED  ENABLED
SUNDAY_WINDOW    01-MAY-11 06:00:00       ENABLED  ENABLED  ENABLED
```

There are several data dictionary views related to the automatically scheduled jobs. See Table 11-1 for descriptions of these views.

Table 11-1. *Automatic Maintenance Task View Descriptions*

View Name	Description
DBA_AUTOTASK_CLIENT	Statistical information about automatic jobs
DBA_AUTOTASK_CLIENT_HISTORY	Window history of job execution
DBA_AUTOTASK_CLIENT_JOB	Currently running automatic scheduled jobs
DBA_AUTOTASK_JOB_HISTORY	History of automatic scheduled job runs
DBA_AUTOTASK_SCHEDULE	Schedule of automated tasks for next 32 days
DBA_AUTOTASK_TASK	Information regarding current and past tasks
DBA_AUTOTASK_OPERATION	Operations for automated tasks
DBA_AUTOTASK_WINDOW_CLIENTS	Displays windows that belong to the MAINTENANCE_WINDOW_GROUP

11-2. Displaying Automatic SQL Tuning Advice

Problem

You're aware that Oracle automatically runs a daily job that generates SQL tuning advice. You want to view the advice.

Solution

If you're using Oracle Database 11g Release 2 or higher, here's the quickest way to display automatically generated SQL tuning advice:

```
SQL> SET LINESIZE 80 PAGESIZE 0 LONG 100000
SQL> SELECT DBMS_AUTO_SQLTUNE.REPORT_AUTO_TUNING_TASK FROM DUAL;
```

■ **Note** Starting with Oracle Database 11g Release 2, the `DBMS_AUTO_SQLTUNE` package should be used (instead of `DBMS_SQLTUNE`) for administrating automatic SQL tuning features. If you are using an older release of Oracle, use `DBMS_SQLTUNE.REPORT_AUTO_TUNING_TASK` to view automated SQL tuning advice.

Depending on the activity in your database, there may be a great deal of output. Here's a small sample of output from a very active database:

```
GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------
Tuning Task Name                  : SYS_AUTO_SQL_TUNING_TASK
Tuning Task Owner                 : SYS
Workload Type                     : Automatic High-Load SQL Workload
Execution Count                   : 30
Current Execution                 : EXEC_3483
Execution Type                    : TUNE SQL
Scope                             : COMPREHENSIVE
.....
Completion Status                 : COMPLETED
Started at                        : 04/10/2011 06:00:01
Completed at                      : 04/10/2011 06:02:41
Number of Candidate SQLs          : 103
Cumulative Elapsed Time of SQL (s)      : 49124
-------------------------------------------------------------------------------
```

```
SUMMARY SECTION
--------------------------------------------------------------------------------
                     Global SQL Tuning Result Statistics
--------------------------------------------------------------------------------
Number of SQLs Analyzed                      : 103
Number of SQLs in the Report                 : 8
Number of SQLs with Findings                 : 8
Number of SQLs with Alternative Plan Findings: 1
Number of SQLs with SQL profiles recommended : 1
--------------------------------------------------------------------------------
    SQLs with Findings Ordered by Maximum (Profile/Index) Benefit, Object ID
--------------------------------------------------------------------------------
object ID  SQL ID        statistics profile(benefit) index(benefit) restructure
---------- ------------- ---------- ---------------- -------------- -----------
      9130 crx9h7tmwwv67                     51.44%
```

AUTOMATICALLY E-MAILING SQL OUTPUT

On Linux/Unix systems, it's quite easy to automate the e-mailing of output from a SQL script. First encapsulate the SQL in a shell script, and then use a utility such as `cron` to automatically generate and e-mail the output. Here's a sample shell script:

```
#!/bin/bash
if [ $# -ne 1 ]; then
  echo "Usage: $0 SID"
  exit 1
fi
# source oracle OS variables
. /var/opt/oracle/oraset $1
#
BOX=`uname -a | awk '{print$2}'`
OUTFILE=$HOME/bin/log/sqladvice.txt
#
sqlplus -s <<EOF
mv_maint/foo
SPO $OUTFILE
SET LINESIZE 80 PAGESIZE 0 LONG 100000
SELECT DBMS_AUTO_SQLTUNE.REPORT_AUTO_TUNING_TASK FROM DUAL;
EOF
cat $OUTFILE | mailx -s "SQL Advice: $1 $BOX" larry@oracle.com
exit 0
```

Here's the corresponding `cron` entry that runs the report on a daily basis:

```
#-------------------------------------------------------------------
# SQL Advice report from SQL auto tuning
16 11 * * * /orahome/oracle/bin/sqladvice.bsh DWREP
   1>/orahome/oracle/bin/log/sqladvice.log 2>&1
#-------------------------------------------------------------------
```

In the prior `cron` entry, the command was broken into two lines to fit on a page within this book.

How It Works

The "Solution" section describes a simple method to display in-depth tuning advice for high-load queries in your database. Depending on the activity and load on your database, the report may contain no suggestions or may provide a great deal of advice. The Automatic SQL Tuning job uses the high-workload SQL statements identified in the AWR as the target SQL statements to report on. The advice report consists of one or more of the following general subsections:

- General information
- Summary
- Details
- Findings
- Explain plans
- Alternate plans
- Errors

The general information section contains high-level information regarding the start and end time, number of SQL statements considered, cumulative elapsed time of the SQL statements, and so on.

The summary section contains information regarding the SQL statements analyzed—for example:

```
                    Global SQL Tuning Result Statistics
-------------------------------------------------------------------------------
Number of SQLs Analyzed                      : 26
Number of SQLs in the Report                 : 5
Number of SQLs with Findings                 : 5
Number of SQLs with Alternative Plan Findings: 1
Number of SQLs with SQL profiles recommended : 5
Number of SQLs with Index Findings           : 2
-------------------------------------------------------------------------------
    SQLs with Findings Ordered by Maximum (Profile/Index) Benefit, Object ID
-------------------------------------------------------------------------------
object ID  SQL ID        statistics profile(benefit) index(benefit) restructure
---------- ------------- ---------- ---------------- -------------- -----------
      1160 31q9w59vpt86t                      98.27%         99.90%
      1167 3u8xd0vf2pnhr                      98.64%
```

The detail section contains information describing specific SQL statements, such as the owner and SQL text. Here is a small sample:

```
DETAILS SECTION
-------------------------------------------------------------------------------
 Statements with Results Ordered by Maximum (Profile/Index) Benefit, Object ID
-------------------------------------------------------------------------------
Object ID  : 1160
Schema Name: CHN_READ
SQL ID     : 31q9w59vpt86t
SQL Text   : SELECT "A2"."UMID","A2"."ORACLE_UNIQUE_ID","A2"."PUBLIC_KEY","A2"
             ."SERIAL_NUMBER",:1||"A1"."USER_NAME","A1"."USER_NAME",NVL("A2"."
             CREATE_TIME_DTT",:2),NVL("A2"."UPDATE_TIME_DTT",:3) FROM
             "COMPUTER_SYSTEM" "A2","USERS" "A1" WHERE
```

The findings section contains recommendations such as accepting a SQL profile or creating an index—for example:

```
FINDINGS SECTION
-------------------------------------------------------------------------------
1- SQL Profile Finding (see explain plans section below)
--------------------------------------------------------
  A potentially better execution plan was found for this statement.
  Recommendation (estimated benefit: 98.27%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile to use
parallel execution for this statement.
    execute dbms_sqltune.accept_sql_profile(task_name =>
            'SYS_AUTO_SQL_TUNING_TASK', object_id => 1160, task_owner =>
            'SYS', replace => TRUE, profile_type => DBMS_SQLTUNE.PX_PROFILE);
.................
2- Index Finding (see explain plans section below)
--------------------------------------------------
The execution plan of this statement can be improved by creating
one or more indices.
  Recommendation (estimated benefit: 99.9%)
  -----------------------------------------
  - Consider running the Access Advisor to improve the physical schema design
    or creating the recommended index.
    create index CHAINSAW.IDX$$_90890002 on
    CHAINSAW.COMPUTER_SYSTEM("UPDATE_TIME_DTT");
```

Where appropriate, the original execution plan for a query is displayed along with a suggested fix and new execution plan. This allows you to see the before and after plan differences. This is very useful when determining if the findings (such as adding an index) would improve performance.

Lastly, there is an error section of the report. For most scenarios, there typically will not be an error section in the report.

The "Solution" section showed how to execute the `REPORT_AUTO_TUNING_TASK` function from a SQL statement. This function can also be called from an anonymous block of PL/SQL. Here's an example:

```
VARIABLE tune_report CLOB;
BEGIN
  :tune_report := DBMS_AUTO_SQLTUNE.report_auto_tuning_task(
    begin_exec   => NULL
   ,end_exec     => NULL
   ,type         => DBMS_AUTO_SQLTUNE.type_text
```

```
   ,level        => DBMS_AUTO_SQLTUNE.level_typical
   ,section      => DBMS_AUTO_SQLTUNE.section_all
   ,object_id    => NULL
   ,result_limit => NULL);
END;
/
--
SET LONG 1000000
PRINT :tune_report
```

The parameters for the REPORT_AUTO_TUNING_TASK function are described in detail in Table 11-2. These parameters allow you a great deal of flexibility to customize the advice output.

Table 11-2. *Parameter Details for the REPORT_AUTO_TUNING_TASK Function*

Parameter Name	Description
BEGIN_EXEC	Name of beginning task execution; NULL means the most recent task is used.
END_EXEC	Name of ending task; NULL means the most recent task is used.
TYPE	Type of report to produce; TEXT specifies a text report.
LEVEL	Level of detail; valid values are BASIC, TYPICAL, and ALL.
SECTION	Section of the report to include; valid values are ALL, SUMMARY, FINDINGS, PLAN, INFORMATION, and ERROR.
OBJECT_ID	Used to report on a specific statement; NULL means all statements.
RESULT_LIMIT	Maximum number of SQL statements to include in report

11-3. Generating a SQL Script to Implement Automatic Tuning Advice

Problem

You've reported on the automatic tuning advice. Now you want to generate a SQL script that can be used to implement tuning advice.

Solution

Use the DBMS_SQLTUNE.SCRIPT_TUNING_TASK function to generate the SQL statements to implement the advice of a tuning task. You need to provide as input the name of the automatic tuning task. In this example, the name of the task is SYS_AUTO_SQL_TUNING_TASK:

```
SET LINES 132 PAGESIZE 0 LONG 10000
SELECT DBMS_SQLTUNE.SCRIPT_TUNING_TASK('SYS_AUTO_SQL_TUNING_TASK') FROM dual;
```

Here is a small snippet of the output for this example:

```
execute dbms_stats.gather_index_stats(ownname => 'STAR2', indname => 'F_CONFIG_P
ROD_INST_FK1', estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE);
create index NSESTAR.IDX$$_17F5F0004 on NSESTAR.D_DATES("FISCAL_YEAR","FISCAL_WE
EK_NUMBER_IN_YEAR","DATE_DTT");
```

How It Works

The `SCRIPT_TUNING_TASK` function generates the SQL to implement the advice recommended by the Automatic SQL Tuning job. If the tuning task doesn't have any advice to give, then there won't be any SQL statements generated in the output. `SYS_AUTO_SQL_TUNING_TASK` is the default name of the Automatic SQL Tuning task. If you're unsure of the details regarding this task, then query the `DBA_ADVISOR_LOG` view:

```
select task_name, execution_start from dba_advisor_log
where task_name='SYS_AUTO_SQL_TUNING_TASK'
order by 2;
```

Here's some sample output for this example:

```
TASK_NAME                      EXECUTION
------------------------------ ---------
SYS_AUTO_SQL_TUNING_TASK       19-APR-11
```

11-4. Modifying Automatic SQL Tuning Features

Problem

You've noticed that sometimes the Automatic SQL Tuning advice job recommends that a SQL profile be applied to a SQL statement (see Chapter 12 for details on SQL profiles). The default behavior of the tuning advice job is to not automatically accept SQL profile recommendations. You want to modify this behavior and have the Automatic SQL Tuning job automatically place any SQL profiles that it recommends into an accepted state.

Solution

Use the `DBMS_AUTO_SQLTUNE.SET_AUTO_TUNING_TASK_PARAMETER` procedure to modify the default behavior of Automatic SQL Tuning. For example, if you want SQL profiles to be automatically accepted, you can do so as follows:

```
BEGIN
  DBMS_AUTO_SQLTUNE.SET_AUTO_TUNING_TASK_PARAMETER(
    parameter => 'ACCEPT_SQL_PROFILES', value => 'TRUE');
END;
/
```

You can verify that auto SQL profile accepting is enabled via this query:

```
SELECT parameter_name, parameter_value
FROM dba_advisor_parameters
WHERE task_name = 'SYS_AUTO_SQL_TUNING_TASK'
AND  parameter_name ='ACCEPT_SQL_PROFILES';
```

Here is some sample output:

```
PARAMETER_NAME                 PARAMETER_VALUE
------------------------------ ------------------------------
ACCEPT_SQL_PROFILES            TRUE
```

To disable automatic acceptance of SQL profiles, pass a `FALSE` value to the procedure:

```
BEGIN
  DBMS_AUTO_SQLTUNE.SET_AUTO_TUNING_TASK_PARAMETER(
    parameter => 'ACCEPT_SQL_PROFILES', value => 'FALSE');
END;
/
```

■ **Note** Starting with Oracle Database 11g Release 2, the `DBMS_AUTO_SQLTUNE` package should be used (instead of `DBMS_SQLTUNE`) for administrating Automatic SQL Tuning features.

How It Works

The `DBMS_AUTO_SQLTUNE.SET_AUTO_TUNING_TASK_PARAMETER` procedure allows you to modify the default behavior of the Automatic SQL Tuning job. You can view all of the current settings for Automatic SQL Tuning via this query:

```
SELECT parameter_name ,parameter_value
FROM dba_advisor_parameters
WHERE task_name = 'SYS_AUTO_SQL_TUNING_TASK'
AND  parameter_name IN ('ACCEPT_SQL_PROFILES',
                        'MAX_SQL_PROFILES_PER_EXEC',
                        'MAX_AUTO_SQL_PROFILES',
                        'EXECUTION_DAYS_TO_EXPIRE');
```

Here's some sample output:

```
PARAMETER_NAME                 PARAMETER_VALUE
------------------------------ ------------------------------
ACCEPT_SQL_PROFILES            FALSE
EXECUTION_DAYS_TO_EXPIRE       30
MAX_SQL_PROFILES_PER_EXEC      20
MAX_AUTO_SQL_PROFILES          10000
```

The prior parameters are described in Table 11-3.

Table 11-3. *Description of `SET_AUTO_TUNING_TASK_PARAMETER` Parameters*

Parameter Name	Description
ACCEPT_SQL_PROFILE	Determines if SQL profiles are automatically accepted
EXECUTION_DAYS_TO_EXPIRE	Number of days to save task history
MAX_SQL_PROFILES_PER_EXEC	Limit of SQL profiles accepted per execution of tuning task
MAX_AUTO_SQL_PROFILES	Maximum limit of SQL profiles automatically accepted

You can also use Enterprise Manager to manage the features regarding Automatic SQL Tuning. From the main database page, navigate to the Advisor Central page. Next, click the SQL Advisors link. Now click the Automatic SQL Tuning Results page. You should be presented with a screen similar to Figure 11-2.

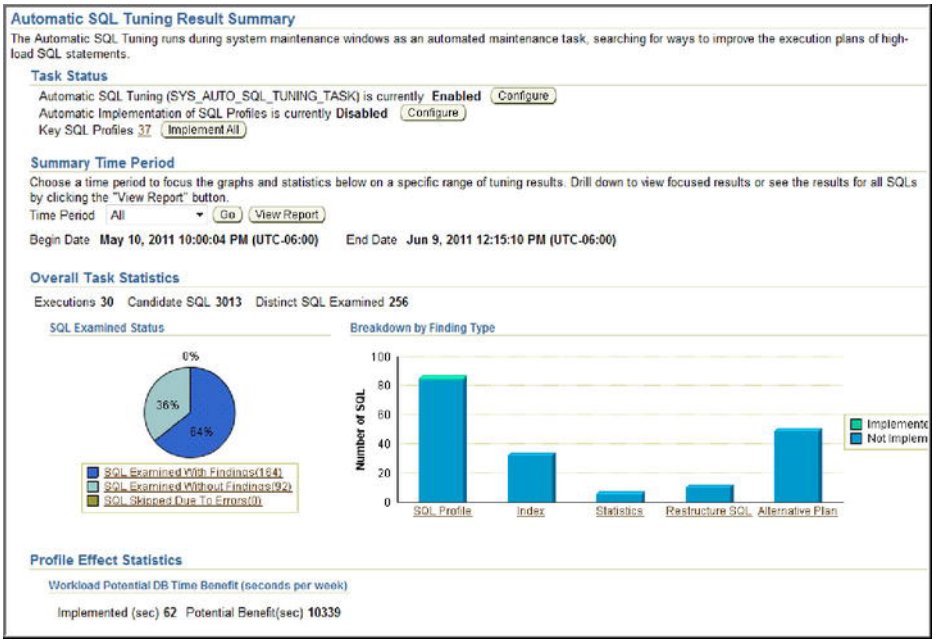

Figure 11-2. *Managing Automatic SQL Tuning with Enterprise Manager*

From this screen, you can configure, view results, disable, and enable various aspects of Automatic SQL Tuning.

11-5. Disabling and Enabling Automatic SQL Tuning

Problem

You want to completely disable and later re-enable the Automatic SQL Tuning job.

Solution

Use the DBMS_AUTO_TASK_ADMIN.DISABLE procedure to disable the Automatic SQL Tuning job. This example disables the Automatic SQL Tuning Advisor job.

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.DISABLE(
  client_name => 'sql tuning advisor',
  operation => NULL,
  window_name => NULL);
END;
/
```

To re-enable the job, use the ENABLE procedure as shown:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.ENABLE(
  client_name => 'sql tuning advisor',
  operation => NULL,
  window_name => NULL);
END;
/
```

You can report on the status of the automatic tuning job via this query:

```
SELECT client_name ,status ,consumer_group
FROM dba_autotask_client
ORDER BY client_name;
```

Here's some sample output:

```
CLIENT_NAME                     STATUS          CONSUMER_GROUP
------------------------------- --------------- ------------------------------
auto optimizer stats collection  ENABLED         ORA$AUTOTASK_STATS_GROUP
auto space advisor               ENABLED         ORA$AUTOTASK_SPACE_GROUP
sql tuning advisor               ENABLED         ORA$AUTOTASK_SQL_GROUP
```

How It Works

You might want to disable the Automatic SQL Tuning job because you have a very active database and want to ensure that this job doesn't impact the overall performance of the database. The DBMS_AUTO_TASK_ADMIN.ENABLE/DISABLE procedures allow you to turn on and off the Automatic SQL Tuning job. These procedures take three parameters (see Table 11-4 for details). The behavior of the procedures varies depending on which parameters you pass in:

- If CLIENT_NAME is provided and both OPERATION and WINDOW_NAME are NULL, then the client is disabled.
- If OPERATION is provided, then the operation is disabled.
- If WINDOW_NAME is provided, and OPERATION is NULL, then the client is disabled in the provided window name.

The prior parameters allow you to control at a granular detail the schedule of the automatic task. Given the prior rules, you would disable the Automatic SQL Tuning job during the Tuesday maintenance window as follows:

```
BEGIN
  dbms_auto_task_admin.disable(
  client_name => 'sql tuning advisor',
  operation => NULL,
  window_name => 'TUESDAY_WINDOW');
END;
/
```

You can verify that the window has been disabled via this query:

```
SELECT window_name,TO_CHAR(window_next_time,'DD-MON-YY HH24:MI:SS')
,sql_tune_advisor
FROM dba_autotask_window_clients;
```

Here is a snippet of the output:

```
WINDOW_NAME      TO_CHAR(WINDOW_NEXT_TIME SQL_TUNE
---------------- ------------------------ --------
TUESDAY_WINDOW   03-MAY-11 22:00:00       DISABLED
```

Table 11-4. *Parameter Descriptions for* `DBMS_AUTO_TASK_ADMIN.ENABLE` *and* `DISABLE` *Procedures*

Parameter	Description
CLIENT_NAME	Name of client; query DBA_AUTOTASK_CLIENT for details.
OPERATION	Name of operation; query DBA_AUTOTASK_OPERATION for details.
WINDOW_NAME	Operation name of the window

11-6. Modifying Maintenance Window Attributes

Problem

You realize that the automatic tasks (such as the Automatic SQL Tuning job) run during regularly scheduled maintenance windows. You want to modify the length of time associated with a maintenance window.

Solution

Here's an example that changes the duration of the Sunday maintenance window to two hours:

```
BEGIN
  dbms_scheduler.set_attribute(
  name => 'SUNDAY_WINDOW',
  attribute => 'DURATION',
  value => numtodsinterval(2, 'hour'));
END;
/
```

You can confirm the changes to the maintenance window with this query:

```
SELECT window_name, next_start_date, duration
FROM dba_scheduler_windows;
```

Here is a snippet of the output:

```
WINDOW_NAME      NEXT_START_DATE                         DURATION
---------------- --------------------------------------- --------------------
SATURDAY_WINDOW  07-MAY-11 06.00.00.000000 AM US/MOUNTAIN +000 20:00:00
SUNDAY_WINDOW    08-MAY-11 06.00.00.000000 AM US/MOUNTAIN +000 02:00:00
```

How It Works

The key to understanding how to modify a maintenance window is that it is an attribute of the database job scheduler and therefore must be maintained via the `DBMS_SCHEDULER` package. When you install Oracle Database 11g, by default three automatic maintenance jobs are configured:

- Automatic SQL Tuning
- Statistics gathering
- Segment advice

These jobs automatically execute in preconfigured daily maintenance windows. A maintenance window consists of a day of the week and the length of time the job runs.

You can view the future one month's worth of scheduled jobs via this query:

```
SELECT window_name, to_char(start_time,'dd-mon-yy hh24:mi'), duration
FROM dba_autotask_schedule
ORDER BY start_time;
```

Here is a small sample of the output:

```
WINDOW_NAME          TO_CHAR(START_TIME,'D DURATION
-------------------- -------------------- --------------------
SATURDAY_WINDOW      14-may-11 06:00      +000 20:00:00
SUNDAY_WINDOW        15-may-11 06:00      +000 02:00:00
```

■ **Tip** See Oracle's Database Administrator's Guide (available on the Oracle Technology Network web site) for further details on managing scheduled jobs.

11-7. Creating a SQL Tuning Set Object

Problem

You're working on a performance issue that requires that you analyze a group of SQL statements. Before you process the SQL statements as a set, you need to create a SQL tuning set object.

Solution

Use the `DBMS_SQLTUNE.CREATE_SQLSET` procedure to create a SQL tuning set object—for example:

```
BEGIN
  DBMS_SQLTUNE.CREATE_SQLSET(
  sqlset_name => 'HIGH_IO',
  description => 'High disk read tuning set');
END;
/
```

The prior code creates a tuning set with the name of `HIGH_IO`. At this point, you have created a named tuning set object. The tuning set does not contain any SQL statements.

How It Works

A SQL tuning set object must be created before populating a tuning set with SQL statements (see Recipes 11-9 through 11-11 for details on adding SQL statements to an STS). You can view any defined SQL tuning sets in the database by querying the `DBA_SQLSET` view:

```
SQL> select id, name, created, statement_count from dba_sqlset;
```

Here is some sample output:

```
        ID NAME                           CREATED   STATEMENT_COUNT
---------- ------------------------------ --------- ---------------
         5 HIGH_IO                        26-APR-11               0
```

If you need to drop a SQL tuning set object, then use the `DBMS_SQLTUNE.DROP_SQLSET` procedure to drop a tuning set. The following example drops a tuning set named `MY_TUNING_SET`:

```
SQL> EXEC  DBMS_SQLTUNE.DROP_SQLSET(sqlset_name => 'MY_TUNING_SET' );
```

11-8. Viewing Resource-Intensive SQL in the AWR

Problem

Before populating a SQL tuning set, you want to view high-load SQL statements in the AWR. You want to eventually use SQL contained in the AWR as input for populating a SQL tuning set.

Solution

The DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY function can be used to extract SQL stored in the AWR. This particular query selects queries in the AWR between snapshots 8200 and 8201 ordered by the top 10 in the disk reads usage category:

```
SELECT
 sql_id
,substr(sql_text,1,20)
,disk_reads
,cpu_time
,elapsed_time
FROM table(DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY(8200,8201,
            null, null, 'disk_reads',null, null, null, 10))
ORDER BY disk_reads DESC;
```

Here is a small snippet of the output:

```
SQL_ID         SUBSTR(SQL_TEXT,1,20 DISK_READS      CPU_TIME  ELAPSED_TIME
-------------- -------------------- ---------- ------------- -------------
achffburdff9j   delete from "MVS"."     101145     814310000     991574249
5vku5ap6g6zh8  INSERT /*+ BYPASS_RE      98172      75350000      91527239
```

How It Works

Before you work with SQL tuning sets, it's critical to understand you can use the DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY function to retrieve high resource-usage SQL from the AWR. The result sets retrieved by this PL/SQL function can be used as input for populating SQL tuning sets. See Table 11-5 for a description of the SELECT_WORKLOAD_REPOSITORY function parameters.

You have a great deal of flexibility in how you use this function. A few examples will help illustrate this. Say you want to retrieve SQL from the AWR that was not parsed by the SYS user. Here is the SQL to do that:

```
SELECT sql_id, substr(sql_text,1,20)
,disk_reads, cpu_time, elapsed_time, parsing_schema_name
FROM table(
DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY(8200,8201,
'parsing_schema_name <> ''SYS''',
NULL, NULL,NULL,NULL, 1, NULL, 'ALL'));
```

The following example retrieves the top ten queries ranked by buffer gets for non-SYS users:

```
SELECT
```

```
 sql_id
,substr(sql_text,1,20)
,disk_reads
,cpu_time
,elapsed_time
,buffer_gets
,parsing_schema_name
FROM table(
DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY(
 begin_snap => 21730
,end_snap => 22900
,basic_filter => 'parsing_schema_name <> ''SYS'''
,ranking_measure1 => 'buffer_gets'
,result_limit => 10
));
```

In the prior queries, the SYS keyword is enclosed by two single quotes (in other words, those aren't double quotes around SYS).

Table 11-5. *Parameter Descriptions of the SELECT_WORKLOAD_REPOSITORY Function*

Parameter	Description
BEGIN_SNAP	Non-inclusive beginning snapshot ID
END_SNAP	Inclusive ending snapshot ID
BASELINE_NAME	Name of AWR baseline
BASIC_FILTER	SQL predicate to filter SQL statements from workload; if not set, then only SELECT, INSERT, UPDATE, DELETE, MERGE, and CREATE TABLE statements are captured.
OBJECT_FILTER	Not currently used
RANKING_MEASURE(n)	Order by clause on selected SQL statement(s), such as elapsed_time, cpu_time, buffer_gets, disk_reads, and so on; N can be 1, 2, or 3.
RESULT_PERCENTAGE	Filter for choosing top N% for ranking measure

Continued

Parameter	Description
RESULT_LIMIT	Limit of the number of SQL statements returned in the result set.
ATTRIBUTE_LIST	List of SQL statement attributes (TYPICAL, BASIC, ALL, and so on)
RECURSIVE_SQL	Include/exclude recursive SQL (HAS_RECURSIVE_SQL or NO_RECURSIVE_SQL)

11-9. Viewing Resource-Intensive SQL in Memory

Problem

Before populating a SQL tuning set, you want to view high-load SQL statements in the cursor cache in memory. You want to eventually use SQL contained in memory as input for populating a SQL tuning set.

Solution

Use the DBMS_SQLTUNE.SELECT_CURSOR_CACHE function to view current high resource-consuming SQL statements in memory. This query selects SQL statements in memory that have required more than a million disk reads:

```
SELECT
 sql_id
,substr(sql_text,1,20)
,disk_reads
,cpu_time
,elapsed_time
FROM table(DBMS_SQLTUNE.SELECT_CURSOR_CACHE('disk_reads > 1000000'))
ORDER BY sql_id;
```

Here is some sample output:

```
SQL_ID        SUBSTR(SQL_TEXT,1,20 DISK_READS   CPU_TIME ELAPSED_TIME
------------- -------------------- ---------- ---------- ------------
0s6gq1c890p4s  delete from "MVS"."    3325320 8756130000   1.0416E+10
b63h4skwvpshj BEGIN dbms_mview.ref    9496353 1.4864E+10   3.3006E+10
```

How It Works

Before you work with SQL tuning sets, it's critical to understand you can use the DBMS_SQLTUNE.SELECT_CURSOR_CACHE function to retrieve high resource-usage SQL from memory. The result set retrieved by this PL/SQL function can be used as input for populating SQL tuning sets. See Table 11-6 for a description of the SELECT_CURSOR_CACHE function parameters.

You have a great deal of flexibility in how you use this function. Here's an example that selects SQL in memory, but excludes statements parsed by the SYS user and also returns statements with an elapsed time greater than 100,000:

```
SELECT sql_id, substr(sql_text,1,20)
,disk_reads, cpu_time, elapsed_time
FROM table(DBMS_SQLTUNE.SELECT_CURSOR_CACHE('parsing_schema_name <> ''SYS''
                                             AND elapsed_time > 100000'))
ORDER BY sql_id;
```

In the prior query, the SYS keyword is enclosed by two single quotes (in other words, those aren't double quotes around SYS). The SQL_TEXT column is truncated to 20 characters so that the output can be displayed on the page more easily. Here is some sample output:

```
SQL_ID        SUBSTR(SQL_TEXT,1,20 DISK_READS   CPU_TIME ELAPSED_TIME
------------- -------------------- ---------- ---------- ------------
byzwu34haqmh4 SELECT /* DS_SVC */           0     140000       159828
```

Once you have identified a SQL_ID for a resource-intensive SQL statement, you can view all of its execution details via this query:

```
SELECT *
FROM table(DBMS_SQLTUNE.SELECT_CURSOR_CACHE('sql_id = ''byzwu34haqmh4'''));
```

Note that the SQL_ID in the prior statement is enclosed by two single quotes (not double quotes).

This next example selects the top ten queries in memory in terms of CPU time for non-SYS users:

```
SELECT
 sql_id
,substr(sql_text,1,20)
,disk_reads
,cpu_time
,elapsed_time
,buffer_gets
,parsing_schema_name
FROM table(
DBMS_SQLTUNE.SELECT_CURSOR_CACHE(
 basic_filter => 'parsing_schema_name <> ''SYS'''
,ranking_measure1 => 'cpu_time'
,result_limit => 10
));
```

Table 11-6. *Parameter Descriptions of the* `SELECT_CURSOR_CACHE` *Function*

Parameter	Description
`BASIC_FILTER`	SQL predicate to filter SQL in the cursor cache
`OBJECT_FILTER`	Currently not used
`RANKING_MEASURE(n)`	`ORDER BY` clause for the SQL returned
`RESULT_PERCENTAGE`	Filter for the top N percent queries for the ranking measure provided; invalid if more than one ranking measure provided
`RESULT_LIMIT`	Top number of SQL statements filter
`ATTRIBUTE_LIST`	List of SQL attributes to return in result set
`RECURSIVE_SQL`	Include recursive SQL

11-10. Populating SQL Tuning Set from High-Resource SQL in AWR

Problem

You want to create a SQL tuning set and populate it with the top I/O-consuming SQL statements found in the AWR.

Solution

Use the following steps to populate a SQL tuning set from high resource-consuming statements in the AWR:

1. Create a SQL tuning set object.
2. Determine begin and end AWR snapshot IDs.
3. Populate the SQL tuning set with high-resource SQL found in AWR.

The prior steps are detailed in the following subsections.

Step 1: Create a SQL Tuning Set Object

Create a SQL tuning set. This next bit of code creates a tuning set named `IO_STS`:

```
BEGIN
  dbms_sqltune.create_sqlset(
    sqlset_name => 'IO_STS'
   description => 'STS from AWR');
END;
/
```

Step 2: Determine Begin and End AWR Snapshot IDs

If you're unsure of the available snapshots in your database, you can run an AWR report or select the SNAP_ID from DBA_HIST_SNAPSHOTS:

```
select snap_id, begin_interval_time
from dba_hist_snapshot order by 1;
```

Step 3: Populate the SQL Tuning Set with High-Resource SQL Found in AWR

Now the SQL tuning set is populated with the top 15 SQL statements ordered by disk reads. The begin and end AWR snapshot IDs are 26800 and 26900 respectively:

```
DECLARE
  base_cur dbms_sqltune.sqlset_cursor;
BEGIN
  OPEN base_cur FOR
    SELECT value(x)
    FROM table(dbms_sqltune.select_workload_repository(
      26800,26900, null, null,'disk_reads',
      null, null, null, 15)) x;
  --
  dbms_sqltune.load_sqlset(
    sqlset_name => 'IO_STS',
    populate_cursor => base_cur);
END;
/
```

The prior code populates the top 15 SQL statements contained in the AWR ordered by disk reads. The DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY function is used to populate a PL/SQL cursor with AWR information based on a ranking criterion. Next the DBMS_SQLTUNE.LOAD_SQLSET procedure is used to populate the SQL tuning set using the cursor as input.

How It Works

The DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY function can be used in a variety of ways to populate a SQL tuning set using queries in the AWR. You can instruct it to load SQL statements by criteria such as disk reads, elapsed time, CPU time, buffer gets, and so on. See Table 11-5 for descriptions for parameters to this function. When designating the AWR as input, you can use either of the following:

- Begin and end AWR snapshot IDs
- An AWR baseline that you've previously created

You can view the details of the SQL tuning set (created in the "Solution" section) via this query:

```
SELECT
 sqlset_name
,elapsed_time
,cpu_time
,buffer_gets
,disk_reads
,sql_text
FROM dba_sqlset_statements
WHERE sqlset_name = 'IO_STS';
```

11-11. Populating a SQL Tuning Set from Resource-Consuming SQL in Memory

Problem

You want to populate a tuning set from high resource-consuming SQL statements that are currently in the memory.

Solution

Use the DBMS_SQLTUNE.SELECT_CURSOR_CACHE function to populate a SQL tuning set with statements currently in memory. This example creates a tuning set and populates it with high-load resource-consuming statements not belonging to the SYS schema and having disk reads greater than 1,000,000:

```
-- Create the tuning set
EXEC DBMS_SQLTUNE.CREATE_SQLSET('HIGH_DISK_READS');
-- populate the tuning set from the cursor cache
DECLARE
  cur DBMS_SQLTUNE.SQLSET_CURSOR;
BEGIN
  OPEN cur FOR
  SELECT VALUE(x)
  FROM table(
  DBMS_SQLTUNE.SELECT_CURSOR_CACHE(
  'parsing_schema_name <> ''SYS'' AND disk_reads > 1000000',
  NULL, NULL, NULL, NULL, 1, NULL,'ALL')) x;
--
  DBMS_SQLTUNE.LOAD_SQLSET(sqlset_name => 'HIGH_DISK_READS',
    populate_cursor => cur);
END;
/
```

In the prior code, notice that the SYS user is bookended by sets of two single quotes (not double quotes). The SELECT_CURSOR_CACHE function loads the SQL statements into a PL/SQL cursor, and the LOAD_SQLSET procedure populates the SQL tuning set with the SQL statements.

How It Works

The DBMS_SQLTUNE.SELECT_CURSOR_CACHE function (see Table 11-6 for function parameter descriptions) allows you to extract from memory SQL statements and associated statistics into a SQL tuning set. The procedure allows you to filter SQL statements by various resource-consuming criteria, such as the following:

- ELAPSED_TIME
- CPU_TIME
- BUFFER_GETS
- DISK_READS
- DIRECT_WRITES
- ROWS_PROCESSED

This allows you a great deal of flexibility on how to filter and populate the SQL tuning set.

11-12. Populating SQL Tuning Set with All SQL in Memory

Problem

You want to create a SQL tuning set and populate it with all SQL statements currently in memory.

Solution

Use the DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET procedure to efficiently capture all of the SQL currently stored in the cursor cache (in memory). This example creates a SQL tuning set named PROD_WORKLOAD and then populates by sampling memory for 3,600 seconds (waiting 20 seconds between each polling event):

```
BEGIN
  -- Create the tuning set
  DBMS_SQLTUNE.CREATE_SQLSET(
    sqlset_name => 'PROD_WORKLOAD'
   ,description => 'Prod workload sample');
  --
  DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET(
    sqlset_name     => 'PROD_WORKLOAD'
   ,time_limit      => 3600
   ,repeat_interval => 20);
END;
/
```

How It Works

The DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET procedure allows you to poll for queries and memory and use any queries found to populate a SQL tuning set. This is a powerful technique that you can use when it's required to capture a sample set of all SQL statements executing.

You have a great deal of flexibility on instructing DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET to capture SQL statements in memory (see Table 11-7 for details on all parameters). For example, you can instruct the procedure to capture a cumulative set of statistics for each SQL statement by specifying a CAPTURE_MODE of DBMS_SQLTUNE.MODE_ACCUMULATE_STATS.

```
BEGIN
  DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET(
    sqlset_name     => 'PROD_WORKLOAD'
   ,time_limit      => 60
   ,repeat_interval => 10
   ,capture_mode    => DBMS_SQLTUNE.MODE_ACCUMULATE_STATS);
END;
/
```

This is more resource-intensive than the default settings, but produces more accurate statistics for each SQL statement.

Table 11-7. *CAPTURE_CURSOR_CACHE_SQLSET Parameter Descriptions*

Parameter	Description	Default Value
SQLSET_NAME	SQL tuning set name	none
TIME_LIMIT	Total time in seconds to spend sampling	1800
REPEAT_INTERVAL	While sampling, amount of time to pause in seconds before polling memory again	300
CAPTURE_OPTION	Either INSERT, UPDATE, or MERGE statements when new statements are detected	MERGE
CAPTURE_MODE	When capture option is UPDATE or MERGE, either replace statistics or accumulate statistics. Possible values are MODE_REPLACE_OLD_STATS or MODE_ACCUMULATE_STATS.	MODE_REPLACE_OLD_STATS
BASIC_FILTER	Filter type of statements captured	NULL
SQLSET_OWNER	SQL tuning set owner; NULL indicates the current user.	NULL
RECURSIVE_SQL	Include (or not) recursive SQL; possible values are HAS_RECURSIVE_SQL, NO_RECURSIVE_SQL.	HAS_RECURSIVE_SQL

11-13. Displaying the Contents of a SQL Tuning Set

Problem

You have populated a SQL tuning set and want to verify its characteristics such as the SQL statements and corresponding statistics.

Solution

You can determine the name and number of SQL statements for SQL tuning sets in your database via this query:

```
SELECT name, created, statement_count
FROM dba_sqlset;
```

Here is some sample output:

```
NAME                           CREATED   STATEMENT_COUNT
------------------------------ --------- ---------------
test1                          19-APR-11              29
```

Use the following query to display the SQL text and associated statistical information for each query within the SQL tuning set:

```
SELECT sqlset_name, elapsed_time, cpu_time, buffer_gets, disk_reads, sql_text
FROM dba_sqlset_statements;
```

Here is a small snippet of the output. The `SQL_TEXT` column has been truncated in order to fit the output on the page:

```
SQLSET_NAME     ELAPSED_TIME   CPU_TIME BUFFER_GETS DISK_READS SQL_TEXT
--------------- ------------ ---------- ----------- ---------- ----------------------------
test1              235285363   45310000      112777       3050 INSERT ......
test1               52220149   22700000      328035      18826  delete from.....
```

How It Works

Recall that a SQL tuning set consists of one or more SQL statements and the corresponding execution statistics. This information is viewable from the `DBA_SQLSET_*` views. Table 11-8 describes the type of SQL tuning set information contained within each of these views.

Table 11-8. *Views Containing SQL Tuning Set Information*

View Name	Description
DBA_SQLSET	Displays information regarding SQL tuning sets
DBA_SQLSET_BINDS	Displays bind variable information associated with SQL tuning sets
DBA_SQLSET_PLANS	Shows execution plan information for queries in a SQL tuning set
DBA_SQLSET_STATEMENTS	Contains SQL text and associated statistics
DBA_SQLSET_REFERENCES	Shows whether a SQL tuning set is active

You can also use the DBMS_SQLTUNE.SELECT_SQLSET function to retrieve information about SQL tuning sets—for example:

```
SELECT
 sql_id
,elapsed_time
,cpu_time
,buffer_gets
,disk_reads
,sql_text
FROM TABLE(DBMS_SQLTUNE.SELECT_SQLSET('&&sqlset_name'));
```

Whether you use the DBMS_SQLTUNE.SELECT_SQLSET function or directly query the data dictionary views depends entirely on your personal preference or business requirement.

11-14. Selectively Deleting Statements from a SQL Tuning Set

Problem

You want to prune SQL statements from an STS that don't meet a performance measure, such as queries that have less than 2,000,000 disk reads.

Solution

First view the existing SQL information associated with an STS:

```
select sqlset_name, disk_reads, cpu_time, elapsed_time, buffer_gets
from dba_sqlset_statements;
```

Here is some sample output:

```
SQLSET_NAME                    DISK_READS   CPU_TIME ELAPSED_TIME BUFFER_GETS
------------------------------ ---------- ---------- ------------ -----------
IO_STS                            3112941 3264960000   7805935285     2202432
IO_STS                            2943527 3356460000   8930436466     1913415
IO_STS                            2539642 2310610000   5869237421     1658465
IO_STS                            1999373 2291230000   6143543429     1278601
IO_STS                            1993973 2243180000   5461607976     1272271
IO_STS                            1759096 1930320000   4855618689     1654252
```

Now use the DBMS_SQLTUNE.DELETE_SQLSET procedure to remove SQL statements from the STS based on the specified criterion. This example removes SQL statements that have less than 2,000,000 disk reads from the SQL tuning set named IO_STS:

```
BEGIN
  DBMS_SQLTUNE.DELETE_SQLSET(
    sqlset_name  => 'IO_STS'
   ,basic_filter => 'disk_reads < 2000000');
END;
/
```

How It Works

The key to understanding is that a SQL tuning set consists of the following:

- One or more SQL statements
- Associated metrics/statistics for each SQL statement

Because the metrics/statistics are part of the STS, you can remove SQL statements from a SQL tuning set based on characteristics of the associated metrics/statistics. You can use the DBMS_SQLTUNE.DELETE_SQLSET procedure to remove statements from the STS based on statistics such as the following:

- ELAPSED_TIME
- CPU_TIME
- BUFFER_GETS
- DISK_READS
- DIRECT_WRITES
- ROWS_PROCESSED

If you want to delete all SQL statements from a SQL tuning set, then don't specify a filter—for example:

```
SQL> exec  DBMS_SQLTUNE.DELETE_SQLSET(sqlset_name  => 'IO_STS');
```

11-15. Transporting a SQL Tuning Set

Problem

You've identified some resource-intensive SQL statements in a production environment. You want to transport these statements and associated statistics to a test environment, where you can tune the statements without impacting production.

Solution

The following steps are used to copy a SQL tuning set from one database to another:

1. Create a staging table in source database.
2. Populate the staging table with STS data.
3. Copy the staging table to the destination database.
4. Unpack the staging table in the destination database.

The prior steps are elaborated on in the following subsections.

Step 1: Create a Staging Table in the Source Database

Use the `DBMS_SQLTUNE.CREATE_STGTAB_SQLSET` procedure to create a table that will be used to contain the SQL tuning set metadata. This example creates a table named `STS_TABLE`:

```
BEGIN
  dbms_sqltune.create_stgtab_sqlset(
  table_name => 'STS_TABLE'
 ,schema_name => 'MV_MAINT');
END;
/
```

Step 2: Populate Staging Table with STS Data

Now populate the staging table with STS metadata using `DBMS_SQLTUNE.PACK_STGTAB_SQLSET`:

```
BEGIN
  dbms_sqltune.pack_stgtab_sqlset(
  sqlset_name          => 'IO_STS'
 ,sqlset_owner         => 'SYS'
 ,staging_table_name   => 'STS_TABLE'
 ,staging_schema_owner => 'MV_MAINT');
END;
/
```

If you're unsure of the names of the STS you want to transport, run the following query to get the details:

```
SELECT name, owner, created, statement_count
FROM dba_sqlset;
```

Step 3: Copy the Staging Table to the Destination Database

You can copy the table from one database to the other via Data Pump, the old `exp`/`imp` utilities, or by using a database link. This example creates a database link in the destination database and then copies the table from the source database:

```
create database link source_db
connect to mv_maint
identified by foo
using 'source_db';
```

In the destination database, the table can be copied directly from the source with the `CREATE TABLE AS SELECT` statement:

```
SQL> create table STS_TABLE as select * from STS_TABLE@source_db;
```

Step 4: Unpack the Staging Table in the Destination Database

Use the `DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET` procedure to take the contents of the staging table and populate the data dictionary with the SQL tuning set metadata. This example unpacks all SQL tuning sets contained within the staging table:

```
BEGIN
DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET(
  sqlset_name        => '%'
 ,replace            => TRUE
 ,staging_table_name => 'STS_TABLE');
END;
/
```

How It Works

A SQL tuning set consists of one or more queries and corresponding execution statistics. You will occasionally have a need to copy a SQL tuning set from one database to another. For example, you might be having performance problems with a production database but want to capture and move the top resource-consuming statements to a test database where you can diagnose the SQL (within the STS) without impacting production.

Keep in mind that an STS can be used as input for any of the following tools:

- SQL Tuning Advisor
- SQL Access Advisor

- SQL Plan Management
- SQL Performance Analyzer

The prior tools are used extensively to troubleshoot and test SQL performance. Transporting a SQL tuning set from one environment to another allows you to use these tools in a testing or development environment.

■ **Note** SQL tuning sets can be transported to Oracle Database 10g R2 or higher versions of the database only.

11-16. Creating a Tuning Task

Problem

You realize that as part of manually running the SQL Tuning Advisor, you need to first create a tuning task.

■ **Tip** Refer to Figure 11-1 for the details on the flow of processes required when manually running the SQL Tuning Advisor.

Solution

Use the `DBMS_SQLTUNE.CREATE_TUNING_TASK` procedure to create a SQL tuning task. You can use the following as inputs when creating a SQL tuning task:

- Text for a specific SQL statement
- SQL identifier for a specific SQL statement from the cursor cache in memory
- Single SQL statement from the AWR given a range of snapshot IDs
- SQL tuning set name (see Recipes 11-7 through 11-11 for details on how to create a SQL tuning set)

Examples of the prior techniques for creating a SQL tuning task are described in the following subsections.

■ **Note** The user creating the tuning task needs the `ADMINISTER SQL MANAGEMENT OBJECT` system privilege.

Text for a Specific SQL Statement

This example provides the text of a SQL statement when creating the tuning task:

```
DECLARE
  tune_task VARCHAR2(30);
  tune_sql  CLOB;
BEGIN
  tune_sql := 'select count(*) from mgmt_db_feature_usage_ecm';
  tune_task := DBMS_SQLTUNE.CREATE_TUNING_TASK(
    sql_text    => tune_sql
   ,user_name   => 'MV_MAINT'
   ,scope       => 'COMPREHENSIVE'
   ,time_limit  => 60
   ,task_name   => 'tune_test'
   ,description => 'Provide SQL text'
);
END;
/
```

SQL ID for a Specific SQL Statement from the Cursor Cache

First identify the `SQL_ID` by querying `V$SQL`:

```
SELECT sql_id, sql_text
FROM v$sql where sql_text like '%&&mytext%';
```

Once you have the `SQL_ID`, you can provide it as input to `DBMS_SQLTUNE.CREATE_TUNING_TASK`:

```
DECLARE
  tune_task VARCHAR2(30);
  tune_sql  CLOB;
BEGIN
  tune_task := DBMS_SQLTUNE.CREATE_TUNING_TASK(
    sql_id      => '98u3gf0xzq03f'
   ,task_name   => 'tune_test2'
   ,description => 'Provide SQL ID'
);
END;
/
```

Single SQL Statement from the AWR Given a Range of Snapshot IDs

Here's an example of creating a SQL tuning task by providing a `SQL_ID` and range of AWR snapshot IDs:

```
DECLARE
  tune_task VARCHAR2(30);
  tune_sql  CLOB;
```

```
BEGIN
  tune_task := DBMS_SQLTUNE.CREATE_TUNING_TASK(
    sql_id      => '1tbu2jp7kv0pm'
   ,begin_snap  => 21690
   ,end_snap    => 21864
   ,task_name   => 'tune_test3'
);
END;
/
```

If you're not sure which SQL_ID (and associated query) to use, then run this query:

```
SQL> select sql_id, sql_text from dba_hist_sqltext;
```

If you're unaware of the available snapshot IDs, then run this query:

```
SQL> select snap_id from dba_hist_snapshot order by 1;
```

■ **Tip** By default, the AWR contains only high resource-consuming queries. You can modify this behavior and ensure that a specific SQL statement is included in every snapshot (regardless of its resource consumption) by adding it to the AWR via the following:

```
SQL> exec dbms_workload_repository.add_colored_sql('98u3gf0xzq03f');
```

SQL Tuning Set Name

If you have the requirement of running the SQL Tuning Advisor against multiple SQL queries, then a SQL tuning set is required. To create a tuning task using a SQL tuning set as input, do so as follows:

```
SQL> variable mytt varchar2(30);
SQL> exec :mytt := DBMS_SQLTUNE.CREATE_TUNING_TASK(sqlset_name => 'IO_STS');
SQL> print :mytt
```

How It Works

Before manually executing the SQL Tuning Advisor, you first need to define what SQL statements will be used as input. You do this by creating a SQL tuning task. Oracle provides a great deal of flexibility on how you add SQL statements to a tuning task. As shown in the "Solution" section, you can do the following:

- Hard-code the text for a specific SQL query
- Use a SQL query in memory
- Use a SQL query in the AWR
- Define a SQL tuning set when tuning multiple queries

The prior techniques provide a variety of ways to identify SQL statements to be analyzed by the SQL Tuning Advisor. Once you've created a tuning task, you can view its details via this query:

```
select owner, task_name, advisor_name, created
from dba_advisor_tasks
order by created;
```

Once you have created a tuning task, you can now manually execute the SQL Tuning Advisor (Recipe 11-17). If you need to drop the tuning task, you can do so as follows:

```
SQL> exec dbms_sqltune.drop_tuning_task(task_name => '&&task_name');
```

11-17. Manually Running SQL Tuning Advisor

Problem

You want to manually execute SQL Tuning Advisor and get tuning advice for a SQL statement.

Solution

Use the following steps to manually run the SQL Tuning Advisor:

1. Create a tuning task (see Recipe 11-16 for complete details); this defines which SQL statements will be tuned. This can be a single SQL statement or several SQL statements within a SQL tuning set.
2. Execute the tuning task.
3. Display the results of the tuning task.

This example runs the SQL Tuning Advisor for a single SQL statement. First a tuning task is created.

```
DECLARE
  tune_task VARCHAR2(30);
  tune_sql  CLOB;
BEGIN
  tune_sql := 'select a.emp_id, b.dept_name ' ||
              'from emp a, dept b ' ||
              'where a.dept_id = b.dept_id';
  --
  tune_task := DBMS_SQLTUNE.CREATE_TUNING_TASK(
    sql_text    => tune_sql
   ,user_name   => 'MV_MAINT'
   ,scope       => 'COMPREHENSIVE'
   ,time_limit  => 60
   ,task_name   => 'tune_test'
   ,description => 'Tune a SQL statement.'
);
END;
/
```

Next the tuning task is executed:

```
SQL> exec dbms_sqltune.execute_tuning_task(task_name => 'tune_test');
```

Lastly, a report is generated that displays the tuning advice:

```
SQL> set long 10000 longchunksize 10000
SQL> set linesize 132 pagesize 200
SQL> select dbms_sqltune.report_tuning_task('tune_test') from dual;
```

Here is some sample output:

```
1- Statistics Finding
---------------------
  Table "MV_MAINT"."DEPT" was not analyzed.
  Recommendation
  --------------
  - Consider collecting optimizer statistics for this table.
....
2- Index Finding (see explain plans section below)
--------------------------------------------------
  The execution plan of this statement can be improved by creating one or more
  indices.
  Recommendation (estimated benefit: 97.98%)
  ------------------------------------------
  - Consider running the Access Advisor to improve the physical schema design
    or creating the recommended index.
    create index MV_MAINT.IDX$$_21E10001 on MV_MAINT.EMP("DEPT_ID");
```

The prior output has specific recommendations on generating statistics for a table in the query and adding an index. You'll need to test the recommendations to ensure that performance does improve before implementing them in a production environment.

OPTIMIZER TUNING MODES

The optimizer operates in two different modes: normal and tuning. When a SQL statement executes, the optimizer operates in normal mode and quickly identifies a reasonable execution plan. In this mode, the optimizer spends only a fraction of a second to try to determine the best plan.

When the SQL Tuning Advisor analyzes a query, it runs the optimizer in tuning mode. In this mode, the optimizer can take several minutes to analyze each step of the execution plan and generate an execution plan that is potentially much more efficient than the plan generated under normal mode.

This is somewhat similar to a computer chess game. When you allow the chess software to spend only a second or less on each move, it's easy to beat the game. However, if you allow the chess game to spend a minute or more on each move, in this mode the game makes much more optimal decisions.

How It Works

The SQL Tuning Advisor helps automate the task of tuning poorly performing queries. The tool is fairly easy to use, and it provides suggestions on how to tune a query, such as the following:

- Rewriting the SQL
- Adding indexes
- Implementing a SQL profile or plan baselines
- Generating statistics

You can also manually run the SQL Tuning Advisor from either SQL Developer or Enterprise Manager. Running the SQL Tuning Advisor from these tools is briefly described in the next two subsections.

Running SQL Tuning Advisor from SQL Developer

If you have access to SQL Developer 3.0 or higher, then it's very easy to run the SQL Tuning Advisor for a query. Follow these simple steps:

1. Open a SQL worksheet.
2. Type in the query.
3. Click the button associated with the SQL Tuning Advisor.

You will be presented with any findings and recommendations. If you have access to SQL Developer (it's a free download), this is the easiest way to run the SQL Tuning Advisor.

■ **Note** Before running SQL Tuning Advisor, ensure the user that you're connected to has the `ADVISOR` system privilege granted to it.

Running SQL Tuning Advisor from Enterprise Manager

You can also run the advisor from within Enterprise Manager. Log into Enterprise Manager and follow these steps:

1. From the main database page, click the Advisor Central link (near the bottom).
2. Under the Advisors section, click the SQL Advisors link.
3. Click the SQL Tuning Advisor link.

You should be presented with a page similar to the one shown in Figure 11-3.