You must use a job template when creating a lightweight job, with the template containing the metadata for the lightweight job as well as the privileges to be inherited by the lightweight job. You can use a stored procedure or a Scheduler program as a job template. You must reference a Scheduler program in order to specify a job action. The program type must be a PLSQL_BLOCK or STORED_PROCEDURE. If a user has privileges on the program, that user will automatically have privileges on the lightweight job.

You can use the following query to find out the details about lightweight jobs in your database:

```
SQL> SELECT job_name, program_name FROM dba_scheduler_jobs
       WHERE job_style='LIGHTWEIGHT';
```

```
JOB_NAME              PROGRAM_NAME
-----------          -------------
TEST_JOB1            TEST_PROG1
```

Unlike the regular database jobs, lightweight jobs aren’t shown in the DBA_SCHEDULER_JOBS view, since lightweight jobs aren’t schema objects like regular jobs.

You create a lightweight job in a manner similar to how you create a regular job, by executing the CREATE_JOB procedure. Just specify the value LIGHTWEIGHT for the JOB_STYLE parameter, instead of REGULAR, the default value for this parameter. Here’s an example showing how to create a lightweight job:

```
begin
    dbms_scheduler.create_job (
        job_name => 'test_ltwtjob1',
        program_name => 'test_prog',
        repeat_interval => 'freq=daily,by_hour=10',
        end_time => '31-DEC-08 06:00:00 AM Australia/Sydney',
        job_style => 'lightweight',
        comments => 'A lightweight job based on a program');
end;
```

In this example, the program test_prog serves as the template for the lightweight job TEST_LTWTJOB1. You can also specify a schedule instead of the REPEAT_INTERVAL and the END_TIME attributes.

You can use a job array to create a set of Scheduler lightweight jobs. The job array comes in handy when you have to create a large number of Scheduler jobs. The following example shows how to create a set of lightweight jobs using a job array:

1. Create two variables, one to define the Scheduler job and the other for the job array definition.
   
   ```
   declare
   testjob sys.job;
   testjobarr sys.job_array;
   ```

2. Use the sys.job_array constructor to initialize the job array.
   
   ```
   begin
   testjobarr := sys.job_array();
   ```

   When you initialize the job array testjobarr, which is an array of JOB object types, the database creates a slot for a single job in that array.
3. You must set the size of the job array to the number of jobs you expect to create.

   testjobarr.extend(500);

   The statement shown here allocates space in the job array to hold information for 500 jobs.

4. The following code creates the 500 jobs and places them in the job array:

   for I in 1 . . . 500 loop
     testjob := sys.job(job_name => 'TESTJOB'||TO_CHAR(I),
                      job_style    => 'LIGHTWEIGHT',
                      job_template => 'TEST_PROG',
                      enabled      => TRUE);
     testjobarr(i) := TESTJOB;
   end loop;

   The code shown here creates all 500 jobs using the TEST_PROG template. The jobs are added to the job array by the assignment operator testjobarr(i).

5. Use the CREATE_JOBS procedure to submit the array consisting of 500 jobs.

   dbms_scheduler.create_jobs (testjobarr, 'transactional');

   The CREATE_JOBS procedure creates all 500 jobs at once. In this example, I chose to create lightweight jobs as part of the array, by specifying LIGHTWEIGHT as the value for the JOB_STYLE parameter when I created the job array. By not specifying the JOB_STYLE parameter, I can create a job array of regular database jobs instead of lightweight jobs. This is so, because the default value of the JOB_STYLE parameter is REGULAR.

Managing External Jobs

External jobs are operating system executables that you run outside the database. You specify EXECUTABLE as the value for the JOB_TYPE parameter for an external job. If you use a named program for an external job, you must specify the complete directory path, for example, /usr/local/bin/perl, where you stored the executable, either in the JOB_ACTION attribute or the PROGRAM_ACTION attribute.

You can create local external jobs and remote external jobs. A local external job runs on the same server as the job-originating database, and a remote external job runs on a remote host. You can use remote external jobs to manage jobs across your entire network from a single database. The interesting thing about remote external jobs is that you don’t need to have an Oracle database instance running on the remote hosts. You’ll just need to install a Scheduler Agent on each of the remote hosts where you wish to run external jobs, to accept job requests from the job-originating database, execute them on the remote host, and transmit the job results to the job-originating database.

Running local external jobs is straightforward. All you need to do is to specify EXECUTABLE as the value for the JOB_TYPE or PROGRAM_TYPE arguments. To run remote external jobs, you’ll need to install and configure the Scheduler Agent as well as assign credentials for executing the remote jobs. I explain the steps involved in setting up remote external jobs in the following sections.

Setting Up the Database

You must set up the database from where you want to issue external job requests by doing the following:
1. Since you'll need the Oracle XML DB to run a remote external job, first check whether the Oracle XML DB option has been successfully installed in your database by issue the following DESCRIBE command:

```
SQL> desc resource_view
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES</td>
<td></td>
<td>XMLTYPE (XMLSchema &quot;<a href="http://xmlns.oracle.com/xdb/XDBResource.xsd">http://xmlns.oracle.com/xdb/XDBResource.xsd</a>&quot; Element &quot;Resource&quot;)</td>
</tr>
<tr>
<td>ANY_PATH</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
<tr>
<td>RESID</td>
<td></td>
<td>RAW(16)</td>
</tr>
</tbody>
</table>

SQL>

The DESCRIBE command shows that the Oracle XML DB option is correctly installed. If the query shows that Oracle XML DB isn't an option, you must install it before you can proceed.

2. Execute the Oracle-provided prvtsch.plb script, located in the $ORACLE_HOME/rdbms/admin directory.

```
SQL> connect sys/sammyy1 as sysdba
SQL> @$ORACLE_HOME/rdbms/admin/prvtsch.plb
PL/SQL procedure successfully completed.
... PL/SQL procedure successfully completed.
o rows selected
Package created.
Package body created.
No errors.
...
User altered.
SQL>
```

3. Finally, set a registration password for the Scheduler Agent.

```
SQL> EXEC dbms_scheduler.set_agent_registration_pass(
registration_password => 'sammyy1'.-
expiration_date       => systimestamp + interval '7' day,-
max_uses              => 25)

PL/SQL procedure successfully completed.
SQL>
```

The Scheduler Agent uses the password to register with the database. The EXPIRATION_DATE and the MAX_USES parameters show the date when the password expires and the number of times the password can be used, respectively.
Installing and Configuring the Scheduler Agent

You must install the Scheduler Agent on every remote host where you plan on running external jobs. You can either download the software for installation from Oracle or use the Database CD pack. In either case, you’ll need to use the installation media for the Oracle Database Gateway. Here are the steps to install the Scheduler Agent:

1. Log in as the Oracle software owner (usually the user Oracle).
2. Go to where the Oracle Database Gateway installation files are stored and issue the following command to start up the Oracle Universal Installer:
   
   ```bash
   $ /oracle11g/gateways/runInstaller
   ```
3. On the Welcome screen, click Next.
4. On the Select a Product page, select Oracle Scheduler Agent 11.1.0.6.0, and click Next.
5. On the Specify Home Details page, select a name and provide the directory path for the Oracle Scheduler Agent home. Click Next.
6. On the Oracle Scheduler Agent page, provide the host name and the port number the agent must use to communicate with the external job request originating database. Click Next.
7. On the Summary page, review the selections you made and click Install.

---

**Note** You can also use a silent install to automate the installation of the Scheduler Agent on a larger number of hosts.

---

8. When the installer prompts you to run the `root.sh` script as the root user, do so and click OK.
9. Click Exit after you see the End of Installation page.

You need to use the `schagent` executable to invoke the Scheduler Agent. But first, you must register the agent with the database from where you want run an external job on the host where you installed the Scheduler Agent. Here’s how you register the Scheduler Agent with a database:

```bash
$ schagent -registerdatabase prod1 1522
```

In the example, the database host is named `prod1`, and the port number assigned to the Scheduler Agent is `1522`. Once you issue this command, you’ll be prompted to supply the agent registration password you created earlier:

```bash
$.schagent -registerdatabase localhost.localdomain 1522
Agent Registration Password ? ******
$
```

You start the Scheduler Agent by issuing the following command:

```bash
$.schagent -start
Scheduler agent started
$
```

Stop the agent by issuing the following command:

```bash
$.schagent -stop
Scheduler agent stopped
$
```
The preceding examples show how to work with the Scheduler Agent on a UNIX/Linux system. You must install the OracleSchedulerExecutionAgent service before you can use the agent. You can install the service in the following way:

```
$ schagent --installagentservice
```

The OracleSchedulerExecutionAgent service is different from the Oracle service that you use to start and stop an Oracle instance on a Windows server.

### Creating and Enabling Remote External Jobs

Since an external job must execute as an operating system user's job, the Scheduler lets you assign operating system credentials to an external job. You use a credential, which is a schema object that contains a username and password combination, to designate the credentials for an external job. Use the `CREDENTIAL_NAME` attribute when you create an external job to specify the credentials for executing that job.

You aren't required to specify credentials for a local external job, although Oracle recommends that you do so. Before you can create a remote external job, you must first create a credential. You can then assign that credential object to the user under whose account the remote external executable will be run. Note that a user must have the execute privilege on a credential object before the user can use that credential to execute a job.

Here are the steps you must follow to create a remote external job:

1. First, execute the `CREATE_CREDENTIAL` procedure to create a credential object.
   ```sql
   SQL> exec dbms_scheduler.create_credential('hrcredential, 'hr', 'sammyy1');
   ```

2. Grant privileges on the newly created credential to the user who'll need to use the credential.
   ```sql
   SQL> grant execute on system.hrcrdential to sam;
   ```
   
   You can query the `DBA_SCHEDULER_VIEW` to examine all credentials in the database.

3. Create a remote external job by executing the `CREATE_JOB` procedure.
   ```sql
   SQL> begin
   2   dbms_scheduler.create_job(
   3       job_name => 'remove_logs',
   4       job_type => 'executable',
   5       job_action => '/u01/app/oracle/logs/removelogs',
   6       repeat_interval => 'freq=daily; byhour=23',
   7       enabled => false);
   8   end;
   SQL> /
   ```
   
   PL/SQL procedure successfully completed.

4. Once you create the remote external job REMOVE_LOGS, set the `CREDENTIAL_NAME` attribute of the remote job by executing the `SET_ATTRIBUTE` procedure.
   ```sql
   SQL> exec dbms_scheduler.set_attribute('remove_logs', 'credential_name', 'hrcredential');
   ```
   
   PL/SQL procedure successfully completed.

SQL>
5. Execute the SET_ATTRIBUTE procedure again, this time to set the DESTINATION attribute.
   SQL> exec dbms_scheduler.set_attribute('remove_logs',
   2  'destination', 'localhost.localdomain:1521');
   PL/SQL procedure successfully completed.

   SQL>

6. Execute the ENABLE procedure to enable the external job.
   SQL> exec dbms_scheduler.enable('remove_logs');
   PL/SQL procedure successfully completed.
   SQL>

   You can disable the capability to run external jobs in a database by dropping the user remote_
   scheduler_agent, who is created by the prvtsch.plb script that you ran earlier.

   SQL> drop user remote_scheduler_agent cascade;

   You must reexecute the prvtsch.plb script for the database to run a remote external job, once
   you drop the remote_scheduler_agent.

Managing Programs

A program contains metadata about what the Scheduler will run, including the name and type of the
program, and what a job will execute. Different jobs can share a single program.

Creating a Program

You create a new program using the CREATE_PROGRAM procedure of the DBMS_SCHEDULER
package, as shown here:

   SQL> BEGIN
   2  DBMS_SCHEDULER.CREATE_PROGRAM(
   3  PROGRAM_NAME   => 'MY_PROGRAM',
   4  PROGRAM_ACTION => 'UPDATE_SCHEMA_STATS',
   5  PROGRAM_TYPE   => 'STORED_PROCEDURE',
   6  enabled        => TRUE);
   7* end;
   SQL> /
   PL/SQL procedure successfully completed.
   SQL>

   Once you create a program, you can simplify your job creation statement by replacing the
   JOB_TYPE and JOB_ACTION attributes with the name of the program that already contains the specifi-
   cation of these attributes. The PROGRAM_TYPE and PROGRAM_ACTION attributes thus replace the job
   attributes that you normally provide when creating a new job. You can see why this type of modular
   approach is beneficial—different jobs can use the same program, thus simplifying the creation of
   new jobs.

   The following example re-creates the TEST_JOB job that was created in Listing 18-17, but using
   the program component this time:
SQL> BEGIN
  2  DBMS_SCHEDULER.CREATE_JOB(
  3  JOB_NAME => 'TEST_JOB',
  4  PROGRAM_NAME => 'TEST_PROGRAM',
  5  REPEAT_INTERVAL => 'FREQ=DAILY;BYHOUR=12', ENABLED => TRUE);
  7* END;
SQL> /
PL/SQL procedure successfully completed.

In the preceding example, using a program lets you avoid specifying the JOB_TYPE and JOB_ACTION parameters in the CREATE_JOB statement.

Administering Programs

You can enable, disable, and drop Scheduler programs using various procedures from the DBMS_SCHEDULER package, as shown in the following examples.

The ENABLE procedure is used to enable a Scheduler program:

SQL> EXEC DBMS_SCHEDULER.ENABLE('TEST_PROGRAM');
PL/SQL procedure successfully completed.

You use the DISABLE procedure to disable a program:

SQL> EXEC DBMS_SCHEDULER.DISABLE('TEST_PROGRAM');
PL/SQL procedure successfully completed.

The DROP_PROGRAM procedure is used to drop a program:

SQL> EXEC DBMS_SCHEDULER.DROP_PROGRAM('TEST_PROGRAM');
PL/SQL procedure successfully completed.

Managing Schedules

Let’s say you have a number of jobs, all of which execute at the same time. By using a common schedule, you can simplify the creation and managing of such jobs. The following sections explain how you can manage schedules.

Creating a Schedule

You use the CREATE_SCHEDULE procedure of the DBMS_SCHEDULER package to create a schedule, as shown here:

SQL> BEGIN
  2  DBMS_SCHEDULER.CREATE_SCHEDULE(
  3  SCHEDULE_NAME => 'TEST_SCHEDULE',
  4  START_DATE => SYSTIMESTAMP,
  5  END_DATE => SYSTIMESTAMP + 90,
  6  REPEAT_INTERVAL => 'FREQ=HOURLY;INTERVAL= 4',
  7  COMMENTS => 'Every 4 hours');
  8* END;
SQL> /
PL/SQL procedure successfully completed.
The TEST_SCHEDULE schedule states that a job with this schedule will be executed immediately and then be reexecuted every 4 hours, for a period of 90 days. Note the following things about this new schedule:

- The CREATE_SCHEDULE procedure has three important parameters: START_DATE, END_DATE, and REPEAT_INTERVAL.
- You specify the start and end times using the TIMESTAMP WITH TIME_ZONE data type.
- You must use a calendaring expression when creating the repeat interval.

Once you create the TEST_SCHEDULE schedule, you can simplify the job creation process even further by using both a program and a schedule when creating a new job, as shown here:

```sql
SQL> BEGIN
  2  DBMS_SCHEDULER.CREATE_JOB(
  3    JOB_NAME => 'MY_JOB',
  4    PROGRAM_NAME => 'MY_PROGRAM',
  5    SCHEDULE_NAME => 'MY_SCHEDULE');
  6  END;
  7  /
PL/SQL procedure successfully completed.
SQL>
```

As you can see, using saved schedules and programs makes creating new jobs a breeze.

### Administering Schedules

You can alter various attributes of a schedule by using the SET_ATTRIBUTE procedure of the DBMS_SCHEDULER package. You can alter all attributes except the name of the schedule itself.

You can drop a schedule by using the DROP_SCHEDULE procedure, as shown here:

```sql
SQL> BEGIN
  2  DBMS_SCHEDULER.DROP_SCHEDULE (SCHEDULE_NAME => 'TEST_SCHEDULE');
  3  END;
  4  /
PL/SQL procedure successfully completed.
SQL>
```

If a job or window is using the schedule you want to drop, your attempt to drop the schedule will result in an error instead, by default. You can force the database to drop the schedule anyway, by using an additional FORCE parameter in the preceding example and setting it to TRUE.

---

**Tip** When you create a schedule, Oracle provides access to PUBLIC, thus letting all users use your schedule by default.

### Managing Chains

A Scheduler chain consists of a set of related programs that run in a specified sequence. The successive positions in the chain are referred to as “steps” in the chain, and each step can point to another chain, a program, or an event. The chain includes the “rules” that determine what is to be done at each step of the chain.

We’ll create a simple Scheduler chain by first creating a Scheduler chain object, and then the chain steps and the chain rules.
Creating a Chain

Since Scheduler chains use Oracle Streams Rules Engine objects, a user must have both the CREATE JOB privilege and the Rules Engine privileges to create a chain. You can grant all the necessary Rules Engine privileges by using a statement like this, which grants the privileges to the user nina:

```sql
BEGIN
    DBMS_RULE_ADM.GRANT_SYSTEM_PRIVILEGE(DBMS_RULE_ADM.CREATE_RULE_OBJ, 'nina'),
    DBMS_RULE_ADM.GRANT_SYSTEM_PRIVILEGE (DBMS_RULE_ADM.CREATE_RULE_SET_OBJ, 'nina'),
    DBMS_RULE_ADM.GRANT_SYSTEM_PRIVILEGE (DBMS_RULE_ADM.CREATE_EVALUATION_CONTEXT_OBJ, 'nina')
END;
```

Now that you have the necessary privileges, let's create a Scheduler chain called TEST_CHAIN using the CREATE_CHAIN procedure:

```sql
BEGIN
    DBMS_SCHEDULER.CREATE_CHAIN (chain_name            =>   'test_chain',
                                   rule_set_name         =>   NULL,
                                   evaluation_interval   =>   NULL,
                                   comments              =>   NULL);
END;
```

Next, define the steps for the new chain using the DEFINE_CHAIN_STEP procedure. Note that a chain step can point to a program, an event, or another chain:

```sql
BEGIN
    DBMS_SCHEDULER.DEFINE_CHAIN_STEP('test_chain', 'step1', 'program1');
    DBMS_SCHEDULER.DEFINE_CHAIN_STEP('test_chain', 'step2', 'program2');
    DBMS_SCHEDULER.DEFINE_CHAIN_STEP('test_chain', 'step3', 'program3');
END;
```

Finally, to make the chain operative, you must add rules to the chain using the DEFINE_CHAIN_RULE procedure. Chain rules determine when a step is run and specify the conditions under which a step is run. Usually, a rule specifies that a step be run based on the fulfillment of a specific condition. Here’s an example:

```sql
BEGIN
    DBMS_SCHEDULER.DEFINE_CHAIN_RULE('test_chain', 'TRUE', 'START step1');
    DBMS_SCHEDULER.DEFINE_CHAIN_RULE('test_chain', 'step1 COMPLETED',
                                       'Start step2, step3');
    DBMS_SCHEDULER.DEFINE_CHAIN_RULE('test_chain', 'step2 COMPLETED AND step3 COMPLETED', 'END');
END;
```

The first rule in the preceding example specifies that step1 be run, which means that the Scheduler will start program1. The second rule specifies that step2 (program2) and step3 (program3) be run if step1 has completed successfully ('step1 COMPLETED'). The final rule says that when step2 and step3 finish, the chain will end.

Enabling a Chain

You must enable a chain before you can use it. Here’s how to do so:

```sql
BEGIN
    DBMS_SCHEDULER.ENABLE ('test_chain');
END;
```
Embedding Jobs in Chains

In order to run a job within a Scheduler chain, you must create a job with the JOB_TYPE attribute set to CHAIN, and the JOB_ACTION attribute pointing to the name of the particular chain you wish to use. Of course, this means that you must first create the chain.

Here’s the syntax for creating a job for a Scheduler chain:

```
SQL> BEGIN
    DBMS_SCHEDULER.CREATE_JOB (
        JOB_NAME        => 'test_chain_job',
        JOB_TYPE        => 'CHAIN',
        JOB_ACTION      => 'test_chain',
        REPEAT_INTERVAL => 'freq=daily;byhour=13;byminute=0;bysecond=0',
        ENABLED         => TRUE);
END;
```

You also have the option of using the RUN_CHAIN procedure to run a chain without creating a job first. The procedure will create a temporary job and immediately run the chain. Here’s how you do this:

```
SQL> BEGIN
    DBMS_SCHEDULER.RUN_CHAIN (
        CHAIN_NAME    =>  'my_chain1',
        JOB_NAME      =>  'quick_chain_job',
        START_STEPS   =>  'my_step1, my_step2');
END;
```

As with the other components of the Scheduler, there are procedures that enable you to drop a chain, drop rules from a chain, disable a chain, alter a chain, and so on. For the details, please refer to the section about the DBMS_SCHEDULER package in the Oracle manual, PL/SQL Packages and Types Reference.

Managing Events

So far, you’ve seen how to create jobs with and without a schedule. When you create a job without a schedule, you’ll have to provide the start time and the frequency, whereas using a schedule enables you to omit these from a job specification. In both cases, the job timing is based on calendar time. However, you can create both jobs and schedules that are based strictly on events, not calendar time. We’ll briefly look at event-based jobs and schedules in the following sections.

Creating Event-Based Jobs

The following example shows how to create a Scheduler job using a program and an event. The job will start when the event, FILE ARRIVAL, occurs:

```
SQL> BEGIN
    dbms_scheduler.create_job(
        JOB_NAME            =>  test_job,
        PROGRAM_NAME        =>  test_program,
        START_DATE          =>  '01-AUG-08 5.00.00AM US/Pacific',
        EVENT_CONDITION     =>  'tab.user_data.event_name = ''FILE_ARRIVAL''',
        QUEUE_SPEC          =>  'test_events_q',
        ENABLED             =>  TRUE,
        COMMENTS            =>  'An event based job');
END;
```
There are two unfamiliar attributes in the preceding CREATE_JOB procedure, both of which are unique to event-based jobs:

- **EVENT_CONDITION**: The EVENT_CONDITION attribute is a conditional expression that takes its values from the event source queue table and uses Oracle Streams Advanced Queuing rules. You specify object attributes in this expression and prefix them with `tab.user_data`. Review the DBMS_AQADM package to learn more about Oracle Streams Advanced Queuing and related rules.

- **QUEUE_SPEC**: The QUEUE_SPEC attribute determines the queue into which the job-triggering event will be queued. In the preceding example, `test_events_q` is the name of the queue.

### Creating Event-Based Schedules

The following example shows how to create an event-based schedule. Whenever an event (`FILE_ARRIVAL`) occurs, the Scheduler will start a job based on the schedule created in this example. In this case, the event indicates that a file has arrived before noon.

```sql
SQL> BEGIN
    dbms_scheduler.create_event_schedule(
        SCHEDULE_NAME => 'appowner.file_arrival',
        START_DATE    => systimestamp,
        EVENT_CONDITION => 'tab.user_data.object_owner = 'APPOWNER''
            AND tab.user_data.event_name = 'FILE_ARRIVAL''
            AND extract hour FROM tab.user_data.event_timestamp < 12',
        QUEUE_SPEC    => 'test_events_q');
END;
```

You were introduced to the EVENT_CONDITION and QUEUE_SPEC attributes in the previous example.

### Managing Advanced Scheduler Components

So far, you’ve learned how to manage the basic Scheduler components—jobs, programs, schedules, chains, and events. In this section, let’s look at how to manage the advanced Scheduler components—job classes and windows (and window groups).

You’ll also learn how the Scheduler makes good use of the Database Resource Manager features, such as resource consumer groups and resource plans, to efficiently allocate scarce OS and database resources. Too often, heavy batch jobs run past their window and spill over into the daytime, when OLTP transactions demand the lion’s share of the resources. Prioritizing jobs to ensure that they are guaranteed adequate resources to perform along accepted lines is an essential requirement in production databases. The Scheduler uses the concepts of job classes and windows to prioritize jobs.

### Managing Job Classes

Using job classes helps you prioritize jobs by allocating resources differently among various groups of jobs. The scheduler associates each job class with a resource consumer group, which lets the Scheduler determine the appropriate resource allocation for each job class. The ability to associate job classes with the resource consumer groups created by the Database Resource Manager helps in prioritizing jobs.

---

**Note** All jobs must belong to a job class. There is a default job class, DEFAULT_JOB_CLASS, to which all jobs will belong by default, if they aren’t assigned to any other job class. A job class will be associated with the DEFAULT_CONSUMER_GROUP by default if you don’t expressly assign it to a specific resource consumer group.
Creating a Job Class

All job classes are created in the SYS schema, regardless of which user creates it. The following example uses the CREATE_JOB_CLASS procedure to create a new job class called ADMIN_JOBS.

```sql
BEGIN
    DBMS_SCHEDULER.CREATE_JOB_CLASS(
        JOB_CLASS_NAME => 'admin_jobs',
        RESOURCE_CONSUMER_GROUP => 'admin_group',
        LOGGING_LEVEL => dbms_scheduler.logging_runs,
        LOG_HISTORY => 15);
END;
```

These are the attributes in the preceding example:

- **JOB_CLASS_NAME**: This is the name of the job class.
- **RESOURCE_CONSUMER_GROUP**: This attribute specifies that all jobs that are members of this class will be assigned to the ADMIN_GROUP resource consumer group.
- **LOGGING_LEVEL**: This attribute can take the following three values:
  - **DBMS_SCHEDULER.LOGGING_OFF**: Specifies no logging of any kind for the jobs in the job class
  - **DBMS_SCHEDULER.LOGGING_RUNS**: Specifies detailed log entries for each run of a job
  - **DBMS_SCHEDULER.LOGGING_FULL**: Specifies detailed entries for each run of a job in the job class, as well as for all other operations on the jobs, including the creation, dropping, altering, enabling, or disabling of jobs

**Note** The **DBMS_SCHEDULER.LOGGING_FULL** value for the **LOGGING_LEVEL** attribute provides the most information about jobs in a job class; the default logging level is **DBMS_SCHEDULER.LOGGING_RUNS**.

- **LOG_HISTORY**: This attribute specifies the number of days that the database will retain the logs before purging them using the automatically scheduled **PURGE_LOG** job. You can also manually clear the logs using the **PURGE_LOG** procedure of the DBMS_SCHEDULER package.

The **PURGE_LOG** procedure of the DBMS_SCHEDULER package takes two important parameters—**LOG_HISTORY** and **WHICH_LOG**. You use the **LOG_HISTORY** parameter to specify the number of days to keep logs before the Scheduler purges them. The **WHICH_LOG** parameter enables you to specify whether you want to purge job or window logs. For example, to purge all job logs more than 14 days old, you would use the following statement:

```sql
SQL> EXEC DBMS_SCHEDULER.PURGE_LOG(LOG_HISTORY=14, WHICH_LOG='JOB_LOG');
```

Dropping a Job Class

You drop a job class using the DROP_JOB_CLASS procedure, as shown here:

```sql
BEGIN
    DBMS_SCHEDULER.DROP_JOB_CLASS('TEST_CLASS');
END;
```

**Tip** You must specify the **force=true** option to drop job classes with jobs in them. If the job is already running, it will be allowed to complete before the dropped job class is disabled.
Changing Job Class Attributes

You can change job class attributes with the ALTER_ATTRIBUTES procedure. The following example will change the START_DATE attribute, and its new value is specified by the VALUE parameter:

```sql
SQL> BEGIN
  2  DBMS_SCHEDULER.ALTER_ATTRIBUTES(
  3      NAME        => 'ADMIN_JOBS',
  4      ATTRIBUTE   => 'START_DATE',
  5      VALUE       => '01-JUL-2008 9:00:00 PM US/Pacific');
  6* END;
SQL>
```

Changing Resource Plans Using Windows

A window is an interval with a specific start and end time, such as “from 12 midnight to 6:00 a.m.” However, a window is not merely a chronological device like a schedule, specifying when a job will run; every window is associated with a resource plan. When you create a window, you specify a resource plan as a parameter. This ability to activate different resource plans at different times is what makes windows special scheduling devices that enable you to set priorities.

The basic purpose of a window is to switch the active resource plan during a certain time frame. All jobs that run during a window will be controlled by the resource plan that’s in effect for that window. Without windows, you would have to manually switch the resource manager plans. Windows enable the automatic changing of resource plans based on a schedule.

Note  All windows are created in the SYS schema, no matter which user creates them. To manage windows, you must have the MANAGE SCHEDULER system privilege.

A Scheduler window consists of the following three major attributes:

- **Start date, end date, and repeat interval attributes**: These determine when and how frequently a Window will open and close (thus, these attributes determine when a window is in effect).
- **Duration**: This determines the length of time a window stays open.
- **Resource plan**: This determines the resource priorities among the job classes.

Note  The V$RSRC_PLAN view provides information on currently active resource plans in your database.

On the face of it, both a schedule and a window seem to be serving the same purpose, since both enable you to specify the start and end times and the repeat interval for a job. However, it’s the resource plan attribute that sets a window apart from a simple schedule. Each time a window is open, a specific active resource plan is associated with it. Thus, a given job will be allocated different resources if it runs under different windows.

You can specify what resources you want to allocate to various job classes during a certain time period by associating a resource plan with the window you create for this period. When the window opens, the database automatically switches to the associated resource plan, which becomes the active resource plan. The systemwide resource plan associated with the window will control the resource allocation for all jobs and sessions that are scheduled to run within this window. When the window closes, there will be another switch to the original resource plan that was in effect, provided no other window is in effect at that time.
You can see which window is currently active and which resource plan is associated with that window by using the following query:

```
SQL> SELECT window_name, resource_plan, enabled, active
2  FROM DBA_SCHEDULER_WINDOWS;
WINDOW_NAME           RESOURCE_PLAN          ENABLED       ACTIVE
--------------------  ------------------     --------      -------
TEST_WINDOW           TEST_RESOURCEPLAN       TRUE          FALSE
. . .
SQL>
```

You can see that the window TEST_WINDOW is enabled, but not currently active.

### Creating a Window

Creating a window by using the CREATE_WINDOW procedure. Let’s look at two examples using this procedure, one with an inline specification of the start and end times and the repeat interval, and the other where you use a saved schedule instead to provide these three scheduling attributes.

In the first example, the window-creation statement specifies the schedule for the window:

```
SQL> BEGIN
    2    DBMS_SCHEDULER.CREATE_WINDOW(
    3        WINDOW_NAME       =>  'MY_WINDOW',
    4        START_DATE        =>  '01-JUN-08 12:00:00AM',
    5        REPEAT_INTERVAL   =>  'FREQ=DAILY',
    6        RESOURCE_PLAN     =>  'TEST_RESOURCEPLAN',
    7        DURATION          =>  interval '60' minute,
    8        END_DATE          =>  '31-DEC-08 12:00:00AM',
    9        WINDOW_PRIORITY   =>  'HIGH',
   10        COMMENTS          =>  'Test Window');
    11  END;
```

Let’s look at the individual attributes of the new window created by the preceding statement:

- **RESOURCE_PLAN**: This attribute specifies that while this window is open, resource allocation to all the jobs that run in this window will be guided by the resource plan directives in the TEST_RESOURCEPLAN resource plan.

- **WINDOW_PRIORITY**: This attribute is set to HIGH, and the default priority level is LOW; these are the only two values possible. If two windows overlap, the window with the high priority level has precedence. Since only one window can be open at a given time, when they overlap, the high-priority window will open and the low-priority window doesn’t open.

- **START_DATE**: The setting for this attribute specifies that the window first becomes active at 12:00 a.m. on June 1, 2008. You can also say that the window will open at this time.

- **DURATION**: This attribute is set so that the window will remain open for a period of 60 minutes, after which it will close.

- **REPEAT_INTERVAL**: This attribute specifies the next time the window will open again. In this example, it is 12:00 a.m. on June 2, 2008.

- **END_DATE**: This attribute specifies that this window will open for the last time on December 31, 2008, after which it will be disabled and closed.
Note Since the Scheduler doesn't check to make sure that there are prior windows for any given schedule, windows can overlap sometimes.

The following example creates a window using a saved schedule. Obviously, it is much simpler to create a window this way:

```
SQL> BEGIN
    2    DBMS_SCHEDULER.CREATE_WINDOW(
    3        WINDOW_NAME         =>  'TEST_WINDOW',
    4        SCHEDULE_NAME       =>  'TEST_SCHEDULE',
    5        RESOURCE_PLAN       =>  'TEST_RESOURCEPLAN',
    6        DURATION            =>  interval '180' minute,
    7        COMMENTS            =>  'Test Window');
7
END;
```

In the preceding CREATE_WINDOW procedure, the use of the TEST_SCHEDULE schedule lets you avoid specifying the START_DATE, END_DATE, and REPEAT_INTERVAL parameters.

Tip A window is automatically enabled upon creation.

Once you create a window, you must associate it with a job or job class, so the jobs can take advantage of the automatic switching of the active resource plans.

Managing Windows

You can open, close, alter, enable, disable, or drop a window using the appropriate procedure in the DBMS_SCHEDULER package, and you need the MANAGE SCHEDULER privilege to perform any of these tasks. Note that since all windows are created in the SYS schema, you must always use the [SYS].window_name syntax when you reference any window.

A window will automatically open at a time specified by its START_TIME attribute. You can also open a window manually anytime you wish by using the OPEN_WINDOW procedure. Even when you manually open a window, that window will still open at its regular opening time as specified by its interval.

Here’s an example that shows how you can open a window manually:

```
SQL> EXECUTE DBMS_SCHEDULER.OPEN_WINDOW(
    2        WINDOW_NAME  => 'BACKUP_WINDOW',
    3        DURATION     => '0 12:00:00');
```

Look at the DURATION attribute in the preceding statement. When you specify the duration, you can specify days, hours, minutes, and seconds, in that order. Thus, the setting means 0 days, 12 hours, 0 minutes, and 0 seconds.

You can also open an already open window. If you do this, the window will remain open for the time specified in its DURATION attribute. That is, if you open a window that has been running for 30 minutes, and its duration is 60 minutes, that window will last be open for the initial 30 minutes plus an additional 60 minutes, for a total of 90 minutes.

To close a window, you use the CLOSE_WINDOW procedure, as illustrated by the following example:

```
SQL> EXECUTE DBMS_SCHEDULER.CLOSE_WINDOW('BACKUP_WINDOW');
```
If a job is running when you close a window, the job will continue to run to its completion. However, if you created a job with the `STOP_ON_WINDOW_CLOSE` attribute set to `TRUE`, that running job will close upon the closing of its window.

To disable a window, you use the DISABLE procedure, as shown here:

```
SQL> EXECUTE DBMS_SCHEDULER.DISABLE (NAME => 'BACKUP_WINDOW');
```

You can only disable a window if no job uses that window or if the window isn’t open. If the window is open, you can disable it by using the DISABLE procedure with the `FORCE=TRUE` attribute.

You can drop a window by using the DROP_WINDOW procedure. If a job associated with a window is running, a DROP_WINDOW procedure will continue to run through to completion, and the window is disabled after the job completes. If you set the job’s `STOP_ON_WINDOW_CLOSE` attribute to `TRUE`, however, the job will immediately stop when you drop an associated window. If you use the `FORCE=TRUE` setting, you’ll disable all jobs that use that window.

**Prioritizing Jobs**

You can map each Scheduler job class to a specific resource consumer group. A resource plan is assigned to a resource consumer group, and thus indirectly to each job class as well, by the Database Resource Manager. The active resource plan (as determined by the currently open window) will apportion resources to groups, giving different levels of resources to different jobs, based on their job class.

The Scheduler works closely with the Database Resource Manager to ensure proper resource allocation to the jobs. The Scheduler will start a job only if there are enough resources to run it.

Within each Scheduler window, you can have several jobs running, with varying degrees of priority. You can prioritize jobs at two levels—class and job. The prioritization at the class level is based on the resources allocated to each resource consumer group by the currently active resource plan. For example, the `FINANCE_JOBS` class might rank higher than the `ADMIN_JOBS` class, based on the resource allocations dictated by its active resource plan.

Within the `FINANCE_JOBS` and `ADMIN_JOBS` classes, there will be several individual jobs. Each of these jobs has a job priority, which can range from 1 to 5, with 1 being the highest priority. You can use the `SET_ATTRIBUTES` procedure to change the job priority of any job, as shown here:

```
SQL> BEGIN
  2    dbms_scheduler.SET_ATTRIBUTE(
  3      NAME       =>    'test_job',
  4      ATTRIBUTE  =>    'job_priority',
  5      VALUE      =>     1);
  6  END;
```

The default job priority for a job is 3, which you can verify with the following query:

```
SQL> SELECT job_name, job_priority FROM dba_scheduler_jobs;

JOE_NAME                                JOB_PRIORITY
------------------------------ ---------------------
ADV_SQLACCESS1523128                               3
ADV_SQLACCESS5858921                               3
GATHER_STATS_JOB                                   3
PURGE_LOG                                          3
TEST_JOB03                                         3
TEST_JOB1                                          3
6 rows selected
SQL>
```
When you have more than one job within the same class scheduled for the same time, the job_priority of the individual jobs determines which job starts first.

Window Priorities
Since windows might have overlapping schedules, you may frequently have more than one window open at the same time, each with its own resource plan. At times like this, the Scheduler will close all windows except one, using certain rules of precedence. Here is how the precedence rules work:

- If two windows overlap, the window with the higher priority opens and the window with the lower priority closes.
- If two windows of the same priority overlap, the active window remains open.
- If you are at the end of a window and you have other windows defined for the same time period with the same priority, the window that has the highest percentage of time remaining will open.

Window Groups
A window group is a collection of windows, and it is part of the SYS schema. Window groups are optional entities, and you can make a window a part of a window group when you create it, or you can add windows to a group at a later time. You can specify either a single window or a window group as the schedule for a job.

As explained earlier in this chapter, you can take two or more windows that have similar characteristics—for example, some night windows and a holiday window—and group them together to create a downtime window group. Window groups are used for convenience only, and their use is purely optional.

Managing Scheduler Attributes
In earlier sections in this chapter, you’ve seen how you can use the SET_ATTRIBUTE procedure to modify various components of the Scheduler. Attributes like JOB_NAME and PROGRAM_NAME are unique to the job and program components. You can retrieve the attributes of any Scheduler component with the GET_SCHEDULER_ATTRIBUTE procedure of the DBMS_SCHEDULER package.

Unsetting Component Attributes
You can use the SET_ATTRIBUTE_NULL procedure to set a Scheduler component’s attributes to NULL. For example, to unset the COMMENTS attribute of the TEST_PROGRAM program, you can use the following code:

```sql
SQL> EXECUTE dbms_scheduler.SET_ATTRIBUTE_NULL('TEST_PROGRAM', 'COMMENTS');
```

Altering Common Component Attributes
There are some attributes that are common to all Scheduler components. The SET_SCHEDULER_ATTRIBUTE procedure lets you set these common, or global, attribute values, which affect all Scheduler components. The common attributes include the default time zone, the log history retention period, and the maximum number of job worker processes.

Monitoring Scheduler Jobs
There are several dynamic performance views you can use to monitor Scheduler jobs, and I briefly discuss the important views here.
DBA_SCHEDULER_JOBS

The DBA_SCHEDULER_JOBS view provides the status and general information about scheduled jobs in your database. Here’s a simple query using the view:

```sql
SQL> SELECT job_name, program_name
  2  FROM DBA_SCHEDULER_JOBS;
```

<table>
<thead>
<tr>
<th>JOB_NAME</th>
<th>PROGRAM_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURGE_LOG</td>
<td>PURGE_LOG_PROG</td>
</tr>
<tr>
<td>GATHER_STATS_JOB</td>
<td>GATHER_STATS_PROG</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>

```sql
SQL>
```

DBA_SCHEDULER_RUNNING_JOBS

The DBA_SCHEDULER_RUNNING_JOBS view provides information regarding currently running jobs.

DBA_SCHEDULER_JOB_RUN_DETAILS

You can use the DBA_SCHEDULER_JOB_RUN_DETAILS view to check the status and the duration of execution for all jobs in your database, as the following example shows:

```sql
SQL> SELECT job_name, status, run_duration
  2  FROM DBA_SCHEDULER_JOB_RUN_DETAILS;
```

<table>
<thead>
<tr>
<th>JOB_NAME</th>
<th>STATUS</th>
<th>RUN_DURATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURGE_LOG</td>
<td>SUCCEEDED</td>
<td>+000 00:00:02</td>
</tr>
<tr>
<td>PURGE_LOG</td>
<td>SUCCEEDED</td>
<td>+000 00:00:04</td>
</tr>
<tr>
<td>GATHER_STATS_JOB</td>
<td>SUCCEEDED</td>
<td>+000 00:31:18</td>
</tr>
</tbody>
</table>

```sql
SQL>
```

DBA_SCHEDULER_SCHEDULES

The DBA_SCHEDULER_SCHEDULES view provides information on all current schedules in your database, as shown here:

```sql
SQL> SELECT schedule_name, repeat_interval
  2  FROM dba_scheduler_schedules;
```

<table>
<thead>
<tr>
<th>SCHEDULE_NAME</th>
<th>REPEAT_INTERVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAILY_PURGE_SCHEDULE</td>
<td>freq=daily;byhour=12;byminute=0;bysecond=0</td>
</tr>
</tbody>
</table>

```sql
SQL>
```

DBA_SCHEDULER_JOB_LOG

The DBA_SCHEDULER_JOB_LOG view enables you to audit job-management activities in your database. The data that this view contains depends on how you set the logging parameters for your jobs and job classes.

In the “Creating a Job Class” section, earlier in the chapter, you saw how to set the logging level for a job at the job class level. In order to set the logging levels at the individual job level, you use the
SET_ATTRIBUTE procedure of the DBMS_SCHEDULER package. In the SET_ATTRIBUTE procedure, you can set the LOGGING_LEVEL attribute to two different values:

DBMS_SCHEDULER.LOGGING_FULL
DBMS_SCHEDULER.LOGGING_RUNS

The DBMS_SCHEDULER.LOGGING_RUNS option will merely record the job runs, while the DBMS_SCHEDULER.LOGGING_FULL option turns on full job logging.

Here is an example showing how you can turn on full job logging at the job level:

```sql
SQL> EXECUTE dbms_scheduler.set_attribute ('TESTJOB', 'LOGGING_LEVEL', dbms_scheduler.LOGGING_FULL);
```

**Purging Job Logs**

By default, once a day, the Scheduler will purge all window logs and job logs that are older than 30 days. You can also manually purge the logs by executing the PURGE_LOG procedure, as shown here:

```sql
SQL> EXECUTE DBMS_SCHEDULER.PURGE_LOG(
    LOG_HISTORY   => 1,
    JOB_NAME      => 'TEST_JOB1');
```

**Default Scheduler Jobs**

By default, all Oracle Database 11.1 databases use the Scheduler to run the following jobs, though you can, of course, disable any of these jobs if you wish:

```sql
SQL> SELECT owner, job_name, job_type FROM dba_scheduler_jobs;
```

<table>
<thead>
<tr>
<th>OWNER</th>
<th>JOB_NAME</th>
<th>JOB_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS</td>
<td>ADV_SQLACCESS1821051</td>
<td>PLSQL_BLOCK</td>
</tr>
<tr>
<td>SYS</td>
<td>XMLDB NFS CLEANUP_JOB</td>
<td>STORED_PROCEDURE</td>
</tr>
<tr>
<td>SYS</td>
<td>FGR$AUTOPURGE_JOB</td>
<td>PLSQL_BLOCK</td>
</tr>
<tr>
<td>SYS</td>
<td>BSLN_MAINTAIN_STATS_JOB</td>
<td></td>
</tr>
<tr>
<td>SYS</td>
<td>DRA_REEVALUATE_OPEN_FAILURES</td>
<td>STORED_PROCEDURE</td>
</tr>
<tr>
<td>SYS</td>
<td>HM_CREATE_OFFLINE_DICTIONARY</td>
<td>STORED_PROCEDURE</td>
</tr>
<tr>
<td>SYS</td>
<td>ORA$AUTOTASK_CLEAN</td>
<td></td>
</tr>
<tr>
<td>SYS</td>
<td>PURGE_LOG</td>
<td></td>
</tr>
<tr>
<td>ORACLE_OCM</td>
<td>MGMT_STATS_CONFIG_JOB</td>
<td>STORED_PROCEDURE</td>
</tr>
<tr>
<td>ORACLE_OCM</td>
<td>MGMT_CONFIG_JOB</td>
<td>STORED_PROCEDURE</td>
</tr>
<tr>
<td>EXFSYS</td>
<td>RLM$SCHDNEGACTION</td>
<td>PLSQL_BLOCK</td>
</tr>
<tr>
<td>EXFSYS</td>
<td>RLM$EVTCLEANUP</td>
<td>PLSQL_BLOCK</td>
</tr>
</tbody>
</table>

12 rows selected.

SQL>

The Scheduler is a welcome addition to the Oracle DBA’s arsenal of tools. By providing a sophisticated means of scheduling complex jobs, it does away with the need for third-party tools or complex shell scripts to schedule jobs within the database.

**Automated Maintenance Tasks**

Automated maintenance tasks are jobs that run automatically in the database to perform maintenance operations. Following are the automated maintenance tasks in Oracle Database 11g:
• Automatic Optimizer Statistics Collection
• Automatic Segment Advisor
• Automatic SQL Tuning Advisor

All three automated maintenance tasks run during the default system maintenance window on a nightly basis. I discuss predefined maintenance windows next.

Predefined Maintenance Windows

In Oracle Database 11g, there are seven predefined maintenance windows, as shown here:

MONDAY_WINDOW       Starts and 10 P.M. on Monday and ends at 2 A.M.
TUESDAY_WINDOW      Starts and 10 P.M. on Tuesday and ends at 2 A.M.
WEDNESDAY_WINDOW    Starts and 10 P.M. on Wednesday and ends at 2 A.M.
THURSDAY_WINDOW     Starts and 10 P.M. on Thursday and ends at 2 A.M.
FRIDAY_WINDOW       Starts and 10 P.M. on Friday and ends at 2 A.M.
SATURDAY_WINDOW     Starts at 6 A.M on Saturday and ends at 2 A.M.
SUNDAY_WINDOW       Starts and 6 A.M. on Sunday and ends at 2 A.M.

The weekday maintenance windows are open for 4 hours and the weekend windows for 20 hours. The seven maintenance windows come under the group named MAINTENANCE_WINDOW_GROUP. You can manage the maintenance windows by altering their start and end times. You can also create new maintenance widows and remove or disable the default maintenance windows. I explain these tasks in the following sections.

Managing Automated Maintenance Tasks

Since the database doesn’t assign permanent Scheduler jobs to the three automated maintenance tasks, you can’t manage these tasks with the DBMS_SCHUDULER package. If you want to perform fine-grained management tasks that modify the automated maintenance tasks, you must use the DBMS_AUTO_TASK_ADMIN package.

Monitoring Automated Maintenance Tasks

Query the DBA_AUTOTASK_CLIENT and the DBA_AUTOTASK_OPERATION views to get details about the automated maintenance task execution in the database. The two views share several columns. Here’s a query on the DBA_AUTOTASK_CLIENT view:

```sql
SQL> SELECT client_name, status, 
        2 attributes, window_group, service_name 
        3 FROM dba_autotask_client;
```

<table>
<thead>
<tr>
<th>CLIENT_NAME</th>
<th>STATUS</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto optimizer statistics collection</td>
<td>ENABLED</td>
<td>ON BY DEFAULT, VOLATILE, SAFE TO KILL</td>
</tr>
<tr>
<td>auto space advisor</td>
<td>ENABLED</td>
<td>ON BY DEFAULT, VOLATILE, SAFE TO KILL</td>
</tr>
<tr>
<td>sql tuning advisor</td>
<td>ENABLED</td>
<td>ONCE PER WINDOW, ON BY DEFAULT; VOLATILE, SAFE TO KILL</td>
</tr>
</tbody>
</table>

SQL>

The ATTRIBUTES column shows that all three automated maintenance tasks are enabled by default, as evidenced by the attribute ON BY DEFAULT. When a maintenance window opens, the database automatically creates the three automated maintenance tasks and executes those jobs. However,
only the SQL Tuning Advisor task shows the `ONCE PER WINDOW` attribute. This is because the database executes both the Automatic Optimizer Statistics Collection and the Auto Space Advisor tasks more than once, if the maintenance window is long enough, while it executes the SQL Tuning Advisor just once during any maintenance window.

The database assigns a client name to each of the three automated maintenance tasks, which it deems clients. The Scheduler job associated with the three clients is given an operation name, since the jobs are considered operations. Here are the operation names associated with each of the three automated maintenance tasks:

```
SQL> SELECT client_name, operation_name FROM dba_autotask_operation;
```

<table>
<thead>
<tr>
<th>CLIENT_NAME</th>
<th>OPERATION_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto optimizer stats collection</td>
<td>auto optimizer stats job</td>
</tr>
<tr>
<td>auto space advisor</td>
<td>auto space advisor job</td>
</tr>
<tr>
<td>sql tuning advisor</td>
<td>automatic sql tuning task</td>
</tr>
</tbody>
</table>

SQL>

**Enabling a Maintenance Task**

Execute the `ENABLE` procedure to enable a previously disabled client or operation, as shown here:

```
SQL> begin
  2  dbms_auto_task_admin.enable
  3  (client_name  => 'sql tuning advisor',
  4  operation     => 'automatic sql tuning task',
  5  window_name   => 'monday_window');
  6* end;
SQL> /
```

PL/SQL procedure successfully completed.

SQL>

You can retrieve the `CLIENT_NAME` and the `OPERATION_NAME` attributes by querying the `DBA_AUTOTASK-CLIENT` and the `DBA_AUTOTASK_OPERATION` views.

**Disabling a Maintenance Task**

You can disable any of the three automated maintenance jobs during a specific maintenance window by executing the `DISABLE` procedure.

```
SQL> begin
  2  dbms_auto_task_admin.disable
  3  (client_name  => 'sql tuning advisor',
  4  operation     => 'automatic sql tuning task',
  5  window_name   => 'monday_window');
  6* end;
SQL> /
```

PL/SQL procedure successfully completed.

SQL>

The example shown here disables the SQL Tuning Advisor task only during the `MONDAY_WINDOW` but keeps the task enabled during all other windows.
Implementing Automated Maintenance Tasks

The Autotask Background Process (ABP) is responsible for implementing the three automated maintenance tasks by converting the tasks into Scheduler jobs. For each automated task, ABP creates a task list and assigns a priority. The three priority levels are high, medium, and urgent. The Scheduler also creates job classes and maps a consumer group to the appropriate job class. The ABP assigns jobs to each of the job classes, and the job classes map the jobs to a consumer group based on the job priority level. The MMON background process spawns restarts and monitors the ABP process. The DBA_AUTOTASK view shows the tasks stored in the ABP repository, which is in the SYSAUX tablespace.

You can view the ABP repository by querying the DBA_AUTOTASK_TASK view.

Resource Allocation for Automatic Tasks

The default resource plan assigned to all maintenance windows is the DEFAULT MAINTENANCE PLAN. When a maintenance window opens, the database activates the DEFAULT MAINTENANCE_PLAN to control the CPU resources used by the automated maintenance tasks. The three automated maintenance tasks are run under the ORA$AUTOTASK_SUB_PLAN, a subplan for the DEFAULT MAINTENANCE_PLAN. You can change the resource allocation for automated tasks by changing the resource allocations for this subplan in the resource plan for a specific maintenance window.

Fault Diagnosability

Oracle Database 11g uses a built-in fault diagnosability infrastructure that helps you detect, diagnose, and resolve problems in your database. The fault diagnosability infrastructure focuses on trapping and resolving critical errors such as data corruption and database code bugs. The goal is to proactively detect problems and limit damage to the databases, while reducing the time it takes to diagnose and resolve problems. The fault diagnosability feature also contains elements that make it easier to interact with Oracle Support. Here are the key components of the new fault diagnosability infrastructure:

- **Automatic Diagnostic Repository**: This is a file-based repository for storing database diagnostic data. You can access the ADR through the command line or the Enterprise Manager. It includes trace files, alert logs, and health monitor reports, among other things. Each database instance has its own ADR home directory, but the directory structure is uniform across instances and products, thus enabling Oracle Support to correlate and analyze diagnostic data from multiple products and instances. Immediately after a problem occurs in the database, the diagnostic information is captured and stored within the ADR. You use this diagnostic data to send what are called *incident packages* to Oracle Support.

- **The ADR Command Interpreter (ADRCI)**: This is a command-line tool to manage diagnostic information and create and manage incident reports.

- **Health Monitor**: This tool runs automatic diagnostic checks following database errors. You can also run manual health checks.

- **The Support Workbench**: This is an Enterprise Manager wizard that helps you diagnose critical errors and process and package diagnostic data for transmittal to Oracle Support and filing of technical assistance requests.

- **Incident packaging service**: This is a brand-new tool that enables you to easily create, edit, and modify incident information into physical packages for transmittal to Oracle Support for diagnosing purposes.
• **Data Recovery Advisor**: This tool automatically diagnoses data failures such as missing or corrupt datafiles and reports appropriate repair options. I discuss this in Chapter 16, so I won’t include a section on it in this chapter.

• **SQL Repair Advisor**: This is a new tool that generates a failed SQL statement and recommends a patch to repair it.

• **SQL Test Case Builder**: This tool helps Oracle Support reproduce a failure.

### Automatic Diagnostic Repository

Database instances as well as other Oracle products and components store various types of diagnostic data in the ADR. You can always access the ADR, even when the instance is down, thus leading some to compare the ADR to a plane’s black box, which helps diagnose plane crashes.

### Setting the Automatic Diagnostic Repository Directory

You set the location of the ADR with the initialization parameter `DIAGNOSTIC_DEST`. Setting the `DIAGNOSTIC_DEST` parameter means you don’t have to set the traditional initialization parameters such as `CORE_DUMP_DEST`. If you omit the `DIAGNOSTIC_DEST` parameters, the database assigns the default location of the ADR base directory in the following manner:

- If you’ve set the `ORACLE_BASE` variable, the ADR base will be the same as the directory you assigned for the `ORACLE_BASE` directory.
- If you haven’t set the `ORACLE_BASE` variable, the value of the `DIAGNOSTIC_DEST` parameter defaults to `$ORACLE_HOME/log`.

The `DIAGNOSTIC_DEST` parameter sets the location of the ADR base on a server. An ADR home represents the ADR home directory for an individual database instance. An ADR base may consist of multiple ADR homes, each for a different database instance or Oracle product.

The ADR home for an instance is relative to the ADR base. The following is the general directory structure of the ADR home for an instance, starting from the ADR base:

```
diag/product_type/product_id/instance_id
```

So, if your database has a database name and SID of `orcl1`, and the ADR base is `/u01/app/oracle/`, the ADR home for the database `orcl1` would be

```
/u01/app/oracle/diag/rdbms/orcl1/orcl1
```

### Structure of the ADR

The various subdirectories under the ADR home for an instance store different types of diagnostic data, such as alert logs, Health Monitor reports, incident reports, and trace files for errors. Note that you have two types of alert logs in Oracle Database 11g: a normal text file and an XML-formatted log.

You can query the `V$DIAG_INFO` view to see the different subdirectories of the ADR for an instance:

```
SQL> select * from v$diag_info;
```

```
INST_ID | NAME     | VALUE
--------|----------|--------
1       | Diag Enabled | TRUE
1       | ADR Base   | /u01/app/oracle
1       | Diag Trace | /u01/app/oracle/diag/rdbms/orcl2/orcl2/trace
```
The following directories bear examination:

- **ADR Base** is the directory path for the ADR base.
- **ADR Home** is the ADR home for the instance.
- **Diag Trace** contains the text-based alert log.
- **Diag Alert** contains the XML-formatted alert log.
- **Diag Incident** is the directory for storing incident packages that you create.

**ADRCI**

The new ADRCI is a command-line utility to help you interact with the ADR. You can use ADRCI to view diagnostic data, create incident packages, and view Health Monitor reports.

You invoke ADRCI by simply typing `adrci` at the command line:

```
$ adrci
```

ADRCI: Release 11.1.0.6.0 - Beta on Thu Sep 27 16:59:27 2007
Copyright (c) 1982, 2007, Oracle. All rights reserved.

ADR base = "/u01/app/oracle"

```
adrci>
```

Type the `help` command to view the commands you can use at the ADRCI prompt. When you would like to leave the ADRCI utility, type `exit` or `quit`.

You can also use ADRCI in batch mode, which lets you use ADRCI commands within shell scripts and batch files. You must use the command-line parameters `exec` and `script` to execute ADRCI in batch mode, as shown here:

```
adrci exec 'command [; command]. . . '
adrci script=file_name
```
The ADR Homepath

If you have multiple Oracle instances, all of them will be current when you log into ADRCI. There are some ADRCI commands that work when you have multiple ADR homes current, but others require only a single instance to be current. The default behavior for the ADR homepath is for it to be null when you start up ADRCI. When the ADR home is null, all ADR homes are current. Here’s an example:

```
adrci> show homes
ADR Homes:
diag/rdbms/orcl/orcl
diag/rdbms/orcl2/orcl2
diag/rdbms/eleven/eleven
diag/rdbms/nina/nina

adrci>
```

All ADR homes are always shown relative to the ADR base. Thus, if the ADR base is `/u01/app/oracle` and the database name and SID are both named `orcl1`, the ADR homepath’s complete directory path would be `/u01/app/oracle/diag/rdbms/orcl1/orcl1`.

In the example shown here, the ADR homepath indicates that multiple ADR homes are current. You can set the ADR path to point to a single instance by executing the `SET HOMEPATH` command.

```
adrci> set homepath diag/rdbms/orcl1/orcl1
adrci> show homes
ADR Homes:
diag/rdbms/orcl1/orcl1

adrci>
```

Now, when you issue an `adrci` command, the database will fetch diagnostic data only from the `orcl1` instance.

Viewing the Alert Log

You can view the alert log using the ADRCI utility, as shown here:

```
adrci> show alert –tail
2008-10-17 16:49:50.579000 -
Starting background process FBDA
Starting background process SMCO
... 
Completed: ALTER DATABASE OPEN
adrci>
```

Before you issue this command, make sure you set the homepath for the correct instance. You can return to the ADRCI prompt by pressing Ctrl+C.
Besides ADRCI, there are other ways to view the contents of the alert log. You can view the traditional text-based alert log by going to the directory path listed under the Diag Trace entry in the V$DIAG_INFO view. Of course, you can also view the alert log contents from the database home page in Enterprise Manager. Click Alert Log Contents under Related Links to view the alert log.

### Incident Packaging Service

Oracle bases its diagnostic infrastructure on two key concepts: problems and incidents. A problem is a critical error such as the one accompanied by the Oracle error ORA-4031, which is issued when there isn’t enough shared memory. An incident is a single occurrence of a problem; thus, if a problem occurs multiple times, there will be different incidents to mark the events, each identified by a unique incident ID. When an incident occurs in the database, the database collects diagnostic data for it and attaches an incident ID to the event and stores it a subdirectory in the ADR. An incident is connected to a problem with the help of a problem key. The database creates incidents automatically when a problem occurs, but you can also create your own incidents when you want to report errors that don’t raise and send a critical error alert to Oracle Support.

The ADR uses a flood-controlled incident system, which allows a limited number of incidents for a given problem. This is done to avoid a large number of identical incidents from flooding the ADR with identical information. The database allows each problem to log the diagnostic data for only a certain number of incidents in the ADR. For example, after 25 incidents occur for the same problem during a day, the ADR won’t record any more incidents for the same problem key. The ADR employs two types of retention policies, one governing the retention of the incident metadata and the other governing the retention of incident and dump files. By default, the ADR retains the incident metadata for a month and the incident and dump files for a period of one year.

### Viewing Incidents

You can check the current status of an incident by issuing the `SHOW INCIDENT` command, as shown here:

```
adrci> show incident
```

ADR Home = /u01/app/oracle/diag/rdbms/orcl1/orcl1:
****************************************************************
INCIDENT_ID PROBLEM_KEY CREATE_TIME
------------ ----------- ------------------
17060 ORA 1578 2007-09-25 17:00:18.019731 -04:00
14657 ORA 600 2007-09-09 07:01:21.395179 -04:00
2 rows fetched
```

You can view detailed incident information by issuing the `SHOW INCIDENT ... MODE DETAIL` command as shown here:

```
adrci> show incident -mode DETAIL -p "incident_id=1234"
```

The previous command shows detailed information for the incident with the ID 1234.

An incident package contains all diagnostic data relating to an incident or incidents (it can cover one or more problems). An incident package enables you to easily transmit diagnostic information.
to Oracle Support. You can create incident packages with either the Support Workbench or from the command line using the ADRCI tool. You send an incident package to Oracle Support when you are seeking Oracle's help in resolving problems and incidents. After creating an incident package, you can edit the package by adding and deleting diagnostic files to it.

Creating an Incident Package

The incident packaging service enables you to create an incident package. Using IPS, you gather diagnostic data for an error, such as trace files, dump files, and health-check reports, SQL test cases, and other information, and package this data into a zip file for sending to Oracle Support. IPS tracks and gathers diagnostic information for an incident by using incident numbers. You may add, delete, or scrub the diagnostic files before transmitting the zip file to Oracle Support. Here are the key things you must know about incident packages:

- An incident package is a logical entity that contains problem metadata only. By default, the database includes the first and the last three incidents for a problem in a zip package.
- The zip file that you’ll actually send to Oracle is, of course, a physical package and contains the diagnostic files specified by the metadata in the logical incident package.
- You can send incremental or complete zip files to Oracle Support.

Here are the steps you must follow to create an incident packaging service using IPC commands in the ADRCI:

1. Create a logical package that’ll be used to store incident metadata. You can create an empty or a non-empty logical package. A non-empty package requires an incident number, problem number, or problem key and will automatically contain diagnostic information for the specified incident or problem. In the following example, I create an empty package using the IPS CREATE PACKAGE command:

   ```
adrci> ips create package
   Created package 4 without any contents,
correlation level typical
   ```

   ```
adrci>
   ```

   In order to create a non-empty package, specify the incident number, as shown here:

   ```
adrci> ips create package incident 17060
   Created package 5 based on incident id 17060,
correlation level typical
   ```

   ```
adrci>
   ```

   You may also choose to create a package that spans a time interval:

   ```
adrci> ips create package time '2007-09-20 00:00:00 -12:00' to
          '2007-09-30 00:00:00 -12:00'
   ```
2. If you’ve created an empty logical package in the first step, you must add diagnostic data to it as shown here:

```
adrci> ips add incident 17060 package 4
```

Added incident 17060 to package 4

```
adrci>
```

You can add diagnostic files to the package in the following way:

```
adrci> ips add file <file_name> package <package_number>
```

3. Generate the physical package that you’ll be transmitting to Oracle Support.

```
adrci> ips generate package 4 in /u01/app/oracle/support
```

Generated package 4 in file

```
/u01/app/oracle/diag/IPSPKG_20070929163401_COM_1.zip, mode complete
```

```
adrci>
```

The COM_1 in the filename indicates that it’s a complete file, not incremental, in order to create an incremental physical incident package. Use the following command:

```
adrci> ips generate package 4 in /u01/app/oracle/diag incremental
```

Generated package 4 in file

```
/u01/app/oracle/diag/IPSPKG_20070929163401_INC_2.zip, mode incremental
```

```
adrci>
```

4. Before you can send your incident package to Oracle Support for diagnosis and help, you must formally finalize the incident package, as shown here:

```
adrci> ips finalize package 4
```

Finalized package 4

```
adrci>
```

You can now transmit the finalized zip file to Oracle Support by manually uploading it. In the next section, which discusses the Support Workbench, I’ll show how to automate the transmission of incident packages to Oracle Support.

The Support Workbench

The Support Workbench, which you can access from Enterprise Manager, enables you to automate the management of incidents including the process of filing service requests with Oracle Support and tracing their progress. Besides viewing problems and incidents, you can also generate diagnostic data for a problem as well as run advisors to fix the problem. You can create incident packages easily with the Support Workbench and automatically send them in to Oracle Support.
In order to enable the Support Workbench to upload IPS zip files to Oracle Support, you must install and configure the Oracle Configuration Manager. You can install the Oracle Configuration Manager during the installation of the Oracle software, as shown in Figure 18-7.

![Oracle Universal Installer: Oracle Configuration Manager Registration](image)

**Figure 18-7. Registering the Oracle Configuration Manager**

You can install and configure the Oracle Configuration Manager after the server installation as well, by invoking the Oracle Universal Installer.

The following sections summarize how you can use the Support Workbench to resolve problems.

**Tip** Although the database automatically tracks all critical errors by storing the diagnostic data in the ADR, you can also create a *user-created problem* through the Support Workbench for errors that aren’t automatically tracked by the database as critical errors. To do this, click Create User-Reported Problems under Related Links.

**Viewing Error Alerts**

You can view outstanding problems from the Support Workbench home page. You can check for critical alerts on the Database Home page in the Diagnostic Summary section by clicking the Active Incidents link there or by going to the Critical Alerts section under the Alerts section. To access the Support Workbench, click the Software and Support link and then click the Support Workbench link under the Support section. Figure 18-8 shows the Support Workbench page.

On the Support Workbench home page, select All from the View list to examine all problems.
Examining Problem Details

To view the details of any problem, click View Incident Details on the Incident page.

Collecting Additional Data

Besides the automatic collection of diagnostic data following a critical error in the database, you can also use the Support Workbench to invoke a health check to collect additional diagnostic data. I explain health checks later in this chapter in the section “Running a Health Check.”

Creating Service Requests

From the Support Workbench, you can create service requests with MetaLink. For further reference, you may note down the service request number.

Creating Incident Packages

You can choose either the Quick Packaging method or the Custom Packaging method to create and submit incident packages. The Quick Packaging method is simpler but won’t let you edit or customize the diagnostic data you’re sending to Oracle Support. The Custom Packaging method is more elaborate but enables you customize your incident package.

Following are the steps you must take to create an incident package and send it to Oracle Support using the Custom Packaging method:
1. On the Incident Details page, click the Package link.
2. In the Select Packaging Mode page, select Custom Packaging and click OK.
3. On the Select Package page, select the Create New Package option. Enter the package name and click OK.
4. The Support Workbench takes you to the Customize Package page, confirming that your package was created. Figure 18-9 shows the Customize Package page.

![Customize Package page](image)

**Figure 18-9. The Customize Package page**

5. Once you finish tasks such as editing the package contents or adding diagnostic data, finalize the package by clicking Finish Contents Preparation under the Send to Oracle Support section (in the Packaging Tasks section of the Customize Package page).
6. Generate an upload file by clicking Generate Upload File. Click Immediately or Later followed by Submit to schedule the incident package submission for Oracle Support.
7. Once you submit the package to Oracle Support, IPS processes the zip file and confirms it, before returning you to the Custom Package page. Click Send to Oracle to send the confirmed zip file to Oracle. You must then fill in your MetaLink credentials and choose whether to create a new service request. Click Submit to send the file to Oracle Support.

**Tracking Service Requests**

After submission of an incident package to Oracle Support, you can still add new incidents to the package. You can also let the other DBAs at your organization get the details of the incident package by adding comments to the problem activity log.
Implementing Repairs and Closing Incidents

If the recommendations for repair involve the use of an Oracle advisor, you can make the repair directly from the Support Workbench itself. For example, you can run the Data Recovery Advisor and the SQL Repair Advisor (which I explain later in this chapter in the section “Repairing SQL Statements with the SQL Repair Advisor”) from the Support Workbench.

You can close a resolved incident, or let Oracle purge it; Oracle purges all incidents after 30 days by default.

The Health Monitor

The Health Monitor is a diagnostic framework in the database that runs automatic diagnostic checks when the database encounters critical errors. In addition to these reactive checks, you can also run manual checks whenever you want. You can use either the Enterprise Manager or the DBMS_HM package to run manual health checks. In response to a reactive or a manual check, the database examines components such as memory and transaction integrity and reports back to you.

The following query on the V$HM_CHECK view shows the various types of health checks that can be run:

```
SQL> SELECT name, description FROM v$hm_check;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>HM Test Check</td>
<td>Check for HM Functionality</td>
</tr>
<tr>
<td>DB Structure Integrity Check</td>
<td>Checks integrity of all Database files</td>
</tr>
<tr>
<td>Data Block Integrity Check</td>
<td>Checks integrity of a datafile block</td>
</tr>
<tr>
<td>Redo Integrity Check</td>
<td>Checks integrity of redo log content</td>
</tr>
<tr>
<td>Logical Block Check</td>
<td>Checks logical content of a block</td>
</tr>
<tr>
<td>Transaction Integrity Check</td>
<td>Checks a transaction for corruptions</td>
</tr>
<tr>
<td>Undo Segment Integrity Check</td>
<td>Checks integrity of an undo segment</td>
</tr>
<tr>
<td>All Control Files Check</td>
<td>Checks all control files in the database</td>
</tr>
<tr>
<td>CF Member Check</td>
<td>Checks a multiplexed copy of the control file</td>
</tr>
<tr>
<td>All Datafiles Check</td>
<td>Check for all datafiles in the database</td>
</tr>
<tr>
<td>Single Datafile Check</td>
<td>Checks a datafile</td>
</tr>
<tr>
<td>Log Group Check</td>
<td>Checks all members of a log group</td>
</tr>
<tr>
<td>Log Group Member Check</td>
<td>Checks a particular member of a log group</td>
</tr>
<tr>
<td>Archived Log Check</td>
<td>Checks an archived log</td>
</tr>
<tr>
<td>Redo Revalidation Check</td>
<td>Checks redo log content</td>
</tr>
<tr>
<td>IO Revalidation Check</td>
<td>Checks file accessibility</td>
</tr>
<tr>
<td>Block IO Revalidation Check</td>
<td>Checks file accessibility</td>
</tr>
<tr>
<td>Txn Revalidation Check</td>
<td>Revalidate corrupted txn</td>
</tr>
<tr>
<td>Failure Simulation Check</td>
<td>Creates dummy failures</td>
</tr>
</tbody>
</table>
Dictionary Integrity Check            Checks dictionary
integrity

21 rows selected.

SQL>

You can run all checks except the redo check and the data cross-check when the database is in
the open or mount mode.

Running a Health Check

You can run a health check from the Health Monitor interface in the Enterprise Manager console,
which you can access by clicking the Checkers tab in the Advisor Central page. You can also run a
health check by using the DBMS_HM package. The following example shows how to run a health
check with the RUN_CHECK procedure:

BEGIN
    dbms_hm.run_check (    
        check_name   => 'Transaction Integrity Check',
        run_name     => 'testrun1',
        input_params => 'TXN_ID=9.44.1');
END;
/
PL/SQL procedure successfully completed.

SQL>

The example shown here runs a transaction integrity check for a specified transaction.

Viewing the Results of a Health Check

The Health Monitor stores all its execution results in the ADR. You can query the V$HM_
RECOMMENDATION, V$HM_FINDING, and the V$HM_RUN views to view the recommendations
and findings of a health check. But the easiest way to view a health check result is through the GET_
RUN_REPORT function, as shown in the following example:

SQL> SET LONG 100000

SQL> SELECT dbms_hm.get_run_report('TestCheck1') FROM DUAL;

+--------------------------------------------------+
<table>
<thead>
<tr>
<th>DBMS_HM.GET_RUN_REPORT('TESTCHECK1')</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Run Information</td>
</tr>
<tr>
<td>Run Name : TestCheck1</td>
</tr>
<tr>
<td>Run Id : 42721</td>
</tr>
<tr>
<td>Check Name : Dictionary Integrity Check</td>
</tr>
<tr>
<td>Mode : MANUAL</td>
</tr>
<tr>
<td>Status : COMPLETED</td>
</tr>
<tr>
<td>Start Time : 2008-10-03 16:40:47.464989 -04:00</td>
</tr>
<tr>
<td>End Time : 2008-10-03 16:41:23.068746 -04:00</td>
</tr>
<tr>
<td>Error Encountered : 0</td>
</tr>
<tr>
<td>Source Incident Id : 0</td>
</tr>
<tr>
<td>Number of Incidents Created : 0</td>
</tr>
</tbody>
</table>
+--------------------------------------------------+
Input Parameters for the Run
  TABLE_NAME=ALL_CORE_TABLES
  CHECK_MASK=ALL

Run Findings And Recommendations
  Finding
  Finding Name : Dictionary Inconsistency
  Finding ID   : 42722
  Type          : FAILURE
  Status        : OPEN
  Priority      : CRITICAL
  Message       : SQL dictionary health check:
                  dependency$.dobj# fk 126 on object DEPENDENCY$ failed
  Message       : Damaged rowid is AAAABnAABAAOiHABI –
                  description: No further damage description available

SQL>

You can also use ADRCI to view the results of a Health Monitor check. First, issue the SHOW
HM_RUN command to view the results of a health check:

adrci> SHOW hm_run

****************************************************************
HM RUN RECORD 2131
****************************************************************

RUN_ID                        42721
RUN_NAME                      TestCheck1
CHECK_NAME                    Dictionary Integrity Check
NAME_ID                       24
MODE                          0
START_TIME                    2008-10-03 16:40:47.4649 -04:00
RESUME_TIME                   <NULL>
END_TIME                      2008-10-03 16:41:23.0687 -04:00
MODIFIED_TIME                 2008-10-03 16:41:59.7867 -04:00
TIMEOUT                       0
FLAGS                         0
STATUS                        5
SRC_INCIDENT_ID               0
NUM_INCIDENTS                 0
ERR_NUMBER                    0
REPORT_FILE
   /u01/app/oracle/diag/rdbms/orcl2/orcl2/hm/HMREPORT_TestCheck1
2131 rows fetched

adrci>
In this example, the `SHOW HM_RUN` command shows the report filename under the `REPORT_FILE` column. Once you see the filename, you can view the report itself by issuing the `SHOW REPORT HM_RUN` command, as shown here:

```
adrci> SHOW REPORT hm_run TestCheck1
```

If the `REPORT_FILE` column shows a `NULL` value, you must first generate the report file in the following manner:

```
adrci> CREATE REPORT hm_run TestCheck1
```

Once you generate the report as shown here, you can use the `SHOW REPORT HM_RUN` command to view the report’s contents.

**Repairing SQL Statements with the SQL Repair Advisor**

The SQL Repair Advisor is a new tool that helps you navigate situations where a SQL statement fails with a critical error. For example, if there’s a known bug that’s causing a critical error, you can use the SQL Repair Advisor. Contrary to what its name indicates, the SQL Repair Advisor doesn’t really rewrite or repair the offending SQL statement—it recommends a patch that’ll keep the SQL statement from erroring out. In other words, the advisor provides you a workaround for a problem SQL statement. Note that a SQL patch in this case is very similar to a SQL profile, and adopting the patch results in a change in the query execution plan. You can invoke the SQL Repair Advisor from the Support Workbench or with the help of the DBMS_SQLDIAG package. I explain both methods in the following sections.

**Invoking the SQL Repair Advisor from the Support Workbench**

Follow these steps to invoke the SQL Repair Advisor from the Support Workbench:

1. In the Support Workbench home page, click the ID of the problem you’re trying to fix.
2. In the Problem Details page, click the problem message from the failed SQL statement.
3. In the Investigate and Resolve section on the Self Service tab, click SQL Repair Advisor.
4. Select the schedule for running the advisor and click Submit. Click View in the SQL Repair results page to view the Report Recommendations page.
5. Click Implement if you want the advisor to implement its recommendations. The SQL Repair Advisor presents a confirmation page after it implements the recommendations.

When you migrate to a new release of the database, you can easily drop the patches you applied through the SQL Repair Advisor.

**Invoking the SQL Repair Advisor with the DBMS_SQLDIAG Package**

The following example illustrates how to create a SQL Repair Advisor task and to apply a SQL patch recommended by the advisor to fix a bad SQL statement.
A WORD ABOUT THE FOLLOWING EXAMPLE

I have adapted the following example from the Oracle University course materials. If you try to execute the example as is, you won’t really see any errors. In the Oracle course, an error is induced by using the fix control mechanism, which lets you turn off fixes for optimizer-related bugs. You control the fix control mechanism by setting the undocumented initialization parameter \_FIX\_CONTROL.

You can query the V\$SESSION\_FIX\_CONTROL view to find out for which bugs you can turn the fixes off. You’d use the query SELECT DISTINCT BUGNO FROM V\$SESSION\_FIX\_CONTROL to do this. Once you decide for which bugs you want to turn the fixes off for testing purposes, you can issue the query ALTER SESSION SET "_FIX\_CONTROL"='4728348:OFF'; to turn the bug fix off temporarily while you’re testing the code in our example.

Once you finish testing, don’t forget to execute the statement ALTER SESSION SET "_FIX\_CONTROL"='4728348:ON'; to turn the bug fix on again. As you can tell, this is a somewhat cumbersome procedure, besides requiring you to use an undocumented parameter on your own without Oracle Support helping you along. I’ve mentioned the testing strategy here if you want to test the following code, but it may be smarter not to use any undocumented initialization parameter, due to potential harm to your database.

Let’s say you identify the following SQL statement as the one responsible for a critical error in the database:

SQL> DELETE FROM t t1
   2 WHERE t1.a = 'a'
   3 AND rowid <> (select max(rowid)
   4 FROM t t2 WHERE t1.a= t2.a AND t1.b = t2.b AND t1.d=t2.d);

You can use the SQL Repair Advisor to fix this problem by following these steps:

1. Execute the CREATE\_DIAGNOSTIC\_TASK procedure from the DBMS\_SQLDIAG package to create a SQL Repair Advisor task.

```
SQL> declare
2  report_out clob;
3  task_id  varchar2(50);
4  begin
5  task_id := dbms_sqldiag.create_diagnosis_task(
6    sql_text=>' delete from t t1 where t1.a = ''a''
7     and rowid <> (select max(rowid) from t t2
8     where t1.a= t2.a and t1.b = t2.b
9     and t1.d=t2.d'),
10  task_name =>'test_task1',
11  problem_type=>dbms_sqldiag.problem_type_compilation
12             _error);
13* end;
SQL> /
```

```
PL/SQL procedure successfully completed.
SQL>
```

I chose PROBLEM\_TYPE\_COMPILATION as the value for the PROBLEM\_TYPE parameter in the CREATE\_DIAGNOSTIC\_TASK procedure. You can also choose PROBLEM\_TYPE\_EXECUTION as the value for the PROBLEM\_TYPE parameter.
2. Execute the `SET_TUNING_TASK_PARAMETERS` procedure to supply the task parameters for the new task you created in the previous step.

   SQL> exec dbms_sqltune.set_tuning_task_parameter('task_id, '-SQLDIAG_FINDING_MODE', dbms_sqldiag.SQLDIAG_FINDING_ FILTER_PLANS);

3. Execute the new task, after supplying a task name as the parameter to the `EXECUTE_DIAGNOSTIC_TASK` procedure.

   SQL> exec dbms_sqldiag.execute_diagnosis_task('test_task1');

   PL/SQL procedure successfully completed.

   SQL>

   Note that you only need the `TASK_NAME` parameter to execute the `EXECUTE_DIAGNOSTIC_TASK` procedure.

4. You can get the report of the diagnostic task by executing the `REPORT_DIAGNOSTIC_TASK` function.

   SQL> declare rep_out clob;
   2 begin
   3  rep_out := dbms_sqldiag.report_diagnosis_task
   4     ('test_task1',dbms_sqldiag.type_text);
   5  dbms_output.put_line ('Report : ' || rep_out);
   6*end;
   SQL> /

   Report : GENERAL INFORMATION
   SECTION
   -----------------------------------------------
   Tuning Task Name : test_task1
   Tuning Task Owner : SYS
   Tuning Task ID : 3219
   Workload Type : Single SQL Statement
   Execution Count : 1
   Current Execution : EXEC_3219
   Execution Type : SQL DIAGNOSIS
   Scope : COMPREHENSIVE
   Time Limit(seconds) : 1800
   Completion Status : COMPLETED
   Started at : 10/20/2007 06:33:42
   Completed at : 10/20/2007 06:36:45
   Schema Name : SYS
   SQL ID : 44wx3x03jx01v
   SQL Text : delete from t t1 where t1.a = 'a'
   and rowid <> (select max(rowid)
   from t t2 where t1.a= t2.a
   and t1.b = t2.b and t1.d=t2.d)

   . . .

   PL/SQL procedure successfully completed.

   SQL>
5. You can accept the patch recommended by the SQL Repair Advisor by executing the
   `ACCEPT_SQL_PATCH` procedure.

   ```sql
   SQL> exec dbms_sqldiag.accept_sql_patch (  
           task_name => 'test_task1',  
           task_owner => 'SYS')
   ```

   If you execute the SQL statement after accepting the recommended patch, you'll see a different
   execution plan for the statement. Use the `DBA_SQL_PATCHES` view to find out the names of all
   patch recommendations made by the SQL Repair Advisor. You can drop a patch by executing the
   `DROP_SQL_PATCH` procedure. You can also export a SQL patch to a different database by using a
   staging table.

The SQL Test Case Builder

You can quickly create a test case for submission to Oracle Support by using the new SQL Test Case
Builder. The SQL Test Case Builder provides information about the SQL query, object definitions,
procedures, functions and packages, optimizer statistics, and initialization parameter settings. The
SQL Test Case Builder creates a SQL script that you can run in a different system to re-create the
database objects that exist in the source database. You can invoke the SQL Test Case Builder by
executing the `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE_DIR_BY_INC` function. This function will generate
a SQL Test Case corresponding to the incident ID you pass to the function. You can instead use the
`DBMS_SQLDIAG.EXPORT_SQL_TESTCASE_DIR_BY_TEXT` function to generate a SQL test case that corresponds
to the SQL text you pass as an argument. However, as usual, it’s a lot easier to access the SQL Test
Case Builder through the Enterprise Manager, where you can get to it from the Support Workbench
page by following these steps:

1. Go to the Problem Details page by clicking the relevant problem ID.
2. Click Oracle Support.
3. Click Generate Additional Dumps and Test Cases.
4. Click the icon in the Go to Task column in the Additional Dumps and Test Cases page. This’ll
   start the SQL Test Case Builder analysis for the relevant SQL statement.
PART 7

Performance Tuning
CHAPTER 19

Improving Database Performance: SQL Query Optimization

Performance tuning is the one area in which the Oracle DBA probably spends most of his or her time. If you’re a DBA helping developers to tune their SQL, you can improve performance by suggesting more efficient queries or table- and index-organization schemes. If you’re a production DBA, you’ll be dealing with user perceptions of a slow database, batch jobs taking longer and longer to complete, and so on.

Performance tuning focuses primarily on writing efficient SQL, allocating appropriate computing resources, and analyzing wait events and contention in the system. This chapter focuses on SQL query optimization in Oracle. You’ll learn about the Oracle optimizer and how to collect statistics for it. You’ll find an introduction to the new Automatic Optimizer Statistics Collection feature. You can also manually collect statistics using the DBMS_STATS package, and this chapter shows you how to do that. You’ll learn the important principles that underlie efficient code. I present a detailed discussion of the various tools, such as the EXPLAIN PLAN and SQL Trace utilities, with which you analyze SQL and find ways to improve performance.

Oracle provides several options to aid performance, such as partitioning large tables, using materialized views, storing plan outlines, and many others. This chapter examines how DBAs can use these techniques to aid developers’ efforts to increase the efficiency of their application code. This chapter introduces the SQL Tuning Advisor to help you tune SQL statements. You can then use the recommendations of this advisor to rewrite poorly performing SQL code. I begin the chapter with a discussion of how to approach performance tuning. More than the specific performance improvement techniques you use, your approach to performance tuning determines your success in tuning a recalcitrant application system.

An Approach to Oracle Performance Tuning

Performance tuning is the 800-pound gorilla that is constantly menacing you and that requires every bit of your ingenuity, knowledge, and perseverance to keep out of harm’s way. Your efforts to increase performance or to revive a bogged-down database can have a major impact on your organization, and users and management will monitor and appreciate your results.

Unlike several other features of Oracle database management, performance tuning isn’t a cut-and-dried subject with clear prescriptions and rules for every type of problem you may face. This is one area where your technical knowledge must be used together with constant experimentation and observation. Practice does make you better, if not perfect, in this field.

Frustrating as it is at times, performance tuning is a rewarding part of the Oracle DBA’s tasks. You can automate most of the mundane tasks such as backup, export and import, and data loading—the simple, everyday tasks that can take up so much of your valuable time. Performance tuning is one
area that requires a lot of detective work on the part of application programmers and DBAs to see why some process is running slower than expected, or why you can’t scale your application to a larger number of users without problems.

A Systematic Approach to Performance Tuning

It’s important to follow a systematic approach to tuning database performance. Performance problems commonly come to the fore only after a large number of users start working on a new production database. The system seems fine during development and rigorous testing, but it slows down to a crawl when it goes to production. This could be because the application isn’t easily scalable for a number of reasons.

The seeds of the future performance potential of your database are planted when you design your database. You need to know the nature of the applications the database is going to support. The more you understand your application, the better you can prepare for it by creating your database with the right configuration parameters. If major mistakes were made during the design stage and the database is already built, you’re left with tuning application code on one hand and the database resources such as memory, CPU, and I/O on the other. Oracle suggests a specific design approach with the following steps:

1. Design the application correctly.
2. Tune the application SQL code.
3. Tune memory.
4. Tune I/O.
5. Tune contention and other issues.

Reactive Performance Tuning

Although the preceding performance tuning steps suggest that you can follow the sequence in an orderly fashion, the reality is completely different. Performance tuning is an iterative process, not a sequential one where you start at the top and end up with a fully tuned database as the product. As a DBA, you may be involved in a new project from the outset, when you have just the functional requirements. In this case, you have an opportunity to be involved in the tuning effort from the beginning stages of the application, a phase that is somewhat misleadingly dubbed proactive tuning by some. Alternatively, you may come in after the application has already been designed and implemented, and is in production. In this case, your performance efforts are categorized as reactive performance tuning. What you can do to improve the performance of the database depends on the stage at which you can have input, and on the nature of the application itself.

In general, developers are responsible for writing the proper code, but the DBA has a critical responsibility to ensure that the SQL is optimal. Developers and QA testers may test the application conscientiously, but the application may not scale well when exposed to heavy-duty real-life production conditions. Consequently, DBAs are left scrambling to find solutions to a poorly performing SQL statement after the code is running in production. Reactive performance tuning comprises most of the performance tuning done by most DBAs, for the simple reason that most problems come to light only after real users start using the application.

In many cases, you’re experiencing performance problems on a production instance that was designed and coded long ago. Try to fix the SQL statements first if that’s at all possible. Many people have pointed out that if the application is seriously flawed, you can do little to improve the overall performance of the database, and they’re probably correct. Still, you can make a significant difference in performance, even when the suboptimal code can’t be changed for one reason or another. You can use several techniques to improve performance, even when the code is poorly written but
can’t be changed in the immediate future. The same analysis, more or less, applies to performance-
tuning packaged systems such as PeopleSoft and SAP, where you can’t delve into the code that underlies
the system. You can make use of the SQL Advisor tool’s SQL Profiles to improve performance, even
though you can’t touch the underlying SQL code. SQL tuning, which is the topic of this chapter, is
how you improve the performance in both of the aforementioned situations. In the next chapter,
you’ll learn ways to tune database resources such as memory, disks, and CPU.

**Optimizing Oracle Query Processing**

When a user starts a data-retrieval operation, the user’s SQL statement goes through several sequential
steps that together constitute query processing. One of the great benefits of using the SQL language is
that it isn’t a procedural language in which you have to specify the steps to be followed to achieve the
statement’s goal. In other words, you don’t have to state how to do something; rather, you just state
what you need from the database.

Query processing is the transformation of your SQL statement into an efficient execution plan to
return the requested data from the database. Query optimization is the process of choosing the most
efficient execution plan. The goal is to achieve the result with the least cost in terms of resource usage.
Resources include the I/O and CPU usage on the server where your database is running. This also
means that the goal is to reduce the total execution time of the query, which is simply the sum of the
execution times of all the component operations of the query. This optimization of throughput may
not be the same as minimizing response time. If you want to minimize the time it takes to get the first
n rows of a query instead of the entire output of the query, the optimizer may choose a different plan.
If you choose to minimize the response time for all the query data, you may also choose to parallelize
the operation.

A user’s SQL statement goes through the parsing, optimizing, and execution stages. If the SQL
statement is a query, data has to be retrieved, so there’s an additional fetch stage before the SQL
statement processing is complete. In the next sections you’ll examine what Oracle does during each
of these steps.

**Parsing**

Parsing primarily consists of checking the syntax and semantics of the SQL statements. The end
product of the parse stage of query compilation is the creation of the parse tree, which represents the
query’s structure.

The SQL statement is decomposed into a relational algebra query that’s analyzed to see whether
it’s syntactically correct. The query then undergoes semantic checking. The data dictionary is consulted
to ensure that the tables and the individual columns that are referenced in the query do exist, as well
as all the object privileges. In addition, the column types are checked to ensure that the data matches
the column definitions. The statement is normalized so it can be processed more efficiently. The
query is rejected if it is incorrectly formulated. Once the parse tree passes all the syntactic and semantic
checks, it’s considered a valid parse tree, and it’s sent to the logical query plan generation stage. All
these operations take place in the library cache portion of the SGA.

**Optimization**

During the optimization phase, Oracle uses its optimizer—the Cost-Based Optimizer (CBO)—to
choose the best access method for retrieving data for the tables and indexes referred to in the query.
Using statistics that you provide and any hints specified in the SQL queries, the CBO produces an
optimal execution plan for the SQL statement.
The optimization phase can be divided into two distinct parts: the query rewrite phase and the physical execution plan generation phase. Let’s look at these two optimization phases in detail.

**Query Rewrite Phase**

In this phase, the parse tree is converted into an abstract logical query plan. This is an initial pass at an actual query plan, and it contains only a general algebraic reformulation of the initial query. The various nodes and branches of the parse tree are replaced by operators of relational algebra. Note that the query rewriting here isn’t the same as the query rewriting that’s involved in using materialized views.

**Execution Plan Generation Phase**

During this phase, Oracle transforms the logical query plan into a physical query plan. The optimizer may be faced with a choice of several algorithms to resolve a query. It needs to choose the most efficient algorithm to answer a query, and it needs to determine the most efficient way to implement the operations. In addition to deciding on the best operational steps, the optimizer determines the order in which it will perform these steps. For example, the optimizer may decide that a join between table A and table B is called for. It then needs to decide on the type of join and the order in which it will perform the table join.

The physical query or execution plan takes into account the following factors:

- The various operations (for example, joins) to be performed during the query
- The order in which the operations are performed
- The algorithm to be used for performing each operation
- The best way to retrieve data from disk or memory
- The best way to pass data from one operation to another during the query

The optimizer may generate several valid physical query plans, all of which are potential execution plans. The optimizer then chooses among them by estimating the cost of each possible physical plan based on the table and index statistics available to it, and selecting the plan with the lowest estimated cost. This evaluation of the possible physical query plans is called *cost-based query optimization*. The cost of executing a plan is directly proportional to the amount of resources such as I/O, memory, and CPU necessary to execute the proposed plan. The optimizer passes this low-cost physical query plan to Oracle’s query execution engine. The next section presents a simple example to help you understand the principles of cost-based query optimization.

**A Cost Optimization Example**

Let’s say you want to run the following query, which seeks to find all the supervisors who work in Dallas. The query looks like this:

```
SQL> SELECT * FROM employee e, dept d
WHERE e.dept_no = d.dept_no
AND(e.job = 'SUPERVISOR'
AND d.city = 'DALLAS');
SQL>
```

Now, you have several ways to arrive at the list of the supervisors. Let’s consider three ways to arrive at this list, and compute the cost of accessing the results in each of the three ways.
Make the following simplifying assumptions for the cost computations:

- You can only read and write data one row at a time (in the real world, you do I/O at the block level, not the row level).
- The database writes each intermediate step to disk (again, this may not be the case in the real world).
- No indexes are on the tables.
- The employee table has 2,000 rows.
- The dept table has 40 rows. The number of supervisors is also 40 (one for each department).
- Ten departments are in the city of Dallas.

In the following sections, you’ll see three different queries that retrieve the same data, but that use different access methods. For each query, a crude cost is calculated, so you can compare how the three queries stack up in terms of resource cost. The first query uses a Cartesian join.

**Query 1: A Cartesian Join**

First, form a Cartesian product of the employee and dept tables. Next, see which of the rows in the Cartesian product satisfies the requirement. Here’s the query:

```sql
WHERE e.job=supervisor AND d.dept=operations AND e.dept_no=d.dept_no.
```

The following would be the total cost of performing the query:

- The Cartesian product of employee and dept requires a read of both tables: 2,000 + 40 = 2,040 reads
- Creating the Cartesian product: 2,000 * 40 = 80,000 writes
- Reading the Cartesian product to compare against the select condition: 2,000 * 40 = 80,000 reads
- Total I/O cost: 2,040 + 80,000 + 80,000 = 162,040

**Query 2: A Join of Two Tables**

The second query uses a join of the employee and dept tables. First, join the employee and dept tables on the `dept_no` column. From this join, select all rows where `e.job=supervisor` and `city=Dallas`.

The following would be the total cost of performing the query:

- Joining the employee and dept tables first requires a read of all the rows in both tables: 2,000 + 40 = 2,040
- Creating the join of the employee and dept tables: 2,000 writes
- Reading the join results costs: 2,000 reads
- Total I/O cost: 2,040 + 2,000 + 2,000 = 6,040

**Query 3: A Join of Reduced Relations**

The third query also uses a join of the employee and dept tables, but not all the rows in the two tables—only selected rows from the two tables are joined. Here’s how this query would proceed to retrieve the needed data. First, read the employee table to get all supervisor rows. Next, read the dept table to get all Dallas departments. Finally, join the rows you derived from the employee and the dept tables.
The following would be the total cost of performing the query:

Reading the employee table to get the supervisor rows: 2,000 reads
Writing the supervisor rows derived in the previous step: 40 writes
Reading the dept table to get all Dallas departments: 40 reads
Writing the Dallas department rows derived from the previous step: 10 writes
Joining the supervisor rows and department rows derived in the previous steps of this query execution: A total of \(40 + 10 = 50\) writes
Reading the join result from the previous step: 50 reads

Total I/O cost: \(2,000 + 2(40) + 10 + 2(50) = 2,190\)

This example, simplified as it may be, shows you that Cartesian products are more expensive than more restrictive joins. Even a selective join operation, the results show, is more expensive than a selection operation. Although a join operation is in query 3, it’s a join of two reduced relations; the size of the join is much smaller than the join in query 2. Query optimization often involves early selection (picking only some rows) and projection (picking only some columns) operations to reduce the size of the resulting outputs or row sources.

**Heuristic Strategies for Query Processing**

The use of the cost-based optimization technique isn’t the only way to perform query optimization. A database can also use less systematic techniques, known as *heuristic strategies*, for query processing. A join operation is called a *binary* operation, and an operation such as selection is called a *unary* operation. A successful strategy in general is to perform the unary operation early on, so the more complex and time-consuming binary operations use smaller operands. Performing as many of the unary operations as possible first reduces the row sources of the join operations. Here are some of the common heuristic query-processing strategies:

- Perform selection operations early so you can eliminate a majority of the candidate rows early in the operation. If you leave most rows in until the end, you’re going to do needless comparisons with the rows that you’re going to get rid of later anyway.
- Perform projection operations early so you limit the number of columns you have to deal with.
- If you need to perform consecutive join operations, perform the operation that produces the smaller join first.
- Compute common expressions once and save the results.

**Query Execution**

During the final stage of query processing, the optimized query (the physical query plan that has been selected) is executed. If it’s a SELECT statement, the rows are returned to the user. If it’s an INSERT, UPDATE, or DELETE statement, the rows are modified. The SQL execution engine takes the execution plan provided by the optimization phase and executes it.

Of the three steps involved in SQL statement processing, the optimization process is the crucial one because it determines the all-important question of how fast your data will be retrieved. Understanding how the optimizer works is at the heart of query optimization. It’s important to know what the common access methods, join methods, and join orders are in order to write efficient SQL. The next section presents a detailed discussion of the all-powerful Oracle CBO.
Query Optimization and the Oracle CBO

In most cases, you have multiple ways to execute a SQL query. You can get the same results from doing a full table scan or using an index. You can also retrieve the same data by accessing the tables and indexes in a different order. The job of the optimizer is to find the optimal or best plan to execute your DML statements such as SELECT, INSERT, UPDATE, and DELETE. Oracle uses the CBO to help determine efficient methods to execute queries.

The CBO uses statistics on tables and indexes, the order of tables and columns in the SQL statements, available indexes, and any user-supplied access hints to pick the most efficient way to access them. The most efficient way, according to the CBO, is the least costly access method, cost being defined in terms of the I/O and the CPU expended in retrieving the rows. Accessing the necessary rows means Oracle reads the database blocks on the file system into the buffer pool. The resulting I/O cost is the most expensive part of SQL statement execution because it involves reading from the disk. You can examine these access paths by using tools such as the EXPLAIN PLAN. The following sections cover the tasks you need to perform to ensure that the optimizer functions efficiently.

Choosing Your Optimization Mode

In older versions of Oracle, you had a choice between a rule-based and a Cost-Based Optimizer. In a rule-based approach, Oracle used a heuristic method to select among several alternative access paths with the help of certain rules. All the access paths were assigned a rank, and the path with the lowest rank was chosen. The operations with a lower rank usually executed faster than those with a higher rank. For example, a query that uses the ROWID to search for a row has a cost of 1. This is expected because identifying a row with the help of the ROWID, an Oracle pointer-like mechanism, is the fastest way to locate a row. On the other hand, a query that uses a full table scan has a cost of 19, the highest possible cost under rule-based optimization. The CBO method almost always performs better than the older rule-based approach because, among other things, it takes into account the latest statistics about the database objects.

Providing Statistics to the Optimizer

By default, the database itself automatically collects the necessary optimizer statistics. Every night, the database schedules a statistics collection job during the maintenance window of the Oracle Scheduler. The maintenance window, by default, extends from 10 p.m. to 6 a.m. on weekdays and all weekend as well. The job is named GATHER_STATS_JOB and runs by default in every Oracle Database 11g database. You have the ability to disable the GATHER_STATS_JOB if you wish. You can get details about this default GATHER_STATS_JOB by querying the DBA_SCHEDULER_JOBS view.

The GATHER_STATS_JOB collects statistics for all tables that either don’t have optimizer statistics or have stale (outdated) statistics. Oracle considers an object’s statistics stale if more than 10 percent of its data has changed since the last time it collected statistics for that object. By default, Oracle monitors all DML changes such as inserts, updates, and deletes made to all database objects. You can also view the information about these changes in the DBA_TAB_MODIFICATIONS view. Based on this default object monitoring, Oracle decides whether to collect new statistics for an object.

To check that the GATHER_STATS_JOB is indeed collecting statistics on a regular basis, use the following:

```
SQL> SELECT last_analyzed, table_name, owner, num_rows, sample_size
      2  FROM dba_tables
      3* ORDER by last_analyzed;
```
Note the following points about the preceding output:

- The job collects statistics during the maintenance window of the database, which is, by default, scheduled between 10 p.m. and 6 a.m. during weekdays and all weekend.
- The statistics are collected by the nightly GATHER_STATS_JOB run by the Scheduler.
- If a table is created that day, the job uses all the rows of the table the first time it collects statistics for the table.
- The sampling percentage varies from less than 1 percent to 100 percent.
- The size of the table and the percentage of the sample aren't correlated.
- The job doesn't collect statistics for all the tables each day.
- If a table's data doesn't change after it's created, the job never collects a second time.

Oracle determines the sample size for each object based on its internal algorithms; there is no standard sample size for all objects. Once you verify the collection of statistics, you can pretty much leave statistics collection to the database and focus your attention elsewhere. This way, you can potentially run huge production databases for years on end, without ever having to run a manual statistics collection job using the DBMS_STATS package. Of course, if you load data during the day, or after the GATHER_STATS_JOB starts running, you'll miss the boat and the object won't have any statistics collected for it. Therefore, keep any eye on objects that might undergo huge changes during the day. You might want to schedule a statistics collection job right after the data changes occur.

In addition, you can provide the necessary statistics to the optimizer with the DBMS_STATS package yourself (the automatic statistics collection process managed by the GATHER_STATS_JOB uses the same package internally to collect statistics), which you'll learn about later on in this chapter. The necessary statistics are as follows:

- The number of rows in a table
- The number of rows per database block
- The average row length
- The total number of database blocks in a table
- The number of levels in each index
- The number of leaf blocks in each index
- The number of distinct values in each column of a table
- Data distribution histograms
The key to the CBO’s capability to pick the best possible query plan is its capability to correctly estimate the cost of the individual operations of the query plan. These cost estimates are derived from the knowledge about the I/O, CPU, and memory resources needed to execute each operation based on the table and index statistics. The estimates are also based on the operating system statistics that I enumerated earlier, and additional information regarding the operating system performance.

The database stores the optimizer statistics that it collects in its data dictionary. The DBA_TAB_STATISTICS table shows optimizer statistics for all the tables in your database. You can also see column statistics by querying the DBA_TAB_COL_STATISTICS view, as shown here:

```
SQL> SELECT column_name, num_distinct
    FROM dba_tab_col_statistics
    WHERE table_name='PERSONNEL';
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>NUM_DISTINCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERSON_ID</td>
<td>22058066</td>
</tr>
<tr>
<td>UPDATED_DATE</td>
<td>1200586</td>
</tr>
<tr>
<td>DATE_OF_BIRTH</td>
<td>32185</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>7281</td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td>1729</td>
</tr>
<tr>
<td>GENDER</td>
<td>2</td>
</tr>
<tr>
<td>HANDICAP_FLAG</td>
<td>1</td>
</tr>
<tr>
<td>CREATED_DATE</td>
<td>2480278</td>
</tr>
<tr>
<td>MIDDLE_NAME</td>
<td>44477</td>
</tr>
</tbody>
</table>

As you can see, more than 22 million PERSON_ID numbers are in the PERSONNEL table. However, there are only 7,281 distinct last names and 1,729 distinct first names. Of course, the GENDER column has only two distinct values. The optimizer takes into account these types of information regarding your table data, before deciding on the best plan of execution for a SQL statement that involves the table’s columns.

**Tip** Optimizer statistics include both object (table and index) statistics and system statistics. Without accurate system statistics, the optimizer can’t come up with valid cost estimates to evaluate alternative execution plans.

### Setting the Optimizer Mode

Oracle optimizes the throughput of queries by default. Optimizing throughput means using the fewest resources to process the entire SQL statement. You can also ask Oracle to optimize the response time, which usually means using the fewest resources to get the first (or first $n$) row(s). For batch jobs, response time for individual SQL statements is less important than the total time it takes to complete the entire operation. For interactive applications, response time is more critical.
You can use any of the following three modes for the optimizer with the CBO. The value you set for the `OPTIMIZER_MODE` initialization parameter is the default mode for the Oracle optimizer. The rule-based optimizer is a deprecated product, and I don’t even mention it here.

- **ALL_ROWS**: This is the default optimizer mode, and it directs Oracle to use the CBO whether you have statistics on any of the tables in a query (derived by you through using the DBMS_STATS package or automatically by the Oracle database) or not, with the express goal of maximizing throughput.

**Tip** In the case of all three values for the optimizing mode discussed here, I state that cost optimization is used regardless of whether there are any statistics on the objects that are being accessed in a query. What this means is that in the absence of any statistics collected with the help of the DBMS_STATS package, Oracle uses dynamic sampling techniques to collect the optimizer statistics at run time. For certain types of objects, such as external tables and remote tables, Oracle uses simple default values, instead of dynamic sampling, for the optimizer statistics. For example, Oracle uses a default value of 100 bytes for row length. Similarly, the number of rows in a table is approximated by using the number of storage blocks used by a table and the average row length. However, neither dynamic sampling nor default values give results as good as using comprehensive statistics collected using the DBMS_STATS package. Whether you collect statistics manually, or rely on Oracle’s Automatic Optimizer Statistics Collection feature (which uses the DBMS_STATS package internally), optimizer statistics are collected through the DBMS_STATS package.

- **FIRST_ROWS_n**: This optimizing mode uses cost optimization regardless of the availability of statistics. The goal is the fastest response time for the first $n$ number of rows of output, where $n$ can take the value of 10, 100, or 1000.

- **FIRST_ROWS**: The `FIRST_ROWS` mode uses cost optimization and certain heuristics (rules of thumb), regardless of whether you have statistics or not. You use this option when you want the first few rows to come out quickly so response time can be minimized. Note that the `FIRST_ROWS` mode is retained for backward compatibility purposes only, with the `FIRST_ROWS_n` mode being the latest version of this model.

**Setting the Optimizer Level**

You can set the optimizer mode at the instance, session, or statement level. You set the optimizer mode at the instance level by setting the initialization parameter `OPTIMIZER_MODE` to `ALL_ROWS`, `FIRST_ROWS_n`, or `FIRST_ROWS`, as explained in the previous section. For example, you can set the goal of the query optimizer for the entire instance by adding the following line in your initialization parameter file:

```sql
OPTIMIZER_MODE = ALL_ROWS
```

Setting the initialization parameter `OPTIMIZER_MODE` to `ALL_ROWS` ensures that we can get the complete result set of the query as soon as feasible.

You can also set the optimizer mode for a single session by using the following `ALTER SESSION` statement:

```sql
SQL> ALTER SESSION SET optimizer_mode = first_rows_10;
Session altered.
SQL>
```

The previous statement directs the optimizer to base its decisions on the goal of the best response time for getting the first ten rows of the output of every SQL statement that is executed.
CHAPTER 19  ■  IMPROVING DATABASE PERFORMANCE: SQL QUERY OPTIMIZATION

Tip  Note that the optimizer mode you choose applies only to SQL statements that are issued directly. If you use an ALTER SESSION statement to change the optimizer mode for SQL that's part of a PL/SQL code block, it'll be ignored. You must use optimizer hints, which I discuss in the section titled “Using Hints to Influence the Execution Plan,” to set the optimizer mode for any SQL statement that’s part of a PL/SQL block.

To determine the current optimizer mode for your database, you can run the following query:

```
SQL> SELECT name, value FROM V$PARAMETER
2  WHERE name = 'optimizer_mode';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer_mode</td>
<td>ALL_ROWS</td>
</tr>
</tbody>
</table>

SQL>

Any SQL statement can override the instance- or session-level settings with the use of optimizer hints, which are directives to the optimizer for choosing the optimal access method. By using hints, you can override the instance-wide setting of the OPTIMIZER_MODE initialization parameter. See the section “Using Hints to Influence the Execution Plan” later in this chapter for an explanation of optimizer hints.

What Does the Optimizer Do?

The CBO performs several intricate steps to arrive at the optimal execution plan for a user’s query. The original SQL statement is most likely transformed, and the CBO evaluates alternative access paths (for example, full-table or index-based scans). If table joins are necessary, the optimizer evaluates all possible join methods and join orders. The optimizer evaluates all the possibilities and arrives at the execution plan it deems the cheapest in terms of total cost, which includes both I/O and CPU resource usage cost.

SQL Transformation

Oracle hardly ever executes your query in its original form. If the CBO determines that a different SQL formulation will achieve the same results more efficiently, it transforms the statement before executing it. A good example is where you submit a query with an OR condition, and the CBO transforms it into a statement using UNION or UNION ALL. Or your statement may include an index hint, but the CBO might transform the statement so it can do a full table scan, which can be more efficient under some circumstances. In any case, it’s good to remember that the query a user wishes to be executed may not be executed in the same form by Oracle, but the query’s results are still the same. Here are some common transformations performed by the Oracle CBO:

- Transform IN into OR statements.
- Transform OR into UNION or UNION ALL statements.
- Transform noncorrelated nested selects into more efficient joins.
- Transform outer joins into more efficient inner joins.
- Transform complex subqueries into joins, semijoins, and antijoins.
- Perform star transformation for data warehouse tables based on the star schema.
- Transform BETWEEN to greater than or equal to and less than or equal to statements.
Choosing the Access Path

Oracle can often access the same data through different paths. For each query, the optimizer evaluates all the available paths and picks the least expensive one in terms of resource usage. The following sections present a summary of the common access methods available to the optimizer. If joins are involved, then the join order and the join method are evaluated to finally arrive at the best execution plan. You’ll take a brief look at the steps the optimizer goes through before deciding on its choice of execution path.

Full Table Scans

Oracle scans the entire table during a full table scan. Oracle reads each block in the table sequentially, so the full table scan can be efficient if the database uses a high default value internally for the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter. The parameter determines the maximum number of blocks the database reads during a sequential scan. However, for large tables, full table scans are inefficient in general.

Table Access by ROWID

Accessing a table by ROWID retrieves rows using unique ROWIDs. ROWIDs in Oracle specify the exact location in the datafile and the data block where the row resides, so ROWID access is the fastest way to retrieve a row in Oracle. Often, Oracle obtains the ROWID through an index scan of the table’s indexes. Using these ROWIDs, Oracle swiftly fetches the rows.

Index Scans

An index stores two things: the column value of the column on which the index is based and the ROWID of the rows in the table that contain that column value. An index scan retrieves data from an index using the values of the index columns. If the query requests only the indexed column values, Oracle will return those values. If the query requests other columns outside the indexed column, Oracle will use the ROWIDs to get the rows of the table.

Choosing the Join Method

When you need to access data that’s in two or more tables, Oracle joins the tables based on a common column. However, there are several ways to join the row sets returned from the execution plan steps. For each statement, Oracle evaluates the best join method based on the statistics and the type of unique or primary keys on the tables. After Oracle has evaluated the join methods, the CBO picks the join method with the least cost.

The following are the common join methods used by the CBO:

- **Nested loop join**: A nested loop join involves the designation of one table as the *driving table* (also called the *outer table*) in the join loop. The other table in the join is called the *inner table*. Oracle fetches all the rows of the inner table for every row in the driving table.
- **Hash join**: When you join two tables, Oracle uses the smaller table to build a hash table on the join key. Oracle then searches the larger table and returns the joined rows from the hash table.
- **Sort-merge join**: The sort join operation sorts the inputs on the join key, and the merge join operation merges the sorted lists. If the input is already sorted by the join column, there’s no need for a sort join operation for that row source.
Choosing the Join Order

Once the optimizer chooses the join method, it determines the order in which the tables are joined. The goal of the optimizer is always to join tables in such a way that the driving table eliminates the largest number of rows. A query with four tables has a maximum of 4 factorial, or 24, possible ways in which the tables can be joined. Each such join order would lead to a number of different execution plans, based on the available indexes and the access methods. The search for an optimal join strategy could take a long time in a query with a large number of tables, so Oracle depends on an adaptive search strategy to limit the time it takes to find the best execution plan. An adaptive search strategy means that the time taken for optimization is always a small percentage of the total time that is taken for execution of the query itself.

Drawbacks of the CBO

The CBO is systematic, but the optimizer is not guaranteed to follow the same plan in similar cases. However, the CBO isn’t always perfect, and you need to watch out for the following:

- The CBO isn’t fixed across Oracle versions. Execution plans can change over time as versions change. Later in this chapter, you’ll see how to use stored outlines so the optimizer always uses a known plan to maintain plan stability.

- Application developers may know more than the CBO when it comes to choosing the best access path. Application developers know the needs of the users, of which the CBO is completely unaware. This could lead to a situation where the CBO may be optimizing throughput, when the users would rather have a quick set of results on their screen. By using hints such as FIRST_ROWS_n, you can overcome this drawback in the CBO.

- The CBO depends enormously on correct statistics gathering. If the statistics are absent or outdated, the optimizer can make poor decisions.

Providing Statistics to the CBO

The CBO can follow optimal execution paths only if it has detailed knowledge of the database objects. Starting with Oracle Database 10g, the recommended way to provide these statistics is by letting the database automatically collect statistics for you. This is known as the Automatic Optimizer Statistics Collection feature, which I explained in Chapter 17. You can also manually provide statistics to the optimizer with the DBMS_STATS package. Note that whether you rely on automatic collection of statistics or collect them yourself manually, Oracle uses the DBMS_STATS package to collect statistics.

Using DBMS_STATS to Collect Statistics

Although letting the database automatically collect optimizer statistics is the recommended approach, you can still manually collect optimizer statistics using the DBMS_STATS package.

**Tip** For large tables, Oracle recommends just sampling the data, rather than looking at all of it. Oracle lets you specify row or block sampling, and it sometimes seems to recommend sampling sizes as low as 5 percent. The default sampling size for an estimate is low too. Oracle also recommends using the DBMS_STATS automatic sampling procedure. However, statistics gathered with sampled data aren’t reliable. The difference between collecting optimizer statistics with the estimate at 30 percent and 50 percent is startling at times in terms of performance. Always choose the option of collecting full statistics for all your objects, even if the frequency is not as high as it could be if you just sampled the data.
As I explained in Chapter 17, you must manually collect optimizer statistics under the following conditions:

- When you use external tables
- When you need to collect system statistics
- To collect statistics on fixed objects, such as the dynamic performance tables (For dynamic tables, you should use the GATHER_FIXED_OBJECTS_STATS procedure to collect optimizer statistics.)
- Immediately after you run a bulk load job, because this makes your automatically collected statistics unrepresentative

The following sections show you how to make use of the DBMS_STATS package to gather statistics.

---

**Note** Oracle recommends that you not use the older ANALYZE statement to collect statistics for the optimizer, but rather use the DBMS_STATS package. The ANALYZE command is retained for backward compatibility, and you must use it for non-optimizer statistics collection tasks, such as verifying the validity of an object (using the VALIDATE clause), or identifying migrated and chained rows in a table (using the LIST CHAINED ROWS clause).

---

**Storing the Optimizer Statistics**

You use various DBMS_STATS package procedures to collect optimizer statistics. Most of these procedures have three common attributes—STATOWN, STATTAB, and STATID—which enable you to save the collected statistics in a database table owned by a user. By default, these attributes are null, and you shouldn’t provide a value for any of these attributes if your goal is to collect statistics for the optimizer. When you ignore these attributes, optimizer statistics you collect are stored in the data dictionary tables by default, where they’re accessible to the Oracle optimizer.

**Collecting the Statistics**

The DBMS_STATS package has several procedures that let you collect data at different levels. The main data collection procedures for database table and index data are as follows:

- **GATHER_DATABASE_STATISTICS** gathers statistics for all objects in the database.
- **GATHER_SCHEMA_STATISTICS** gathers statistics for an entire schema.
- **GATHER_TABLE_STATISTICS** gathers statistics for a table and its indexes.
- **GATHER_INDEX_STATISTICS** gathers statistics for an index.

Let’s use the DBMS_STATS package to collect statistics first for a schema, and then for an individual table.

- Collecting statistics at the schema level:

  ```sql
  SQL> EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS (ownname => 'hr');
  PL/SQL procedure successfully completed.
  SQL>
  ```

- Collecting statistics at the table level:

  ```sql
  SQL> EXECUTE DBMS_STATS.GATHER_TABLE_STATS ('hr','employees');
  PL/SQL procedure successfully completed.
  SQL>
  ```
The GATHER_DATABASE_STATISTICS procedure collects optimizer statistics for the entire database. This is probably the most common way of using the DBMS_STATS package, as you can use this procedure to collect statistics for all database objects with a single statement. Here’s an example:

```sql
SQL> EXECUTE dbms_stats.gather_database_stats (-
    > ESTIMATE_PERCENT => NULL, -
    > METHOD_OPT => 'FOR ALL COLUMNS SIZE AUTO', -
    > GRANULARITY => 'ALL', -
    > CASCADE => 'TRUE',- 
    > OPTIONS => 'GATHER AUTO');

PL/SQL procedure successfully completed.
SQL>
```

**Tip** Although you can use the ESTIMATE_PERCENT attribute to collect optimizer statistics for a sample ranging from 0.000001 to 100 percent of the rows in a table, you should strive to collect statistics for all the rows (by using null as the value for this attribute). Collecting statistics based on a sample is fraught with dangers. Unless the tables are so huge that you can’t collect all statistics within your maintenance window, strive to collect full statistics on all objects, especially those that have heavy DML changes.

Let me explain the preceding GATHER_DATABASE_STATS procedure briefly here:

- The example shows only some of the various attributes or parameters that you can specify. You can see the complete list of attributes by typing in this command:

  ```sql
  SQL> DESCRIBE DBMS_STATS.GATHER_DATABASE_STATS
  ```

- If you don’t specify any of the attributes, Oracle uses the default values for those attributes. Even when I use a default value, I list the attribute here, for exposition purposes.

- The ESTIMATE_PERCENT attribute refers to the percentage of rows that should be used to estimate the statistics. I chose null as the value. Null here, contrary to intuition, means that Oracle collects statistics based on all rows in a table. This is the same as using the COMPUTE STATISTICS option in the traditional ANALYZE command. The default for this attribute is to let Oracle estimate the sample size for each object, using the DBMS_STATS.AUTO_SAMPLE_SIZE procedure.

- You can use the METHOD_OPT attribute to specify several things, including whether histograms should be collected. Here, I chose FOR ALL COLUMNS SIZE AUTO, which is the default value for this attribute.

- The GRANULARITY attribute applies only to tables. The ALL value collects statistics for subpartitions, partitions, and global statistics for all tables.

- The CASCADE=YES option specifies that statistics be gathered on all indexes, along with the table statistics.

- The OPTIONS attribute is critical. The most important values for this attribute are as follows:
  - GATHER gathers statistics for all objects, regardless of whether they have stale or fresh statistics.
  - GATHER AUTO collects statistics for only those objects that Oracle deems necessary.
  - GATHER EMPTY collects statistics only for objects without statistics.
  - GATHER STALE results in collection of statistics for only stale objects, the determination as to the object staleness being made by checking the DBA_TAB_MODIFICATIONS view.
Note that you could also execute the GATHER_DATABASE_STATS procedure in the following format, which produces equivalent results:

```sql
SQL> BEGIN
    dbms_stats.gather_database_stats (ESTIMATE_PERCENT => NULL, METHOD_OPT =>
        'FOR ALL COLUMNS SIZE AUTO',
        GRANULARITY => 'ALL', CASCADE => 'TRUE', OPTIONS => 'GATHER AUTO');
END;
PL/SQL procedure successfully completed.
SQL>
```

You can check when a table has last been analyzed by using the following query:

```sql
SQL> SELECT table_name, last_analyzed FROM dba_tables;
```

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>LAST_ANALYZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST1</td>
<td>07/08/2008</td>
</tr>
<tr>
<td>TEST2</td>
<td>07/08/2008</td>
</tr>
<tr>
<td>TEST3</td>
<td>07/08/2008</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>

SQL>

You can use a similar query for indexes, using the DBA_INDEXES view.

---

**Tip** Make sure you have the initialization parameter JOB_QUEUE_PROCESSES set to a positive number. If this parameter isn't set, it takes the default value of 0, and your DBMS_STATS.GATHER_SYSTEM_STATS procedure won’t work. You can do this dynamically, for example, issue the command `ALTER SYSTEM SET JOB_QUEUE_PROCESSES = 20`.

---

**Deferred Statistics Publishing**

By default, the database publishes the statistics it collects for immediate use by the optimizer. However, there may be times when you don’t want this to happen. Instead, you may wish to first test the statistics and release them for public use only if you’re satisfied with them. Oracle lets you save new statistics collected by the database as pending statistics, which you can publish or not ultimately, based on your testing of the statistics. Current or published statistics are meant to be used by the optimizer, and pending or deferred statistics are private statistics, which are kept from the optimizer.

**Determining and Setting the Status of the Statistics**

Execute the DBMS_STATS.GET_PREFS procedure to determine the publishing status of statistics in your database:

```sql
SQL> select dbms_stats.get_prefs('PUBLISH') publish from dual;
```

<table>
<thead>
<tr>
<th>PUBLISH</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
</tr>
</tbody>
</table>

SQL>
The value `TRUE` indicates that the database automatically publishes all statistics after it collects them. This is the default behavior of the database. If the query had returned the value `FALSE`, it means that the database will keep new statistics pending until you decide to formally publish them. You can also execute the `GET_PREFS` function to find out the publishing mode for a single table:

```sql
SQL> SELECT dbms_stats.get_prefs('PUBLISH','stats','test_table')
       FROM dual;
```

You can change the publishing settings for objects at the database or at the object (table) level by executing the `SET_TABLE_PREFS` function. For example, to keep the database from automatically publishing the statistics it collects for the table `EMPLOYEES`, execute this function:

```sql
SQL> exec dbms_stats.set_table_prefs ('HR','EMPLOYEES',
       'PUBLISH','FALSE');
```

The database stores pending statistics in the `DBA_TAB_PENDING_STATS` view and it stores the published statistics in the `DBA_TAB_STATS` view.

### Making Pending Statistics Public

You can test any pending statistics in your database to see how they affect performance. If they help performance, you can publish them for use by the optimizer; otherwise, just drop the statistics. You publish the pending statistics, that is, make them available to the optimizer for testing purposes, by setting the initialization parameter `OPTIMIZER_USE_PENDING_STATISTICS`. By default, this parameter is set to `FALSE`, which means the optimizer will bypass the pending statistics, as shown here:

```sql
SQL> show parameter optimizer_use_pending_statistics
NAME                                TYPE         VALUE
--------------------------------    -------      ------
optimizer_use_pending_statistics    boolean      FALSE
SQL>
```

You can make the optimizer take the pending statistics into account by setting the `OPTIMIZER_USE_PENDING_STATISTICS` parameter to `TRUE`, as shown here:

```sql
SQL> ALTER SESSION SET optimizer_use_pending_statistics=TRUE ;
```

The optimizer will use the pending statistics once you run the previous statement. Once your tests confirm that the new statistics are OK, you can make the pending statistics public by executing the `PUBLISH_PENDING_STATS` procedure:

```sql
SQL> EXEC dbms_stats.publish_pending_stats (NULL,NULL);
```

If you want to publish statistics for a single table, you can do so as well:

```sql
SQL> EXEC dbms_stats.publish_pending_stats('HR','EMPLOYEES');
```

If you conclude, on the other hand, that the pending statistics aren’t helpful, delete them by executing the `DELETE_PENDING_STATS` procedure:

```sql
SQL> EXEC dbms_stats.delete_pending_stats ('HR','EMPLOYEES');
```

You can also test the pending statistics you collect in one database in another database, by using the `EXPORT_PENDING_STATS` procedure:

```sql
SQL> EXEC dbms_stats.export_pending_stats ('HR', 'EMPLOYEES');
```
Extended Statistics

The statistics that the database collects are sometimes unrepresentative of the true data. Oracle provides the capability for collecting extended statistics under some circumstances to mitigate the problems in statistics collection. Extended statistics include the collection of multicolumn statistics for groups of columns and expression statistics that collect statistics on function-based columns. I explain both types of extended optimizer statistics in the following sections.

Multicolumn Statistics

When Oracle collects statistics on a table, it estimates the selectivity of each column separately, even when two or more columns may be closely related. Oracle assumes that the selectivity estimates of the individual columns are independent of each other and simply multiplies the independent predicates’ selectivity to figure out selectivity of the group of predicates. This approach leads to an underestimation of the true selectivity of the group of columns. You can collect statistics for a group of columns to avoid this underestimation.

I use a simple example to show why collecting statistics for column groups instead of individual columns is a good idea when the columns are related. In the SH.CUSTOMERS table, the CUST_STATE_PROVINCE and the COUNTRY_ID columns are correlated, with the former column determining the value of the latter column. Here’s a query that shows the relationship between the two columns:

```
SQL> SELECT count(*)
FROM sh.customers
WHERE cust_state_province = 'CA';
COUNT(*)
-----------
3341
```

The previous query uses only a single column, CUST_STATE_PROVINCE, to get a count of the number of customers from the province named “CA.” The following query also involves the COUNTRY_ID column, but returns the same count, 3341.

```
SQL> SELECT count(*)
FROM customers
WHERE cust_state_province = 'CA'
AND country_id=52790;
COUNT(*)
-----------
3341
```

Obviously, the same query with a different value for the COUNTRY_ID column will return a different count (most likely zero, since CA stands for California and it’s unlikely that a city of the same name is present in other countries). You can collect statistics on a set of related columns such as CUST_STATE_PROVINCE and COUNTRY_ID by estimating the combined selectivity of the two columns. The database can collect statistics for column groups based on the database workload, but you create column groups by using the DBMS_STATS.CREATE_EXTENDED_STATS function, as I explain next.
Creating Column Groups

You create a column group by executing the `CREATE_EXTENDED_STATS` function, as shown in this example:

```sql
declare
cg_name varchar2(30);
begin
cg_name := dbms_stats.create_extended_stats(null,'customers',
'cust_state_province,country_id');
end;
/
```

Once you create a column group as shown here, the database will automatically collect statistics for the column group instead of the two columns as individual entities. The following query verifies the successful creation of the new column group:

```sql
SQL> SELECT extension_name, extension
FROM dba_stat_extensions
WHERE table_name='CUSTOMERS';
```

You can drop a column group by executing the `DROP_EXTENDED_STATS` function:

```sql
SQL> exec dbms_stats.drop_extended_stats('sh','customers',
'(cust_state_province, country_id)');
```

Collecting Statistics for Column Groups

You can execute the `GATHER_TABLE_STATS` procedure with the `METHOD_OPT` argument set to the value for all columns... to collect statistics for column groups. By adding the `FOR COLUMNS` clause, you can have the database create the new column group as well as collect statistics for it, all in one step, as shown here:

```sql
SQL> exec dbms_Stats.gather_table_stats(
  ownname=>null,-
  tabname=>'customers',-
  method_opt=>'for all columns size skewonly,-
  for columns (cust_state_province,country_id) size skewonly');
```

Expression Statistics

If you apply a function to a column, the column value changes. For example, the `LOWER` function in the following example returns a lowercase string:

```sql
SQL> SELECT count(*)
FROM customers
WHERE LOWER(cust_state_province)= 'ca';
```

Although the `LOWER` function transforms the values of the `CUST_STATE_PROVINCE` column by making them lowercase, the optimizer has only the original column estimates and not the changed
columns estimates. So, the optimizer really doesn’t have an accurate idea about the true selectivity of the transformed values of the column. You can collect expression statistics on some types of column expressions, in those cases where the function preserves the original data distribution characteristics of the original column. This is true when you apply a function such as TO_NUMBER to a column. You can use function-based expressions for user-defined functions as well as function-based indexes.

The expression statistics feature relies on Oracle’s virtual column capabilities. You execute the CREATE_EXTENDED_STATS function to create statistics on column expressions, as shown here:

```sql
SQL> SELECT
    dbms_stats.create_extended_stats(null,'customers',
       '(lower(cust_state_province))')
FROM dual;
```

Alternatively, you can execute the GATHER_TABLE_STATS function to create expression statistics:

```sql
SQL> exec dbms_stats.gather_table_stats(null,'customers',
   method_opt=>'for all columns size skewonly,
   for columns (lower(cust_state_province)) size skewonly');
```

As with the column group statistics, you can query the DBA_STAT_EXTENSIONS view to find out details about expression statistics.

**The Cost Model of the Oracle Optimizer**

The cost model of the optimizer takes into account both I/O cost and CPU cost, both in units of time. The CBO evaluates alternative query costs by comparing the total time it takes to perform all the I/O operations, as well as the number of CPU cycles necessary for the query execution. The CBO takes the total number of I/Os and CPU cycles that will be necessary according to its estimates, and converts them into execution time. It then compares the execution time of the alternative execution paths and chooses the best candidate for execution.

For the CBO to compute the cost of alternative paths accurately, it must have access to accurate system statistics. These statistics, which include items such as I/O seek time, I/O transfer time, and CPU speed, tell the optimizer how fast the system I/O and CPU perform. It’s the DBA’s job to provide these statistics to the optimizer. I show how to collect system statistics in the following section.

**Collecting System Statistics**

Although Oracle can automatically collect optimizer statistics for you regarding your tables and indexes, you need to collect operating system statistics with the GATHER_SYSTEM_STATS procedure. When you do this, Oracle populates the SYS.AUX_STATS$ table with various operating system statistics, such as CPU and I/O performance. Gathering system statistics at regular intervals is critical, because the Oracle CBO uses these statistics as the basis of its cost computations for various queries. System statistics enable the optimizer to compare more accurately the I/O and CPU costs of alternative execution. The optimizer is also able to figure out the execution time of a query more accurately if you provide it with accurate system statistics.

You can run the GATHER_SYSTEM_STATS procedure in different modes by passing values to the GATHERING_MODE parameter. There’s a no-workload mode you can specify to quickly capture the I/O system characteristics. You can also specify a workload-specific mode by using the INTERVAL, START, and STOP values for the GATHERING_MODE parameter. Here’s a brief explanation of the different values you can specify for the GATHERING_MODE parameter:
• **No-workload mode**: By using the NOWORKLOAD keyword, you can collect certain system statistics that mostly pertain to general I/O characteristics of your system, such as I/O seek time (IOSEEKTIM) and I/O transfer speed (IOTFRSPEED). You should ideally run the GATHER_SYSTEM_STATS procedure in no-workload mode right after you create a new database. The procedure takes only a few minutes to complete and is suitable for all types of workloads.

---

**Note** If you collect both workload and no-workload statistics, the optimizer will use the workload statistics.

---

• **Workload mode**: To collect representative statistics such as CPU and I/O performance, you must collect system statistics during a specified interval that represents a typical workload for your instance. You can use the INTERVAL keyword to specify statistics collection for a certain interval of time. You can alternatively use the START and STOP keywords to collect system statistics for a certain length of time. Under both workload settings for the GATHERING_MODE parameter (INTERVAL, or START and STOP), the database collects the following statistics: MAXTHR, SLAVETHR, CPUSPEED, SREADTIM, MREADTIM, and MBRC.

Here’s what the various system statistics I mentioned stand for:

- **IOTFRSPEED**: I/O transfer speed (bytes per millisecond)
- **IOSEEKTIM**: Seek time + latency time + operating system overhead time (milliseconds)
- **SREADTIM**: Average time to (randomly) read a single block (milliseconds)
- **MREADTIM**: Average time to (sequentially) read an MBRC block at once (milliseconds)
- **CPUSPEED**: Average number of CPU cycles captured for the workload (statistics collected using the INTERVAL or START and STOP options)
- **CPUSPEEDNW**: Average number of CPU cycles captured for the no-workload mode (statistics collected using NOWORKLOAD option)
- **MBR**: Average multiblock read count for sequential read, in blocks
- **MAXTHR**: Maximum I/O system throughput (bytes/second)
- **SLAVETHR**: Average slave I/O throughput (bytes/second)

Here’s the structure of the GATHER_SYSTEM_STATS procedure:

```sql
DBMS_STATS.GATHER_SYSTEM_STATS (    gathering_mode VARCHAR2 DEFAULT 'NOWORKLOAD',    interval INTEGER DEFAULT NULL,    stattab VARCHAR2 DEFAULT NULL,    statid VARCHAR2 DEFAULT NULL,    statown VARCHAR2 DEFAULT NULL);
```

Here’s an example that shows how to use the procedure to collect system statistics:

```sql
SQL> EXECUTE dbms_stats.gather_system_stats('start'); PL/SQL procedure successfully completed.
SQL> EXECUTE dbms_stats.gather_system_stats('stop'); PL/SQL procedure successfully completed.
SQL> SELECT * FROM sys.aux_stats$;
```
### Collecting Statistics on Dictionary Objects

You should collect optimizer statistics on data dictionary tables to maximize performance. The two types of dictionary tables are **fixed** and **real**. You can’t change or delete dynamic performance tables, which means they are fixed. Real dictionary tables belong to schemas such as sys and system.

#### Collecting Statistics for Fixed Objects

Oracle recommends that you gather statistics for dynamic performance tables (fixed objects) only once for every database workload, which is usually a week for most OLTP databases. You can collect fixed object statistics in a couple ways, as follows:

- You can use the `DBMS_STATS.GATHER_DATABASE_STATS` procedure and set the `GATHER_SYS` argument to `TRUE` (the default is `FALSE`).
- You can use the `GATHER_FIXED_OBJECTS_STATS` procedure of the `DBMS_STATS` package, as shown here:

  ```sql
  SQL> SHOW USER
  USER is "SYS"
  SQL> EXECUTE DBMS_STATS.GATHER_FIXED_OBJECTS_STATS;
  ```

#### Tip
Before you can analyze any dictionary objects or fixed objects, you need the SYSDBA or ANALYZE ANY DICTIONARY system privilege.

You can use the procedures from the `DBMS_STATS` package that enable table-level statistics collection to collect statistics for an individual fixed table.
Collecting Statistics for Real Dictionary Tables

You can use the following methods to collect statistics for real dictionary tables:

- Set the `GATHER_SYS` argument of the `DBMS_STATS.GATHER_DATABASE_STATS` procedure to `TRUE`. You can also use the `GATHER_SCHEMA_STATS` (`'SYS'`) option.
- Use the `DBMS_STATS.GATHER_DICTIONARY_STATS` procedure, as shown here:

  ```sql
  SQL> SHO user
  USER is "SYS"
  SQL> EXECUTE dbms_stats.gather_dictionary_stats;
  ```

  The `GATHER_DICTIONARY_STATS` procedure helps you collect statistics for tables owned by the SYS and SYSTEM users as well as the owners of all database components.

**Note** You can also use the `DBMS_STATS` package to delete, import, restore, and set optimizer statistics that you have previously collected.

Frequency of Statistics Collection

Theoretically, if your data is static, you may only need to collect statistics once. If your database performs only a small amount of DML activities, you may collect statistics at relatively longer intervals, say weekly or monthly. However, if your database objects go through changes on a daily basis, you need to schedule the statistics collection jobs much more frequently, say daily or even more often. You can avoid having to decide on the frequency of the statistics collection by letting the database itself decide when to collect new statistics. Remember that the database bases its statistics collection on whether the statistics are “fresh” or “stale.” Thus, you can relax and let the database be the arbiter of how often to collect statistics.

What Happens When You Don’t Have Statistics

You’ve seen how the Oracle database can automatically collect optimizer statistics for you. You’ve also learned how to use the `DBMS_STATS` package to collect the statistics manually yourself. But what happens if you disable the automatic statistics collection process, or if you don’t collect statistics in a timely fashion? Even with automatic statistics collection, under which necessary statistics are collected on a nightly basis, you may have a situation where table data is altered after the statistics collection process is over. In situations such as this, Oracle uses data, such as the number of blocks taken up by the table data and other ancillary information, to figure out the optimizer execution plan.

You can also use the initialization parameter `OPTIMIZER_DYNAMIC_SAMPLING` to let Oracle estimate optimizer statistics on the fly, when no statistics exist for a table, or when the statistics exist but are too old or otherwise unreliable. Of course, sampling statistics dynamically would mean that the compile time for the SQL statement involved would be longer. Oracle smartly figures out if the increased compile time is worth it when it encounters objects without statistics. If it’s worth it, Oracle will sample a portion of the object’s data blocks to estimate statistics. Note that the additional compile time is really not relevant because it happens only once at the initial parsing stage and not for all the subsequent executions for a SQL statement. You need to set the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter to 2 or higher to enable dynamic sampling of all unanalyzed tables. Because the default for this parameter is 2, dynamic sampling is turned on by default in your database. Thus, you need not spend sleepless nights worrying about objects with missing or outdated statistics. In any case, if you adhere to Oracle’s recommendation and use the Automatic Optimizer Statistics Collection feature, the `GATHER_STATS_JOB` will automatically collect your database’s statistics.
The \texttt{GATHER_STATS_JOB} is created at database creation time and is managed by the Oracle Scheduler, which runs the job when the maintenance window is opened. By default, the maintenance window opens every night from 10 p.m. to 6 a.m., and all day on weekends. Oracle will collect statistics for all objects that need them if you adopt the Automatic Optimizer Statistics Collection feature. The feature is turned on by default in a new Oracle 11g database or when you upgrade to the 11g release from an older release-based database.

**Using the OEM to Collect Optimizer Statistics**

As with so many other DBA tasks in Oracle Database 11g, you’re better off simply using the OEM Database Control or the Grid Control to schedule the collection of optimizer statistics. Here are the steps to collect optimizer statistics using the Database Control or Grid Control interfaces of the OEM:

1. From the Database Control home page, click the Administration tab.
2. In the Administration page, click the Manage Optimizer Statistics link under the Statistics Management group.
3. You’re now in the Manage Optimizer Statistics page. Click the Gather Statistics link to start collecting statistics and follow the instructions for the five steps you must implement.

Figure 19-1 shows part of the optimizer statistics collection process using the OEM Grid Control interface.

![Figure 19-1. Collecting optimizer statistics through the OEM](image-url)
Note Oracle strongly recommends that you just use the Oracle-created GATHER_STATS_JOB, run by the Scheduler during the scheduled maintenance window, to collect optimizer statistics. You may want to collect optimizer statistics manually under an extreme situation, such as the database not being up during the scheduled maintenance window, or if you want to analyze a newly created table right away.

Writing Efficient SQL

One of the trickiest and most satisfying aspects of a DBA’s job is helping to improve the quality of SQL code in the application. Efficient code means fast performance, and an easy way to decrease the I/O your query requires is to try to lower the number of rows that the optimizer has to examine. The optimizer is supposed to find the optimal plan based on your query. This means the optimizer won’t rewrite an inefficiently written query—it only produces the execution plan for that query. Also, even if your query is efficiently written, the optimizer may not always end up producing the best execution plan. You have better knowledge of your application and data than the optimizer does, and you can, with hints, force the optimizer to use that knowledge. The following sections cover some of the best guidelines for writing good SQL.

Efficient WHERE Clauses

Selective criteria in your WHERE clauses can dramatically decrease the amount of data Oracle has to consider during a query. You can follow some simple principles to ensure that the structure of your SQL statements is not inherently inefficient. Your join methods may be fine, but overlooking some of these principles could doom your statement from a performance point of view.

Careful specification of WHERE conditions can have a significant bearing on whether the optimizer will choose existing indexes. The principle of selectivity—the number of rows returned by a query as a percentage of the total number of rows in a table—is the key idea here. A low percentage means high selectivity and a high percentage means the reverse. Because more selective WHERE clauses mean fewer I/Os, the CBO tends to prefer to choose those kinds of WHERE clauses over others in the same query. The following example makes this clear:

```
SQL> SELECT * FROM national_employees
WHERE ss_no = 515086789
AND city='DALLAS';
```

Two WHERE clauses are in this example, but you can see that the first WHERE clause that uses ss_no requires fewer I/Os. The column ss_no is the primary key and is highly selective—only one row with that ss_no is in the entire table. The optimizer determines the selectivity of each of the two columns in the query by looking at the index statistics, which tell it how many rows in the table contain each of the two column values in the query. If neither of the columns has an index, Oracle will use a full table scan to retrieve the answer to the query. If both of them have indexes, it will use the more selective (and hence more efficient) index on the ss_no column.

If you think that the optimizer should have used an index instead of doing a full table scan, then perform the following steps:

1. Views in a query sometimes prevent the use of indexes. Check to make sure that the execution plan shows that the correct indexes are being used.
2. If you think heavy data skew is in the table, use histograms to provide Oracle with a more accurate representation of the data distribution in the table. The CBO assumes a uniform distribution of column data. The CBO may forego the use of an index even when a column value is selective, because the column itself is unselective in nature. Histograms help by providing the CBO with an accurate picture of the column data distribution. I discuss histograms later in this chapter, in the section “Using Histograms.”

3. If Oracle is still refusing to use the index, force it to do so by using an index hint, as explained in the section “Using Hints to Influence the Execution Plan” later in this chapter.

**Note** It isn’t always obvious why Oracle doesn’t use an index. For example, Oracle may not use an index because the indexed columns are part of an IN list, and the consequent transformation prevents the use of an index.

If you use a WHERE clause such as WHERE \texttt{last\_name} LIKE ‘%MA%’, the optimizer might just decide to skip the index and do a full scan of the table because it needs to perform a pattern match of the entire \texttt{LAST\_NAME} column to retrieve data. The optimizer correctly figures that it will go ahead and look up just the table, instead of having to read both the index and the table values. For example, if a table has 1,000 rows placed in 200 blocks, and you perform a full table scan assuming that the database has set the \texttt{DB\_FILE\_MULTIBLOCK\_READ\_COUNT} to 8, you’ll incur a total of 25 I/Os to read in the entire table. If your index has a low selectivity, most of the index has to be read first. If your index has 40 leaf blocks and you have to read 90 percent of them to get the indexed data first, your I/O is already at 32. On top of this, you have to incur additional I/O to read the table values. However, a full table scan costs you only 25 I/Os, making that a far more efficient choice than using the index. Be aware that the mere existence of an index on a column doesn’t guarantee that it will be used all the time.

You’ll look at some important principles to make your queries more efficient in the following sections.

**Using SQL Functions**

If you use SQL functions in the WHERE clause (for example, the \texttt{SUBSTR}, \texttt{INSTR}, \texttt{TO\_DATE}, and \texttt{TO\_NUMBER} functions), the Oracle optimizer will ignore the index on that column. Make sure you use a function-based index if you must use a SQL function in the WHERE clause.

**Using the Right Joins**

Most of your SQL statements will involve multitable joins. Often, improper table-joining strategies doom a query. Here are some pointers regarding joining tables wisely:

- Using the \texttt{equi\_join} leads to a more efficient query path than otherwise. Try to use equi joins wherever possible.
- Performing filtering operations early reduces the number of rows to be joined in later steps. For example, a WHERE condition applied early reduces the row source that needs to be joined to another table. The goal is to use the table that has the most selective filter as the driving table, because this means fewer rows are passed to the next step.
- Join in the order that will produce the least number of rows as output to the parent step.

**Using the CASE Statement**

When you need to calculate multiple aggregates from the same table, avoid writing a separate query for each aggregate. With separate queries, Oracle has to read the entire table for each query. It’s
more efficient to use the `CASE` statement in this case, as it enables you to compute multiple aggregates from the table with just a single read of the table.

**Efficient Subquery Execution**

Subqueries perform better when you use IN rather than EXISTS. Oracle recommends using the IN clause if the subquery has the selective `WHERE` clause. If the parent query contains the selective `WHERE` clause, use the EXISTS clause rather than the IN clause.

**Using WHERE Instead of HAVING**

Wherever possible, use the `WHERE` clause instead of the `HAVING` clause. The `WHERE` clause restricts the number of rows retrieved at the outset. The `HAVING` clause forces the retrieval of a lot more rows than necessary. It then also incurs the additional overhead of sorting and summing.

**Minimizing Table Lookups**

One of the primary mottos of query writing is “Visit the data as few times as possible.” This means getting rid of SQL that repeatedly accesses a table for different column values. Use multicolour updates instead.

**Using Hints to Influence the Execution Plan**

The assumption that underlies the use of the CBO is that the optimizer knows best. That is, by evaluating the various statistics, the CBO will come to the best decision in terms of choosing the optimal execution plan. However, the optimizer is based on rules, and a good application developer has knowledge about the application and data that the CBO can’t exploit. You can provide hints to the optimizer to override the CBO’s execution plans. For example, if you know that a certain index is more selective than another, you can force Oracle to use that index by providing the hint in your query.

Hints can alter the join method, join order, or access path. You can also provide hints to parallelize the SQL statement operations. The following are some of the common hints that you can use in SQL statements:

- **`ALL_ROWS`**: The `ALL_ROWS` hint instructs Oracle to optimize throughput (that is, minimize total cost), not optimize the response time of the statement.
- **`FIRST_ROWS(n)`**: The `FIRST_ROWS(n)` hint dictates that Oracle return the first $n$ rows quickly. Low response time is the goal of this hint.

**Note**: When you specify `ALL_ROWS` or the `FIRST_ROWS(n)` hint, it overrides the current value of the `OPTIMIZER_MODE` parameter, if it's different from that specified by the hint.

- **`FULL`**: The `FULL` hint requires that a full scan be done on the table, ignoring any indexes that may be present. You would want to do this when you have reason to believe that using an index in this case will be inefficient compared to a full table scan. To force Oracle to do a full table scan, you use the `FULL` hint.
- **`ORDERED`**: This hint forces the join order for the tables in the query.
- **`INDEX`**: This hint forces the use of an index scan, even if the optimizer was going to ignore the indexes and do a full table scan for some reason.
• **INDEX_FFS**: An index fast full scan (INDEX_FFS) hint forces a fast full scan of an index, just as if you did a full table scan that scans several blocks at a time. INDEX_FFS scans all the blocks in an index using multiblock I/O, the size of which is determined by the `DB_FILE_MULTIBLOCK_READ_COUNT` parameter. You can also parallelize an INDEX_FFS hint, and it’s generally preferable to a full table scan.

The `OPTIMIZER_MODE` settings determine the way the query optimizer performs optimization throughout the database. However, at times, due to lack of accurate statistics, the optimizer can be mistaken in its estimates, leading to poor execution plans. In cases such as this, you can use optimizer hints to override this database optimization setting at the individual SQL statement level. Oracle Database 11g also provides the SQL Profile feature. This feature enables you to collect auxiliary information using sampling and partial execution techniques, thereby avoiding the use of optimizer hints. I discuss SQL profiles in the section titled “Using the SQL Tuning Advisor on SQL Statements,” later in this chapter.

**Selecting the Best Join Method**

Choose a join method based on how many rows you expect to be returned from the join. The optimizer generally tries to choose the ideal join condition, but it may not do so for various reasons. It’s up to you to see what join method the optimizer will adopt and change it if necessary. The following guidelines will help you when you’re analyzing output produced by an EXPLAIN PLAN.

**Avoiding Cartesian Joins**

Cartesian joins usually aren’t the result of intentional planning; rather, they happen due to logical mistakes in the query. Cartesian joins are produced when your joins don’t have any `WHERE` clauses. If you’re joining several tables, make sure that each table in the join is referenced by a `WHERE` condition. Even if the tables being joined are small, avoid Cartesian joins because they’re inefficient. For example, if the employee table has 2,000 rows and the dept table has 100 rows, a Cartesian join of employee and dept will have 2,000 * 100 = 200,000 rows.

**Nested Loops**

If you’re joining small subsets of data, the nested loop (NL) method is ideal. If you’re returning fewer than, say, 10,000 rows, the NL join may be the right join method. If the optimizer is using hash joins or full table scans, force it to use the NL join method by using the following hint:

```sql
SELECT /*+ USE_NL (TableA, TableB) */
```

**Hash Join**

If the join will produce large subsets of data or a substantial proportion of a table is going to be joined, use the hash join hint if the optimizer indicates it isn’t going to use it:

```sql
SELECT /* USE_HASH */
```

**Merge Join**

If the tables in the join are being joined with an inequality condition (not an equi join), the merge join method is ideal:

```sql
SELECT /*+ USE_MERGE (TableA, TableB) */
```
Using Bitmap Join Indexes

Bitmap join indexes (BJIs) *prestore* the results of a join between two tables in an index, and thus do away with the need for an expensive runtime join operation. BJIs are specially designed for data warehouse star schemas, but any application can use them as long as there is a primary key/foreign key relationship between the two tables.

Typically, in a data warehouse setting, the primary key is in a dimension table and the fact table has the foreign key. For example, customer_id in the customer dimension table is the primary key, and customer_id in the fact table is the foreign key. Using a BJI, you can avoid a join between these two tables because the rows that would result from the join are already stored in the BJI. Let’s look at a simple example of a BJI here.

Say you expect to use the following SQL statement frequently in your application:

```sql
SQL> SELECT SUM((s.quantity)
  2  FROM sales s, customers c
  3  WHERE s.customer_id = c.customer_id
  4  AND c.city = 'DALLAS';
```

In this example, the sales table is the fact table with all the details about product sales, and the customers table is a dimension table with information about your customers. The column customer_id acts as the primary key for the customers table and as the foreign key for the sales table, so the table meets the requirement for creating a BJI.

The following statement creates the BJI. Notice line 2, where you’re specifying the index on the city column (c.city). This is how you get the join information to place in the new BJI. Because the sales table is partitioned, you use the clause `LOCAL` in line 5 to create a locally partitioned index:

```sql
SQL> CREATE BITMAP INDEX cust_city_BJI
  2  ON city (c.city)
  3  FROM sales s, customers c
  4  WHERE c.cust_id = s.cust_id
  5  LOCAL
  6  TABLESPACE users;
```

Index created.

You can confirm that the intended index has been created with the help of the following query. The first index is the new BJI index you just created:

```sql
SQL> SELECT index_name, index_type, join_index
  2  FROM dba_indexes
  3  WHERE table_name='SALES';
```

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>INDEX_TYPE</th>
<th>JOIN_INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUST_CITY_BJI</td>
<td>BITMAP</td>
<td>YES</td>
</tr>
<tr>
<td>SALES_CHANNEL_BIX</td>
<td>BITMAP</td>
<td>NO</td>
</tr>
<tr>
<td>SALES_CUST_BIX</td>
<td>BITMAP</td>
<td>NO</td>
</tr>
</tbody>
</table>

3 rows selected.

Being a bitmap index, the new BJI uses space extremely efficiently. However, the real benefit of using this index is that when you need to find out the sales for a given city, you don’t need to join the sales and customers tables. You only need to use the sales table and the new BJI that holds the join information already.
Selecting the Best Join Order

When your SQL statement includes a join between two or more tables, the order in which you join the tables is extremely important. The driving table in a join is the first table that comes after the WHERE clause. The driving table in the join should contain the filter that will eliminate the most rows. Choose the join order that gives you the least number of rows to be joined to the other tables. That is, if you’re joining three tables, the one with the more restrictive filter should be joined first to one of the other two tables. Compare various join orders and pick the best one after you consider the number of rows returned by each join order.

Indexing Strategy

An index is a data structure that takes the value of one or more columns of a table (the key) and returns all rows (or the requested columns in a row) with that value of the column quickly. The efficiency of an index comes from the fact that it lets you find necessary rows without having to scan all the rows of a table. As a result, indexes are more efficient in general, because they need fewer disk I/Os than if you had to scan the table.

Note For a quick summary of indexing guidelines, please refer to the section “Guidelines for Creating Indexes” in Chapter 7.

Developers are content when the EXPLAIN PLAN indicates that a query was using indexes. However, there’s more to query optimization than simply using an index for speeding up your queries. If you don’t use good indexes, your queries could slow down the database significantly. Important things to consider are whether you have the right indexes or even if the index is necessary in a certain query. In the next sections you’ll look at some of the issues you should consider regarding the use of indexes.

Caution A common problem is that an index that performs admirably during development and testing phases simply won’t perform well on a production database. Often, this is due to the much larger amounts of data in the “real” system than in the development system. Ideally, you should develop and test queries on an identical version of the production database.

When to Index

You need to index tables only if you think your queries will be selecting a small portion of the table. If your query is retrieving rows that are greater than 10 or 15 percent of the total rows in the table, you may not need an index. Remember that using an index prevents a full table scan, so it is inherently a faster means to traverse a table’s rows. However, each time you want to access a particular row in an indexed table, first Oracle has to look up the column referenced in your query in its index. From the index, Oracle obtains the ROWID of the row, which is the logical address of its location on disk.

If you choose to enforce uniqueness of the rows in a table, you can use a primary index on that table. By definition, a column that serves as a primary index must be non-null and unique. In addition to the primary index, you can have several secondary indexes. For example, the attribute LAST_NAME may serve as a primary index. However, if most of your queries include the CITY column, you may choose to index the CITY column as well. Thus, the addition of secondary indexes would enhance query
performance. However, a cost is associated with maintaining additional secondary indexes. In addition to the additional disk space needed for large secondary indexes, remember that all inserts and updates to the table require that the indexes also be updated.

If your system involves a large number of inserts and deletes, understand that too many indexes may be detrimental, because each DML causes changes in both the table and its indexes. Therefore, an OLTP-oriented database ought to keep its indexes to a minimum. A data warehouse, on the other hand, can have a much larger number of indexes because there is no penalty to be paid. That’s because the data warehouse is a purely query-oriented database, not a transactional database.

What to Index

Your goal should be to use as few indexes as possible to meet your performance criteria. There’s a price to be paid for having too many indexes, especially in OLTP databases. Each INSERT, UPDATE, and DELETE statement causes changes to be made to the underlying indexes of a table, and can slow down an application in some cases. The following are some broad guidelines you can follow to make sure your indexes help the application instead of hindering it:

- Index columns with high selectivity. Selectivity here means the percentage of rows in a table with a certain value. High selectivity, as you learned earlier in this chapter, means that there are few rows with identical values.
- Index all important foreign keys.
- Index all predicate columns.
- Index columns used in table joins.

Proper indexing of tables involves carefully considering the type of application you’re running, the number of DML operations, and the response time expectations. Here are some additional tips that can aid you in selecting appropriate indexes for your application:

- Try to avoid indexing columns that consist of long character strings, unless you’re using the Oracle Text feature.
- Wherever possible, use index-only plans, meaning a query that can be satisfied completely by just the data in the index alone. This requires that you pay attention to the most common queries and create any necessary composite indexes (indexes that include more than one column attribute).
- Use secondary indexes on columns frequently involved in ORDER BY and GROUP BY operations, as well as sorting operations such as UNION or DISTINCT.

Using Appropriate Index Types

The B-tree index (sometimes referred to as the B*tree index) is the default or normal type of Oracle index. You’re probably going to use it for almost all the indexes in a typical OLTP application. Although you could use the B-tree index for all your index needs, you’ll get better performance by using more specialized indexes for certain kinds of data. Your knowledge of the type of data you have and the nature of your application should determine the index type. In the next few sections, you’ll see several alternative types of indexes.

Bitmap Indexes

Bitmap indexes are ideal for column data that has a low cardinality, which means that the indexed column has few distinct values. The index is compact in size and performs better than the B-tree index for these types of data. However, the bitmap index is going to cause some problems if a lot of DML is going on in the column being indexed.
Index-Organized Tables

Index-organized tables (IOTs) are explained in Chapter 7. The traditional Oracle tables are called heap-organized tables, where data is stored in the order in which it is inserted. Indexes enable fast access to the rows. However, indexes also mean more storage and the need for accessing both the index and the table rows for most queries (unless the query can be selected just by the indexed columns themselves). IOTs place all the table data in its primary key index, thus eliminating the need for a separate index.

IOTs are more akin to B-tree indexes than tables. The data in an IOT is sorted, and rows are stored in primary key order. This type of organization of row values gives you faster access in addition to saving space. To limit the size of the row that’s stored in the B-tree leaf blocks, IOTs use an overflow area to store infrequently accessed non-key columns, which leads to lower space consumption in the B-tree.

Concatenated Indexes

Concatenated or composite indexes are indexes that include more than one column, and are excellent for improving the selectivity of the WHERE predicates. Even in cases where the selectivity of the individual columns is poor, concatenating the index improves selectivity. If the concatenated index contains all the columns in the WHERE list, you’re saved the trouble of looking up the table, thus reducing your I/O. However, you have to pay particular attention to the order of the columns in the composite index. If the WHERE clause doesn’t specify the leading column of the concatenated index first, Oracle may not use the index at all.

Up until recently, Oracle used a composite index only if the leading column of the index was used in the WHERE clause or if the entire index was scanned. The index skip scan feature lets Oracle use a composite index even when the leading column isn’t used in the query. Obviously, this is a nice feature that eliminates many full table scans that would have resulted in older versions of Oracle.

Function-Based Indexes

A function-based index contains columns transformed either by an Oracle function or by an expression. When the function or expression used to create the index is referenced in the WHERE clause of a query, Oracle can quickly return the computed value of the function or expression directly from the index, instead of recalculating it each time. Function-based indexes are efficient in frequently used statements that involve functions or complex expressions on columns. For example, the following function-based index lets you search for people based on the last_name column (in all uppercase letters):

```
SQL> CREATE INDEX upper_lastname_idx ON employees (UPPER(last_name));
```

Reverse-Key Indexes

If you’re having performance issues in a database with a large number of inserts, you should consider using reverse-key indexes. These indexes are ideal for insert-heavy applications, although they suffer from the drawback that they can’t be used in index range scans. A reverse-key index looks like this:

<table>
<thead>
<tr>
<th>Index value</th>
<th>Reverse Key Index Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9001</td>
<td>1009</td>
</tr>
<tr>
<td>9002</td>
<td>2009</td>
</tr>
<tr>
<td>9003</td>
<td>3009</td>
</tr>
<tr>
<td>9004</td>
<td>4009</td>
</tr>
</tbody>
</table>

When you’re dealing with columns that sequentially increase, the reverse-key indexes provide an efficient way to distribute the index values more evenly and thus improve performance.
Partitioned Indexing Strategy

As you saw in Chapter 7, partitioned tables can have several types of indexes on them. Partitioned indexes can be local or global. In addition, they can be prefixed or nonprefixed indexes. Here’s a brief summary of important partitioned indexes:

- **Local partitioned indexes** correspond to the underlying partitions of the table. If you add a new partition to the table, you also add a new partition to the local partitioned index.
- **Global partitioned indexes** don’t correspond to the partitions of the local table.
- **Prefixed indexes** are partitioned on a left prefix on the index columns.
- **Nonprefixed indexes** are indexes that aren’t partitioned on the left prefix of the index columns.

In general, local partitioned indexes are a good indexing strategy if the table has been indexed primarily for access reasons. If your queries include columns that aren’t a part of the partitioned table’s key, global prefixed indexes are a good choice. Using global prefixed indexes is a good indexing strategy if the table has been indexed primarily for access reasons. Local nonprefixed indexes are good if you’re using parallel query operations.

---

**Note** In Chapter 5, I showed how to use the SQL Access Advisor to get advice concerning the creation of indexes and materialized views (and materialized view logs). Use the SQL Access Advisor on a regular basis to see if you need to create any new indexes or materialized views (or materialized view logs).

---

Monitoring Index Usage

You may have several indexes on a table, but that in itself is no guarantee that they’re being used in queries. If you aren’t using indexes, you might as well get rid of them, as they just take up space and time to manage them. You can use the V$OBJECT_USAGE view to gather index usage information. Here’s the structure of the V$OBJECT_USAGE view:

```
SQL> DESC V$OBJECT_USAGE
Name                  Null?         Type
-----------------       --------     ------------
INDEX_NAME             NOT NULL     VARCHAR2(30)
TABLE_NAME             NOT NULL     VARCHAR2(30)
MONITORING             NOT NULL     VARCHAR2(3)
USED                   VARCHAR2(3)
START_MONITORING       VARCHAR2(19)
END_MONITORING         VARCHAR2(19)
SQL>
```

Chapter 7 shows how to use the V$OBJECT_USAGE view to find out if a certain index is being used.

---

Removing Unnecessary Indexes

The idea of removing indexes may seem surprising in the beginning, but you aren’t being asked to remove just any index on a table. By all means, keep the indexes that are being used and that are also selective. If an index is being used but it’s a nonselective index, you may be better off in most cases getting rid of it, because the index will slow down the DML operations without significantly increasing performance. In addition, unnecessary indexes just waste space in your system.
Using Similar SQL Statements

As you know by now, reusing already parsed statements leads to a performance improvement, besides conserving the use of the shared pool area of the SGA. However, the catch is that the SQL statements must be identical in all respects, white space and all.

Reducing SQL Overhead Via Inline Functions

Inline stored functions can help improve the performance of your SQL statements. Here’s a simple example to demonstrate how you can use an inline function to reduce the overhead of a SQL statement. The following code chunk shows the initial SQL statement without the inline function:

```
SQL> SELECT r.emp_id,
          e.name, r.emp_type,t.type_des,
          COUNT(*)
FROM employees e, emp_type t, emp_records r
WHERE r.emp_id = e.emp_id
AND r.emp_type = t.emp_type
GROUP BY r.emp_id, e.name, r.emp_type, t.emp_des;
```

You can improve the performance of the preceding statement by using an inline function call. First, you create a couple of functions, which you can call later on from within your SQL statement. The first function is called `SELECT_EMP_DETAIL`, and it fetches the employee description if you provide `emp_type` as an input parameter. Here’s how you create this function:

```
SQL> CREATE OR REPLACE FUNCTION select_emp_detail (type IN) number
  2  RETURN varchar2
  3  AS
  4  detail varchar2(30);
  5  CURSOR a1 IS
  6  SELECT emp_detail FROM emp_type
  7  WHERE emp_type = type;
  8  BEGIN
  9  OPEN a1;
 10  FETCH a1 into detail;
 11  CLOSE a1;
 12  RETURN (NVL(detail,'?'));
 13  END;
Function created.
```

Next, create another function, `SELECT_EMP`, that returns the full name of an employee once you pass it `employee_id` as a parameter:

```
SQL> CREATE OR REPLACE FUNCTION select_emp (emp IN number) RETURN varchar2
  2  AS
  3  emp_name varchar2(30);
  4  CURSOR a1 IS
  5  SELECT name FROM employees
  6  WHERE employee_id = emp;
  7  BEGIN
  8  OPEN a1;
  9  FETCH a1 INTO emp_name;
 10  CLOSE a1;
 11  RETURN (NVL(emp_name,'?'));
 12  END;
```

Function created.
SQL>

Now that you have both your functions, it’s a simple matter to call them from within a SQL statement, as the following code shows:

SQL> SELECT r.emp_id, select_emp(r.emp_id),
   2  r.emp_type, select_emp_desc(r.emp_type),
   3  COUNT(*)
   4  FROM emp_records r
   5* GROUP BY r.emp_id, r.emp_type;
SQL>

Using Bind Variables

The parsing stage of query processing consumes resources, and ideally you should parse just once and use the same parsed version of the statement for repeated executions. Parsing is a much more expensive operation than executing the statement. You should use bind variables in SQL statements instead of literal values to reduce the amount of parsing in the database. Bind variables should be identical in terms of their name, data type, and length. Failure to use bind variables leads to heavy use of the shared pool area and, more often than not, contention for latches and a general slowing down of the database when a large number of queries are being processed. Sometimes your application may not be changeable into a form where bind variables are used.

In Chapter 20, you’ll see how to use Oracle configuration parameters to force statements that fail to use bind variables to do so.

Avoiding Improper Use of Views

Views have several benefits to offer, but faster performance may not necessarily be one of them. Views are useful when you want to present only the relevant portions of a table to an application or a user. Whenever you query a view, it has to be instantiated at that time. Because the view is just a SQL query, it has to perform this instantiation if you want to query the view again. If your query uses joins on views, it could lead to substantial time for executing the query.

Avoiding Unnecessary Full Table Scans

Full table scans can occur sometimes, even when you have indexed a table. The use of functions on indexed columns is a good example for when you unwittingly can cause Oracle to skip indexes and go to a full table scan. You should avoid the use of inequality and the greater than or equal to predicates, as they may also bypass indexes.

How the DBA Can Help Improve SQL Processing

Performance tuning involves the optimization of SQL code and the calibration of the resources used by Oracle. The developers generally perform SQL tuning, and the DBA merely facilitates their tuning efforts by setting the relevant initialization parameters, turning tracing on, and so on. Nevertheless, the DBA can implement several strategies to help improve SQL processing in his or her database.

In some cases, you and the developers might be working together to optimize the application. What if you can’t modify the code, as is the case when you’re dealing with packaged applications? Alternatively, what if even the developers are aware that major code changes are needed to improve performance, but time and budget constraints make the immediate revamping of the application difficult? There are several ways you can help without having to change the code itself.
It’s common for DBAs to bemoan the fact that the response times are slow because of poorly written SQL. I’ve heard this in every place I’ve worked, so I assume this is a universal complaint of DBAs who have to manage the consequences of bad code. A perfectly designed and coded application with all the right joins and smart indexing strategies would be nice, but more often than not, that perfect state of affairs doesn’t happen. The theory of the next best option dictates that you should do everything you can to optimize within the limitations imposed by the application design.

That said, let’s look at some of the important ways in which you can help improve query performance in an application, even when you can’t change the code right away.

**Using Partitioned Tables**

Partitioned tables usually lead to tremendous improvements in performance, and they’re easy to administer. By partitioning a table into several subpartitions, you’re in essence limiting the amount of data that needs to be examined to satisfy your queries. If you have large tables, running into tens of millions of rows, consider partitioning them.

Five table partitioning schemes are available to you in Oracle Database 11g, and I explain them in Chapter 7. You can index partitioned tables in a variety of ways, depending on the needs of the application. Partition maintenance is also easy, and it’s well worth the additional effort when you consider the tremendous gains partitioned tables provide.

**Using Compression Techniques**

The Oracle database lets you use table compression to compress tables, table partitions, and materialized views. Table compression helps reduce space requirements for the tables and enhances query performance. Oracle compresses the tables by eliminating the duplicate values in a data block and replacing those values with algorithms to re-create the data when necessary. The table compression technique is especially suitable for data warehouse and OLAP databases, but OLTP databases can also use the technique fruitfully. The larger the table that is compressed, the more benefits you’ll achieve with this technique. Here’s a simple table compression statement:

```sql
SQL> CREATE table sales_compress
2  COMPRESS
3  AS SELECT * FROM sh.sales;
Table created.
SQL>
```

You can also use index key compression to compress the primary key columns of IOTs. This compression not only saves you storage space, but also enhances query performance. Index compression works by removing duplicate column values from the index.

To compress an index, all you have to do is add the keyword COMPRESS after the index-creation statement, as shown here:

```sql
SQL> CREATE INDEX item_product_x
2  ON order_items(product_id)
3  TABLESPACE order_items_indx_01
4  COMPRESS;
Index created.
SQL>
```

Perform some tests to confirm the space savings and the time savings during the creation statements. Later, you can test query performance to measure the improvement.
Using Materialized Views

If you’re dealing with large amounts of data, you should seriously consider using materialized views to improve response time. *Materialized views* are objects with data in them—usually summary data from the underlying tables. Expensive joins can be done beforehand and saved in the materialized view. When users query the underlying table, Oracle automatically rewrites the query to access the materialized view instead of the tables.

Materialized views reduce the need for several complex queries because you can precalculate aggregates with them. Joins between large tables and data aggregation are expensive in terms of resource usage, and materialized views significantly reduce the response time for complex queries on large tables. If you aren’t sure which materialized views to create, not to worry—you can use the DBMS_OLAP package supplied by Oracle to get recommendations on ideal materialized views.

Chapter 7 discusses materialized views in more detail, and also shows you how to use the SQL Access Advisor tool to get recommendations for creating materialized views and materialized view logs.

Using Stored Outlines to Stabilize the CBO

As I mentioned earlier in this chapter, the CBO doesn’t always use the same execution strategies. Changes in Oracle versions or changes in the initialization parameters concerning memory allocation may force the CBO to modify its plans. You can use Oracle’s plan stability feature to ensure that the execution plan remains stable regardless of any changes in the database environment.

The plan stability feature uses stored outlines to preserve the current execution plans, even if the statistics and optimizer mode are changed. The CBO uses the same execution plan with identical access paths each time you execute the same query. The catch is that the query must be exactly identical each time if you want Oracle to use the stored plan.

---

**Caution** When you use stored outlines to preserve a currently efficient execution plan, you’re limiting Oracle’s capability to modify its execution plans dynamically based on changes to the database environment and changes to the statistics. Ensure you use this feature for valid purposes, such as maintaining similar plans for distributed applications.

---

On the face of it, the stored outline feature doesn’t seem impressive. Let’s consider a simple example to see how a stored outline could be useful in a real production environment.

Suppose you have a system that’s running satisfactorily and, due to a special need, you add an index to a table. The addition of the new index could unwittingly modify the execution plans of the CBO, and your previously fast-running SQL queries may slow down. It could conceivably take a lot of effort, testing, and time to fix the problem by changing the original query. However, if you had created stored outlines, these kinds of problems wouldn’t arise. Once Oracle creates an outline, it stores it until you remove it. In the next section you’ll examine how to implement planned stability in a database.

When to Use Outlines

Outlines are useful when you’re planning migrations from one version of Oracle to another. The CBO could behave differently between versions, and you can cut your risk down by using stored outlines to preserve the application’s present performance. You can also use them when you’re upgrading your applications. Outlines ensure that the execution paths the queries used in a test instance successfully carry over to the production instance.
Stored outlines are especially useful when the users of an application have information about the environment that the Oracle CBO doesn’t possess. By enabling the direct editing of stored outlines, Oracle lets you tune SQL queries without changing the underlying application. This is especially useful when you’re dealing with packaged applications where you can’t get at the source code.

**Implementing Plan Stability**

Implementing plan stability is a simple matter. You have to ensure that the following initialization parameters are consistent in all the environments. You must set the value of the first two parameters to `TRUE`. The default value for `OPTIMIZER_FEATURES_ENABLE` is 11.1.0.6, and if you change it, make sure it’s the same in all environments. Here are the relevant initialization parameters:

- `QUERY_REWRITE_ENABLED`
- `STAR_TRANSFORMATION_ENABLED`
- `OPTIMIZER_FEATURES_ENABLE`

**Creating Outlines**

The outlines themselves are managed through the DBMS_OUTLN and DBMS_OUTLN_EDIT Oracle packages. To create outlines for all your current SQL queries, you simply set the initialization parameter `CREATE_STORED_OUTLINES` to `TRUE`.

The OUTLN user is part of the database when it is created and owns the stored outlines in the database. The outlines are stored in the table OL$. Listing 19-1 shows the structure of the OL$ table.

**Listing 19-1. The OL$ Table**

```
SQL> DESC OL$
Name       Null?   Type
----------  ------   ------------
OL_NAME     NO       VARCHAR2(30)
SQL_TEXT    YES      LONG
TEXTLEN     NO       NUMBER
SIGNATURE   NO       RAW(16)
HASH_VALUE  NO       NUMBER
HASH_VALUE2 NO       NUMBER
CATEGORY    NO       VARCHAR2(30)
VERSION     NO       VARCHAR2(64)
CREATOR     NO       VARCHAR2(30)
TIMESTAMP   NO       DATE
FLAGS       NO       NUMBER
HINTCOUNT   NO       NUMBER
SPARE1      NO       NUMBER
SPARE2      NO       VARCHAR2(1000)
```

The SQL_TEXT column has the SQL statement that is outlined. In addition to the OL$ table, the user OUTLN uses the OL$HINTS and OL$NODES tables to manage stored outlines.

Create a special tablespace for the user OUTLN and the tables OL$, OL$HINTS, and OL$NODES. By default, they’re created in the System tablespace. After you create a new tablespace for user OUTLN, you can use the export and import utilities to move the tables.
Creating Outlines at the Database Level

To let Oracle automatically create outlines for all SQL statements, use the CREATE_STORED_OUTLINES initialization parameter, as shown here:

```
CREATE_STORED_OUTLINES = TRUE
```

You can also dynamically enable the creation of stored outlines for the entire database by using the ALTER SYSTEM statement, as shown here:

```
SQL> ALTER SYSTEM SET CREATE_STORED_OUTLINES=TRUE;
System altered.
```

In both the preceding cases, the outlines that Oracle creates are assigned to a category called DEFAULT. You also have the option of specifying a named category for your stored outlines. Setting the CREATE_STORED_OUTLINES parameter means that the database creates a stored outline for every distinct SQL statement. This means that the System tablespace could potentially run out of space if you have a large number of SQL statements that are being processed. For this reason, use the CREATE_STORED_OUTLINES initialization parameter with care. To keep the overhead low, you may instead use the option to create stored outlines at the session level, or just for a lone SQL statement, as shown in the next section.

Creating Outlines for Specific Statements

You can create outlines for a specific statement or a set of statements by using the ALTER SESSION statement, as shown here:

```
SQL> ALTER SESSION SET create_stored_outlines = true;
Session altered.
```

Any statements you issue after the ALTER SESSION statement is processed will have outlines stored for them.

To create a stored outline for a specific SQL statement, you use the CREATE OUTLINE statement. The user issuing this command must have the CREATE OUTLINE privilege. The following statement shows how to create a simple outline for a SELECT operation on the employees table:

```
SQL> CREATE OUTLINE test_outline
2  ON SELECT employee_id, last_name
3   FROM hr.employees;
Outline created.
```

You can use the DROP OUTLINE statement to drop an outline, as shown here:

```
SQL> DROP OUTLINE test_outline;
Outline dropped.
```

Using the Stored Outlines

After you create the stored outlines, Oracle won’t automatically start using them. You have to use the ALTER SESSION or ALTER SYSTEM statement to set USE_STORED_OUTLINES to TRUE. The following example uses the ALTER SYSTEM statement to enable the use of the stored outlines at the database level:
You can also set the initialization parameter `USE_STORED_OUTLINES` to `TRUE`, to enable the use of the stored outlines. Otherwise, the database won’t use any stored outlines it has created.

**Editing Stored Outlines**

You can easily change the stored access paths while using the plan stability feature. You can use either the `DBMS_OUTLN_EDIT` package or OEM to perform the changes.

**SQL Plan Management**

Changes such as database upgrades, or even minor changes such as adding or deleting an index, could affect SQL execution plans. I explained the Oracle stored outlines feature earlier in this chapter as a way to preserve SQL execution plans to prevent performance deterioration when the database undergoes major changes such as a database upgrade. Oracle recommends that you use the new feature called SQL Plan Management (SPM) to keep performance from being affected by major system changes. SQL Plan Management preserves database performance under the following types of system changes:

- Database upgrades
- New optimizer version
- Changes in optimizer parameters
- Changes in system settings
- Changes in schema and metadata definitions
- Deployment of new application modules

Although you can tune SQL statements using the SQL Tuning Advisor and ADDM, that’s at best a reactive mechanism and requires the DBA to intervene. SPM is designed as a preventative mechanism. The database controls the evolution of SQL plans using the new SQL plan baselines, which are sets of efficient execution plans captured by the database over a period of time. The database allows a new execution plan to become part of a SQL plan baseline for a statement only if the new plan doesn’t cause a regression in performance. The database uses only those execution plans that are part of a SQL plan baseline to execute SQL statements, and thus the database achieves the key goal of preserving database performance in the face of major system changes such as database upgrades.

The SPM comes in very handy when you’re upgrading to Oracle Database 11g. After you upgrade to Oracle Database 11g from, say, the Oracle Database 10g release, first leave the `OPTIMIZER_FEATURES_ENABLE` parameter at 10.2. Once the SPM mechanism collects the execution plans and stores them as SQL plan baselines, you can switch to the 11.1 setting for the `OPTIMIZER_FEATURES_ENABLE` parameter. This way, you ensure that you’re using all the new capabilities of the 11g release, without compromising SQL performance: performance is safeguarded through the use of SQL plan baselines, which are similar in this regard to the stored outlines maintained by the database.

**SQL Plan Baselines**

Under SQL Plan Management, the database maintains a plan history, which is a record of all SQL plans generated over time for a SQL statement by the optimizer. The optimizer uses the plan history to figure out the optimal execution plan for a statement. Not all plans in the plan history for a statement
are acceptable plans, however. The database defines as acceptable only those execution plans that
don’t lead to deterioration in performance relative to other plans in the plan history. The SQL plan
baseline for a SQL statement is the set of all accepted plans in the plan history for a statement.

The very first plan for a statement is always an accepted plan, because there’s nothing to compare
it with. So, the SQL plan baseline and the plan history for a new SQL statement are identical at this
point. Newer execution plans for the statement will automatically be a part of the statement’s plan
history, but are added to the SQL plan baseline for the statement only if the new plans are verified
not to cause a regression in performance. The Automatic SQL Tuning Adviser task, which is a part
of the automate maintenance tasks, verifies SQL plans. The advisor looks for high-load SQL statements
and stores the accepted plans for those statements in that’s statement’s SQL plan baseline.

You can manage SQL plan baselines by using the DBMS_SPM package or through Enterprise
Manager. I explain the steps in the following sections.

Capturing SQL Plan Baselines

There are two ways to capture the SQL plan baselines: have the database automatically capture the
plans or load them in the database yourself. I explain both techniques in this section.

Capturing Plans Automatically

By default, the database doesn’t maintain a plan history for SQL statements you execute. You must
set the initialization parameter OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES to TRUE (the default value is
FALSE) for the database to start capturing the SQL plan baselines. When you set the parameter to
TRUE, the database automatically creates and maintains a plan history for all repeatable SQL state-
ments that are executed in the database.

Tip  By using the SQL Performance Analyzer (see Chapter 20), you can find out which SQL statements are likely
to regress following a database upgrade to, say, Oracle Database 11g Release 1 from Oracle Database 10g Release
2. You can capture the execution plans for these statements and load them into the SQL management base of the
upgraded database, thus avoiding the performance regression.

Manual Plan Loading

You can also load SQL plans manually into the SQL plan baselines. When you load plans manually,
the database loads them automatically as accepted plans, without verifying the performance of the
plans. You can bulk load SQL plans you captured before upgrading the database into a SQL plan
baseline after upgrading your database to a new release.

You can use either a SQL Tuning Set (STS) or load the plans from the database cursor cache. I
show both techniques in the following sections.

Execute the DBMS_SPM function LOAD_PLANS_FROM_SQLSET in order to load SQL plans from an
STS. First create an empty STS as shown here:

begin
  dbms_sqltune.create_sqlset(
    sqlset_name => 'testset1',
    description => 'Test STS to capture AWR Data');
end;
/

Next, load the new STS with SQL statements from the Automatic Workload Repository (AWR)
snapshots.
declare
baseline_cur dbms_sqltune.sqlset_cursor;
begin
  open baseline_cur for
  select value(p) from table (dbms_sqltune.select_workload_repository(
    'peak baseline',null,null,'elapsed_time',null,null,null,20)) p;
  dbms_sqltune.load_sqlset (
    sqlset_name => 'testset1',
    populate_cursor => baseline_cur);
end;
/

The STS shown in this example includes the top 20 statements from the AWR peak baseline, selected based on the criterion of elapsed time. The ref cursor and the table function help select the top 20 statements from the AWR baseline.

Load the SQL plans from the STS into the SQL plan baseline by executing the LOAD_PLANS_FROM_SQLSET function.

declare
test_plans pls_integer;
begin
  test_plans := dbms_spm.load_plans_from_sqlset(
    sqlset_name => 'testset1');
end;
/

You can also use the cursor cache instead of an STS as the source of the SQL plans you want to load into a SQL plan baseline. The following example shows how to load SQL plans from the cursor cache using the LOAD_PLANS_FROM_CURSOR_CACHE function.

declare
test_plans pls_integer;
begin
  test_plans := dbms_spm.load_plans_from_cursor_cache (
    sql_id => '123456789999')
  return pls_integer;
end;
/

Selecting SQL Plan Baselines

Regardless of whether you collect SQL plans using the AWR as a source or from the cursor cache of the database, you must enable the use of those plans by setting the initialization parameter OPTIMIZER_USE_SQL_PLAN_BASELINES to TRUE. Since the parameter's default value is TRUE, it means the plan baselines are enabled by default.

When the database encounters a new repeatable SQL statement, it sees whether it can match it to a plan in the SQL plan baseline. If there’s a match, it uses the best cost plan to execute the statement. If there’s no match, the database adds the new plan to the plan history as a nonaccepted plan. The database will then choose the least costly plan from the set of accepted plans in the SQL plan baseline and execute the statement using that plan. If the database can't reproduce an accepted plan for some reason (such as the dropping of an index), it selects the least costly plan to execute the SQL statement.
Tip Execute the DBMS_XPLAN DISPLAY_SQL_PLAN_BASELINE function to view the execution plan for a specific SQL_HANDLE in a SQL plan baseline.

The end result is that the optimizer will always produce an execution plan that's either the best cost plan or an accepted plan from the SQL plan baseline. The OTHER_XML column in the PLAN_TABLE's EXPLAIN PLAN output reveals the exact final strategy adopted by the optimizer.

Evolving SQL Plan Baselines

The database routinely checks new plans so as to evolve the SQL plan baselines, which involves changing a nonaccepted plan into an accepted plan and this part of the SQL plan baseline. As mentioned earlier, a nonaccepted plan must show superior performance to an accepted plan in order to be converted into an accepted plan in the baseline. If you're manually loading SQL plans, there is no need to formally evolve the plans, as every plan you load is automatically deemed an accepted plan. However, any plans that the database captures automatically must be formally evolved into the SQL plan baseline.

You can evolve SQL plan baselines either by executing the EVOLVE_SQL_PLAN_BASELINE function or with the help of the SQL Tuning Advisor. The following example shows how to execute the EVOLVE_SQL_PLAN_BASELINE function to add new accepted plans to the baseline.

SQL> exec dbms_spn.evolve_sql_plan_baseline (sql_handle => '123456789111');

The example uses the SQL_HANDLE attribute to specify the plan for a specific SQL statement, but by ignoring this attribute, you can make the database evolve all nonaccepted plans in the database. You can also submit a list of SQL plans if you wish, by specifying the PLAN_LIST attribute.

Tip You can export SQL plan baselines from one database to another by using a staging table.

The SQL Tuning Advisor evolves SQL plan baselines by automatically adding all plans for which you have implemented the advisor’s SQL profile recommendations to the SQL plan baseline for a statement.

Fixed SQL Plan Baselines

You can limit the set of possible accepted plans for SQL statements by setting the FIXED attribute to YES for a SQL plan. When you fix a plan in a baseline, the optimizer prefers it to other nonfixed plans in the baseline, even if some of the other plans are actually cheaper to execute. The database stops evolving a fixed SQL plan baseline, but you can always evolve the baseline by adding a new plan to the baseline.

The following query on the DBA_SQL_PLAN_BASELINES view shows important attributes of SQL plans in the baseline:

SQL> SELECT sql_handle, sql_text, plan_name, origin, enabled, accepted, fixed, autopurge
FROM dba_sql_plan_baselines;
The optimizer only uses those plans that are enabled and have the accepted status.

Managing SQL Plan Baselines

Execute the `DISPLAY_SQL_PLAN_BASELINE` function of the `DBMS_XPLAN` package to view all the SQL plans stored in the SQL plan baseline for a SQL statement. Here’s an example:

```
SQL> set serveroutput on
SQL> set long 100000
SQL> SELECT * FROM table(
  2  dbms_xplan.display_sql_plan_baseline(
      3  sql_handle => 'SYS_SQL_ba5e12ccae97040f',
      4* format => 'basic'));
```

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
SQL handle: SYS_SQL_ba5e12ccae97040f
SQL text: select t.week_ending_day, p.prod_subcategory, sum(s.amount_sold) as dollars, s.channel_id, s.promo_id from sales s, times t, products p where s.time_id = t.time_id and s.prod_id = p.prod_id and s.prod_id>10 and s.prod_id <50 group by t.week_ending_day, p.prod_subcategory,
PLAN_TABLE_OUTPUT
-------------------
PLAN_TABLE_OUTPUT

Plan name: SYS_SQL_PLAN_ae97040f6b60c209
Enabled: YES   Fixed: NO   Accepted: YES   Origin: AUTO-CAPTURE
Plan hash value: 1944768804

PLAN_TABLE_OUTPUT

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>HASH GROUP BY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HASH JOIN</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>TIMES</td>
</tr>
<tr>
<td>4</td>
<td>HASH JOIN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>PRODUCTS</td>
</tr>
<tr>
<td>6</td>
<td>INDEX RANGE SCAN</td>
<td>PRODUCTS_PK</td>
</tr>
<tr>
<td>7</td>
<td>TABLE ACCESS FULL</td>
<td>SALES</td>
</tr>
</tbody>
</table>

29 rows selected.

SQL>
```
The output shows that the SQL plan was captured automatically and is enabled and accepted. It also reveals that the plan isn’t fixed.

Tip When the SQL Tuning Advisor finds execution plans that are superior to the plans in the SQL plan baseline for that statement, it recommends a SQL profile. Once you accept the recommendation for implementing the SQL profile, the SQL Tuning Advisor adds the tuned plan to the SQL plan baseline.

The SQL Management Base

The database stores SQL plan baseline information in the SQL Management Base (SMB), which is stored in the Sysaux tablespace. You can control the sizing and retention period of the SMB by setting the parameters SPACE_BUDGET_PERCENT and PLAN_RETENTION_WEEKS, using the DBMS_SPM package. The following query reveals the current values of the two parameters:

```
SQL> SELECT parameter_name, parameter_value
       FROM dba_sql_management_config;
```

<table>
<thead>
<tr>
<th>PARAMETER_NAME</th>
<th>PARAMETER_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPACE_BUDGET_PERCENT</td>
<td>30</td>
</tr>
<tr>
<td>PLAN_RETENTION_WEEKS</td>
<td>53</td>
</tr>
</tbody>
</table>

The SPACE_BUDGET_PERCENT parameter controls the percentage of space the SMB can occupy in the Sysaux tablespace. The default is 10 percent, and you can set it anywhere between 1 and 50 percent. You can purge outdated SQL plan baselines or SQL profiles from the SMB to clear up space, or you can increase the size of the Sysaux tablespace. You can change the value of the SPACE_BUDGET_PERCENT parameter by executing the CONFIGURE parameter, as shown here:

```
SQL> EXEC dbms_spm.configure ('space_budget_percent', 40);
```

The CONFIGURE procedure specifies that the SPM can use up to 40 percent of the space in the Sysaux tablespace.

The database executes a weekly purging job to remove unused SQL baselines. The database removes any SQL baselines that it hasn’t used in over a year (53 weeks). You can adjust the plan retention period by executing the CONFIGURE procedure as shown here:

```
SQL> exec dbms_spm.configure ('plan_retention_weeks', 105);
```

You can also remove specific baselines from the SMB, as shown in the following example:

```
SQL> exec
   dbms_spm.purge_sql_plan_baseline('SYS_SQL_PLAN_b5429511dd6ab0f');
```

You can query the DBA_SQL_MANAGEMENT_CONFIG view for the current space and retention settings of the SMB.

Using Parallel Execution

Parallel execution of statements can make SQL run more quickly, and it’s especially suitable for large warehouse-type databases. You can set the parallel option at the database or table level. If you increase the degree of parallelism of a table, Oracle could decide to perform more full table scans instead of
using an index, because the cost of a full table scan may now be lower than the cost of an index scan. If you want to use parallel operations in an OLTP environment, make sure you have enough processors on your machine so the CPU doesn’t become a bottleneck.

**Other DBA Tasks**

The DBA must perform certain tasks regularly to optimize the performance of the application. Some of these fall under the routine administrative chores, and the following sections cover some of the important DBA tasks related to performance tuning.

**Collecting System Statistics**

Even if you’re using the Automatic Optimizer Statistics Collection feature, Oracle won’t collect system statistics. As explained earlier in this chapter, you must collect system statistics yourself, so the Oracle optimizer can accurately evaluate alternate execution plans.

**Refreshing Statistics Frequently**

This section applies only if you have turned off the automatic statistics collection process for some reason. Refreshing statistics frequently is extremely important if you’re using the CBO and your data is subject to frequent changes.

How often you run the DBMS_STATS package to collect statistics depends on the nature of your data. For applications with a moderate number of DML transactions, a weekly gathering of statistics will suffice. If you have reason to believe that your data changes substantially daily, schedule the statistics collection on a daily basis.

**Using Histograms**

Normally, the CBO assumes that data is uniformly distributed in a table. There are times when data in a table isn’t distributed in a uniform way. If you have an extremely skewed data distribution in a table, you’re better off using histograms to store the column statistics. If the table data is heavily skewed toward some values, the presence of histograms provides more efficient access methods. Histograms use buckets to represent distribution of data in a column, and Oracle can use these buckets to see how skewed the data distribution is.

You can use the following types of histograms in an Oracle database:

- **Height-based histograms** divide column values into bands, with each band containing a roughly equal number of rows. Thus, for a table with 100 rows, you’d create a histogram with 10 buckets if you wanted each bucket to contain 10 rows.
- **Frequency-based histograms** determine the number of buckets based on the distinct values in the column. Each bucket contains all the data that has the same value.

**Creating Histograms**

You create histograms by using the `METHOD_OPT` attribute of the `DBMS_STATS` procedure such as `GATHER_TABLE_STATS`, `GATHER_DATABASE_STATS`, and so on. You can either specify your own histogram creation requirements by using the `FOR_COLUMNS` clause, or use the `AUTO` or `SKEWONLY` values for the `METHOD_OPT` attribute. If you choose `AUTO`, Oracle will decide which columns it should collect histograms for, based on the data distribution and workload. If you choose `SKEWONLY`, Oracle will base the decision only on the data distribution of the columns. In the two examples that follow, I use the `FOR_COLUMNS` clause to specify the creation of the histograms.
The following example shows how to create a height-based histogram while collecting the optimizer statistics:

```
SQL> BEGIN
    DBMS_STATS.GATHER_table_STATS (OWNNAME => 'HR', TABNAME => 'BENEFITS',
           METHOD_OPT => 'FOR COLUMNS SIZE 10 Number_of_visits');
END;
```

The following example shows how to create a frequency-based histogram:

```
SQL> BEGIN
    DBMS_STATS.GATHER_table_STATS(OWNNAME => 'HR', TABNAME => 'PERSONS',
           METHOD_OPT => 'FOR COLUMNS SIZE 20 department_id');
END;
```

**Viewing Histograms**

You can use the DBA_TAB_COL_STATISTICS view to view histogram information. Following are the two queries that show the number of buckets (num_buckets) and the number of distinct values (num_distinct), first for the height-balanced and then for the frequency-based histogram created in the previous section:

```
SQL> SELECT column_name, num_distinct, num_buckets, histogram
    FROM USER_TAB_COL_STATISTICS
    WHERE table_name = 'BENEFITS' AND column_name = 'NUMBER_OF_VISITS';

COLUMN_NAME           NUM_DISTINCT   NUM_BUCKETS    HISTOGRAM
-----------------     -------------  ------------   ---------------
NUMBER_OF_VISITS          320          10           HEIGHT BALANCED

SQL> SELECT column_name, num_distinct, num_buckets, histogram
    FROM USER_TAB_COL_STATISTICS
    WHERE table_name = 'PERSONS' AND column_name = 'DEPARTMENT_ID';

COLUMN_NAME          NUM_DISTINCT    NUM_BUCKETS    HISTOGRAM
--------------       ------------    ------------   ----------
DEPARTMENT_ID               8            8           FREQUENCY
```

**Adaptive Cursor Sharing**

Although using bind variables improves performance and scalability by reducing parse time and memory usage, literal values actually produce better execution plans than bind values for variables. When you force cursor sharing in the database by setting the CURSOR_SHARING parameter to EXACT or SIMILAR, some SQL statements end up with suboptimal plans for some bind variable values. The Cost-Based Optimizer may very well create a suboptimal plan if it happens to peek at the bind values and if the bind values used by the first SQL statements to go into the shared pool are unrepresentative of the true values of the variable. Developers and DBA’s sometimes resort to setting the unofficial Oracle initialization parameter OPTIM_PEEK_USER_BINDS (ALTER SESSION SET "_optim_peek_user_binds"=FALSE;) to prevent the database from peeking at the bind values. Adaptive cursor sharing provides a much more elegant way to prevent the optimizer from creating suboptimal execution plans caused by bind peeking.

Oracle relies on its “bind peeking” technique when it first parses a SQL statement. The optimizer will always hard parse a new statement and peek at the bind variable values to get a sense of what the values look like. The initial bind values it sees during the bind peeking have an inordinate influence
on the execution plan it chooses for the statement. If, for example, the bind peeking dictates using an index, Oracle will continue to do so, even if later values would actually dictate a full scan instead. Since bind peeking actually leads to suboptimal execution plans in cases such as these, hard-coded variable values would be preferable to bind values.

As the preceding discussion indicates, cursor sharing using bind variables may not always lead to the best (optimal) execution plans. Hard-coded values for variables may actually provide more optimal execution plans than using bind variables, especially when dealing with heavily skewed data. Oracle provides you the adaptive cursor sharing feature, which is an attempt to resolve the conflict between cursor sharing using bind variables and query optimization. Using adaptive cursor sharing, whenever the database estimates that it’s cheaper to produce a new execution plan for a statement than reusing the same cursors, it’ll do so, generating new child cursors for the statement. The database strives to minimize the number of child cursors to take advantage of cursor sharing. However, the database won’t blindly try to reuse cursors.

Tip Adaptive cursor sharing is automatic, and it’s always on and you can’t switch it off.

How Adaptive Cursor Sharing Works

Two concepts—the bind sensitivity of a cursor and a bind-aware cursor—play a critical role in how adaptive cursor sharing works. If changing a bind variable’s values leads to different execution plans, a cursor is called a bind-sensitive cursor. Whenever the database figures that it must create new execution plans because the bind values vary considerably, the variable is deemed bind sensitive. Once the database marks a cursor as bind sensitive, the cursor is termed bind aware.

Note The adaptive cursor sharing feature is independent of the cursor sharing feature.

Here’s an example that illustrates how adaptive cursor sharing works. Suppose you execute the following query several times in your database:

```sql
SQL> select * from hr.employees where salary = :1
and department_id = :2;
```

The SQL statement uses two bind variables, SALARY and DEPARTMENT_ID.

During the very first execution of a new SQL statement, the database makes the cursor bind sensitive if it peeks at the bind values and computes the selectivity of the predicate. The database assigns each execution plan with a set of selectivity values such as (0.25, 0.0050), which indicates the selectivity range of the execution plan. If new bind variables fall within the selectivity range, the optimizer reuses the execution plan, and if not, it creates a new execution plan.

The next step is to evaluate whether the cursor is a bind-aware cursor. After the first hard parse, the database performs soft parses for the subsequent executions and compares the execution statistics with the hard parse execution statistics. If the database decides that the cursor is bind aware, it uses bind-aware cursor matching when it executes the query again. If the new pair of bind values falls inside the selectivity range, the database reuses the plan; otherwise, it performs a hard parse, thus generating a new child cursor with a different plan. If the new execution produces a similar plan, the database merges the child cursors, which means that if the bind values are roughly the same, the statements will share the execution plan.
Monitoring Adaptive Cursor Sharing

The V$SQL view contains two columns, named IS_BIND_SENSITIVE and IS_BIND_AWARE, that help you monitor adaptive cursor sharing in the database. The IS_BIND_SENSITIVE column lets you know whether a cursor is bind sensitive, and the IS_BIND_AWARE column shows whether the database has marked a cursor for bind-aware cursor sharing. The following query, for example, tells you which SQL statements are binds sensitive or bind aware:

```
SQL> SELECT sql_id, executions, is_bind_sensitive, is_bind_aware
       FROM v$sql;
```

```
SQL_ID           EXECUTIONS     I      I
--------------   -----------   ---    ---
57pfs5p8xc07w    21             Y      N
1gfaj4z5hn1kf     4             Y      N
1gfaj4z5hn1kf     4             N      N
...
294 rows selected.
```

In this query, the IS_BIND_SENSITIVE column shows whether the database will generate different execution plans based on bind variable values. Any cursor that shows an IS_BIND_SENSITIVE column value of 'Y' is a candidate for an execution plan change. When the database plans to use multiple execution plans for a statement based on the observed values of the bind variables, it marks the IS_BIND_AWARE column 'Y' for that statement. This means that the optimizer realizes that different bind variable values would lead to different data patterns, which requires the statement to be hard-parsed during the next execution. In order to decide whether to change the execution plan, the database evaluates the next few executions of the SQL statement. If the database decides to change a statement’s execution plan, it marks the cursor bind aware and puts a value of 'Y' in the IS_BIND_AWARE column for that statement. A bind-aware cursor is one for which the database has actually modified the execution plan based on the observed values of the bind variables.

You can use the following views to manage the adaptive cursor sharing feature:

- **V$SQL_CS_HISTOGRAM**: Shows the distribution of the execution count across the execution history histogram
- **V$SQL_CS_SELECTIVITY**: Shows the selectivity ranges stored in cursors for predicates with bind variables
- **V$SQL_CS_STATISTICS**: Contains the execution statistics of a cursor using different bind sets gathered by the database

Rebuilding Tables and Indexes Regularly

The indexes could become unbalanced in a database with a great deal of DML. It’s important to rebuild such indexes regularly so queries can run faster. You may want to rebuild an index to change its storage characteristics or to consolidate it and reduce fragmentation. Use the `ALTER INDEX ... REBUILD` statement, because the old index is accessible while you’re re-creating it. (The alternative is to drop the index and re-create it.)

When you rebuild the indexes, include the `COMPUTE STATISTICS` statement so you don’t have to gather statistics after the rebuild. Of course, if you have a 24/7 environment, you can use the `ALTER INDEX ... REBUILD ONLINE` statement so that user access to the database won’t be affected. It is important that your tables aren’t going through a lot of DML operations while you’re rebuilding...
online, because the online feature may not work as advertised under such circumstances. It might even end up unexpectedly preventing simultaneous updates by users.

Reclaiming Unused Space

The Segment Advisor runs automatically during the scheduled nightly maintenance and provides you with recommendations about objects you can shrink to reclaim wasted space. Just remember that you need to use locally managed tablespaces with Automatic Segment Space Management in order to use the Segment Advisor. Shrinking segments saves space, but more importantly, improves performance by lowering the high-water mark of the segments and eliminating the inevitable fragmentation that occurs over time in objects with heavy update and delete operations.

Caching Small Tables in Memory

If the application doesn’t reuse a table’s data for a long period, the data might be aged out of the SGA and need to be read from disk. You can safely pin small tables in the buffer cache with the following:

```
SQL> ALTER TABLE hr.employees CACHE;
Table altered.
SQL>
```

SQL Performance Tuning Tools

SQL performance tuning tools are extremely important. Developers can use the tools to examine good execution strategies, and in a production database they’re highly useful for reactive tuning. The tools can give you a good estimate of resource use by queries. The SQL tools are the EXPLAIN PLAN, Autotrace, SQL Trace, and TKPROF utilities.

Using EXPLAIN PLAN

The EXPLAIN PLAN facility helps you tune SQL by letting you see the execution plan selected by the Oracle optimizer for a SQL statement. During SQL tuning, you may have to rewrite your queries and experiment with optimizer hints. The EXPLAIN PLAN tool is great for this experimentation, as it immediately lets you know how the query will perform with each change in the code. Because the utility gives you the execution plan without executing the code, you save yourself from having to run untuned software to see whether the changes were beneficial or not. Understanding an EXPLAIN PLAN is critical to understanding query performance. It provides a window into the logic of the Oracle optimizer regarding its choice of execution plans.

The output of the EXPLAIN PLAN tool goes into a table, usually called PLAN_TABLE, where it can be queried to determine the execution plan of statements. In addition, you can use GUI tools, such as OEM or TOAD, to get the execution plan for your SQL statements without any fuss. In OEM, you can view the explain statements from the Top Sessions or the Top SQL charts.

A walkthrough of an EXPLAIN PLAN output takes you through the steps that would be undertaken by the CBO to execute the SQL statement. The EXPLAIN PLAN tool indicates clearly whether the optimizer is using an index, for example. It also tells you the order in which tables are being joined and helps you understand your query performance. More precisely, an EXPLAIN PLAN output shows the following:

- The tables used in the query and the order in which they’re accessed.
- The operations performed on the output of each step of the plan. For example, these could be sorting and aggregation operations.
• The specific access and join methods used for each table mentioned in the SQL statement.
• The cost of each operation.

Oracle creates the PLAN_TABLE as a global temporary table, so all the users in the database can use it to save their EXPLAIN PLAN output. However, you can create a local plan table in your own schema by running the utlxplan.sql script, which is located in the $ORACLE_HOME/rdbms/admin directory. The script, among other things, creates the plan table, where the output of the EXPLAIN PLAN utility is stored for your viewing. You are free to rename this table. Here’s how you create the plan table so you can use the EXPLAIN PLAN feature:

```sql
SQL> @$ORACLE_HOME/rdbms/admin/utlxplan.sql
Table created.
SQL>
```

Creating the EXPLAIN PLAN

To create an EXPLAIN PLAN for any SQL data manipulation language statement, you use a SQL statement similar to that shown in Listing 19-2.

**Listing 19-2. Creating the EXPLAIN PLAN**

```sql
SQL> EXPLAIN PLAN
2  SET statement_id = 'test1'
3  INTO plan_table
4  FOR select p.product_id,i.quantity_on_hand
5  FROM oe.inventories i,
6  oe.product_descriptions p,
7  oe.warehouses w
8  WHERE p.product_id=i.product_id
9  AND i.quantity_on_hand > 250
10  AND w.warehouse_id = i.warehouse_id;
Explained.
SQL>
```

Producing the EXPLAIN PLAN

You can’t easily select the columns out of the PLAN_TABLE table because of the hierarchical nature of relationships among the columns. Listing 19-3 shows the code that you can use so the EXPLAIN PLAN output is printed in a form that’s readable and shows clearly how the execution plan for the statement looks.

**Listing 19-3. Producing the EXPLAIN PLAN**

```sql
SQL> SELECT lpad(' ',level-1)||operation||' '||options||'|' || object_name "Plan"
2  FROM plan_table
3  CONNECT BY prior id = parent_id
4  AND prior statement_id = statement_id
5  START WITH id = 0 AND statement_id = '&1'
6  ORDER BY id;
Enter value for 1: test1
old   6:  START WITH id = 0 AND statement_id = '&1'
new   6:  START WITH id = 0 AND statement_id = 'test1'
Plan
```
SELECT STATEMENT
   HASH JOIN
   NESTED LOOPS
   TABLE ACCESS FULL INVENTORIES
   INDEX UNIQUE SCAN WAREHOUSES_PK
   INDEX FAST FULL SCAN PRD_DESC_PK
6 rows selected.
SQL>

Other Ways of Displaying the EXPLAIN PLAN Results

You can also use the DBMS_XPLAN package to display the output of an EXPLAIN PLAN statement in an easily readable format. You use a table function from this package to display the EXPLAIN PLAN output. You use the DISPLAY table function of the DBMS_XPLAN package to display the output of your most recent EXPLAIN PLAN. You can use the table function DISPLAY_AWR to display the output of the SQL statement’s execution plan from the AWR. Here’s an example that shows how to use the DBMS_XPLAN package to produce the output of the most recent EXPLAIN PLAN statement.

First, create the EXPLAIN PLAN for a SQL statement:

```
SQL> EXPLAIN PLAN FOR
   2   SELECT * FROM persons
   3   WHERE PERSONS.last_name LIKE '%ALAPATI%'
   4   AND created_date < sysdate -30;
Explained.
SQL>
```

Make sure you set the proper line size and page size in SQL*Plus:

```
SQL> SET LINESIZE 130
SQL> SET PAGESIZE 0
```

Display the EXPLAIN PLAN output:

```
SQL> SELECT * FROM table (DBMS_XPLAN.DISPLAY);
---------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU) | Time     |
---------------------------------------------------------------------
| 0   | SELECT STATEMENT  |         | 1     | 37    | 3          | 00:00:01 |
|* 1  | TABLE ACCESS FULL| PERSONS | 1     | 37    | 3          | 00:00:01 |
---------------------------------------------------------------------
Predicate Information (identified by operation id) :
-  filter ("ENAME" LIKE '%ALAPATI%' AND "CREATED_DATE">SYSDATE@!-30)

13 rows selected.
SQL>
```

If you wish, you can use the Oracle-provided utlxpls.sql script to get nicely formatted output. The utlxpls.sql script is an alternative to using the DBMS_XPLAN package directly, and it relies on the same package. The utlxpls.sql script is located in the $ORACLE_HOME/rdbms/admin directory, as I mentioned earlier, and uses the DBMS_XPLAN package to display the most recent EXPLAIN PLAN in the database. Of course, you must make sure that the table PLAN_TABLE exists before you can use the utlxpls.sql script. Here’s how you’d run this script:
The output of the utlxpls.sql script is exactly identical to that of the DBMS_XPLAN.DISPLAY, which was presented a few paragraphs prior.

**Interpreting the EXPLAIN PLAN Output**

Reading an EXPLAIN PLAN is somewhat confusing in the beginning, and it helps to remember these simple principles:

- Each step in the plan returns output in the form of a set of rows to the parent step.
- Read the plan outward starting from the line that is indented the most.
- If two operations are at the same level in terms of their indentation, read the top one first.
- The numbering of the steps in the plan is misleading. Start reading the EXPLAIN PLAN output from the inside out. That is, read the most indented operation first.

In the example shown earlier in Listing 19-3 (I reproduce the plan output after the code), Oracle uses the INVENTORIES table as its driving table and uses the following execution path:

- SELECT STATEMENT
- HASH JOIN
- NESTED LOOPS
- TABLE ACCESS FULL INVENTORIES
- INDEX UNIQUE SCAN WAREHOUSES_PK
- INDEX FAST FULL SCAN PRD_DESC_PK

The plan output is as follows:

1. Oracle does a full table scan of the INVENTORIES table.
2. Oracle performs an index unique scan of the WAREHOUSES table using its primary key index.
3. Oracle performs a nested loop operation to join the rows from steps 1 and 2.
4. Oracle performs an index fast full scan of the product_descriptions table using its primary key, PRD_DESC_PK.
5. In the final step, Oracle performs a hash join of the set from step 3 and the rows resulting from the index full scan of step 4.

Using the output of the EXPLAIN PLAN, you can quickly see why some of your queries are taking much longer than anticipated. Armed with this knowledge, you can fine-tune a query until an acceptable performance threshold is reached. The wonderful thing about the EXPLAIN PLAN is that you never have to execute any statement in the database to trace the execution plan of the statement. The next section presents a few examples so you can feel more comfortable using the EXPLAIN PLAN utility.

**More Plan Examples**

In this section, you’ll learn how to interpret various kinds of execution plans derived by using the EXPLAIN PLAN utility.

In the first example, consider what happens when you use a function on an indexed column. Oracle completely ignores the index! As you can see, the optimizer can make mistakes. Good programmers can help the optimizer get it right by using methods such as proper indexing of tables, optimizer hints, and so on.
SQL> EXPLAIN PLAN set statement_id = 'example_plan1'
  2  FOR
  3  SELECT last_name FROM hr.employees
  4  WHERE upper(last_name) = 'FAY';

Explained.
SQL>

example_plan1
-----------------------------
SELECT STATEMENT
  TABLE ACCESS FULL EMPLOYEES
SQL>

The next example is a query similar to the preceding one, but without the upper function on last_name. This time, Oracle uses the index on the last_name column:

SQL> EXPLAIN PLAN SET statement_id = 'example_plan1'
  2  FOR
  3  SELECT last_name FROM hr.employees
  4  WHERE last_name='FAY';
Explained.

SQL>

example_plan1
-----------------------------
SELECT STATEMENT
  INDEX RANGE SCAN EMP_NAME_IX
SQL>

In the third example, two tables (customers and orders) are joined to retrieve the query results:

SQL> EXPLAIN PLAN SET statement_id 'newplan1'
  2  FOR
  3  SELECT o.order_id,
      4  o.order_total,
      5  c.account_mgr_id
      6  FROM customers c,
      7  orders o
      8  WHERE o.customer_id=c.customer_id
      9  AND o.order_date > '01-JUL-05'
Explained.
SQL>

Listing 19-4 shows the EXPLAIN PLAN from the plan table.

Listing 19-4. Another EXPLAIN PLAN Output

SQL> SELECT lpad(' ',level-1)||operation||' '||options||' '||object_name "newplan"
  2  FROM plan_table
  3  CONNECT BY prior id = parent_id
  4  AND prior statement_id = statement_id
  5* START WITH id = 0 AND statement_id = '&1'
    ORDER BY id;
Enter value for 1: newplan1
old 6:  START WITH id = 0 AND statement_id = '&1'
new  6:   START WITH id = 0 AND statement_id = 'newplan1'
newplan
SELECT STATEMENT
   HASH JOIN /* step 4 */
      TABLE ACCESS FULL CUSTOMERS /* step 2 */
      TABLE ACCESS BY INDEX ROWID ORDERS /* step 3 */
      INDEX RANGE SCAN ORD_ORDER_DATE_IX /* step 1 */
Elapsed: 00:00:00.01
SQL>

In step 1, the query first does an index range scan of the orders table using the ORD_ORDER_DATE_IX index. Why an index range scan? Because this index isn’t unique—it has multiple rows with the same data value—the optimizer has to scan these multiple rows to get the data it’s interested in. For example, if the indexed column is a primary key, it will be unique by definition, and you’ll see the notation “Unique Scan” in the EXPLAIN PLAN statement.

In step 2, the customers table is accessed through a full table scan, because account_manager_id in that table, which is part of the WHERE clause, isn’t indexed.

In step 3, the query accesses the orders table by INDEX ROWID, using the ROWID it derived in the previous step. This step gets you the order_id, customer_id, and order_total columns from the orders table for the date specified.

In step 4, the rows from the orders table are joined with the rows from the customers table based on the join condition WHERE o.customer_id=c.customer_id.

As you can see from the preceding examples, the EXPLAIN PLAN facility provides you with a clear idea as to the access methods used by the optimizer. Of course, you can do this without having to run the query itself. Often, the EXPLAIN PLAN will provide you with a quick answer as to why your SQL may be performing poorly. The plan’s output can help you determine how selective your indexes are and let you experiment with quick changes in code.

Using Autotrace

The Autotrace facility enables you to produce EXPLAIN PLANs automatically when you execute a SQL statement in SQL*Plus. You automatically have the privileges necessary to use the Autotrace facility when you log in as SYS or SYSTEM.

First, if you plan to use Autotrace, you should create a plan table in your schema. Once you create this plan table, you can use it for all your future executions of the Autotrace facility. If you don’t have this table in your schema, you’ll get an error when you try to use the Autotrace facility, as shown here:

```
SQL> SET AUTOTRACE ON
  . SP2-0618: Cannot find the Session Identifier
  . Check PLUSTRACE role is enabled
  SP2-0611: Error enabling STATISTICS report
SQL>
```

You can create the PLAN_TABLE table by using the CREATE TABLE statement, as shown in Listing 19-5. You can also create this table by executing the utlxplan.sql script, as I explained earlier.

**Listing 19-5. Manually Creating the Plan Table**

```
SQL> CREATE TABLE PLAN_TABLE(
   2  STATEMENT_ID     VARCHAR2(30), TIMESTAMP          DATE,
   3  REMARKS          VARCHAR2(80), OPERATION          VARCHAR2(30),
   4  OPTIONS          VARCHAR2(30), OBJECT_NODE        VARCHAR2(128),
   5  OBJECT_OWNER     VARCHAR2(30), OBJECT_NAME        VARCHAR2(30),
```
CHAPTER 19 ■ IMPROVING DATABASE PERFORMANCE: SQL QUERY OPTIMIZATION

Table created.

SQL>

Next, the SYS or SYSTEM user needs to grant you the PLUSTRACE role, as shown here:

SQL> GRANT PLUSTRACE TO salapati;
Grant succeeded.

If, as in the preceding case, the PLUSTRACE role doesn’t already exist in the database, the SYS user needs to run the plustrace.sql script, as shown in Listing 19-6, to create the PLUSTRACE role.

Listing 19-6. Creating the PLUSTRACE Role

SQL> @ORACLE_HOME/sqlplus/admin/plustrace.sql
SQL> DROP ROLE plustrace;
drop role plustrace
*ERROR at line 1:
ORA-01919: role 'PLUSTRACE' does not exist.

Third, the user who intends to use Autotrace should be given the PLUSTRACE role, as shown here:

SQL> GRANT plustrace TO salapati;
Grant succeeded.

The user can now set the Autotrace feature on and view the EXPLAIN PLAN for any query that is used in the session. The Autotrace feature can be turned on with different options:

• SET AUTOTRACE ON EXPLAIN: This generates the execution plan only and doesn’t execute the query itself.

• SET AUTOTRACE ON STATISTICS: This shows only the execution statistics for the SQL statement.

• SET AUTOTRACE ON: This shows both the execution plan and the SQL statement execution statistics.
All SQL statements issued after the Autotrace feature is turned on will generate the execution plans (until you turn off the Autotrace facility with the command `SET AUTOTRACE OFF`), as shown in Listing 19-7.

**Listing 19-7. Using the Autotrace Utility**

```sql
SQL> SET AUTOTRACE ON;
SQL> SELECT * FROM EMP;
no rows selected
```

**Execution Plan**

```text
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=74)
1    0   TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=1 Bytes=74)
```

**Statistics**

```text
0  recursive calls
0  db block gets
3  consistent gets
0  physical reads
0  redo size
511  bytes sent via SQL*Net to client
368  bytes received via SQL*Net from client
1  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
0  rows processed
```

```sql
SQL>
```

After showing the execution plan for the SQL statement, the Autotrace feature shows the details about the number of SQL recursive calls incurred in executing the original statement; the number of physical and logical reads, in memory and on disk sorts; and the number of rows processed.

I provide a few simple examples to show how Autotrace helps you optimize SQL queries. In the following examples, the same query is used twice in the courses table, once without an index and once with an index. After the table is indexed, you run the query before you analyze the table. The results are instructive.

In the first example, whose output is shown in Listing 19-8, you run the test query before you create an index on the courses table.

**Listing 19-8. The Execution Plan for a Query Without an Index**

```sql
SQL> SET AUTOTRACE ON
SQL> SELECT COUNT(*) FROM courses
    2  WHERE course_subject='medicine'
    3* AND course_title = 'fundamentals of human anatomy';
COUNT(*)
98304
```

**Execution Plan**

```text
0      SELECT STATEMENT Optimizer=CHOOSE
1    0   SORT (AGGREGATE)
2    1     TABLE ACCESS (FULL) OF 'COURSES'
```

**Statistics**

```text
0  recursive calls
0  db block gets
753  consistent gets
```
As you can see, the query used a full table scan because there are no indexes. There were a total of 338 physical reads. Note that the total number of rows in the courses table is 98,384. Out of this total, the courses with medicine as the course subject were 98,304. That is, the table values aren’t distributed evenly among the courses at all. Now let’s see what happens when you use an index.

The following example uses a query with an index. However, no statistics are collected for either the table or the index. When you create an index on the courses table and run the same query, you’ll see some interesting results. Listing 19-9 tells the story.

Listing 19-9. The Execution Plan for a Query with an Index

```sql
SQL> CREATE INDEX title_idx ON courses (course_title);
Index created.
SQL> SELECT count(*) FROM courses
2  WHERE course_subject='medicine'
3  AND course_title = 'fundamentals of human anatomy';
COUNT(*) 98304
```

After you created the index, the physical reads went from 338 to 1,249! The EXPLAIN PLAN shows that Oracle is indeed using the index, so you would expect the physical reads to be lower when compared to the no-index case. What happened here is that even if a table has an index, this doesn’t mean that it’s always good to use it under all circumstances. The CBO always figures the best way to get a query’s results, with or without using the index. In this case, the query has to look at almost all the rows of the table, so using an index isn’t the best way to go. However, you haven’t collected statistics for the table and the index, so Oracle has no way of knowing the distribution of the actual data.
in the courses table. Lacking any statistics, it falls back to a rule-based approach. Under a rule-based optimization, using an index occupies a lower rank and therefore indicates that this is the optimal approach here. Let’s see the results after analyzing the table.

The third example is a query with an index executed after collecting optimizer statistics for the table. Oracle has the complete statistics, and it uses the CBO this time around. The CBO decides to use an index only if the cost of using the index is lower than the cost of a full table scan. The CBO decides that it won’t use the index, because the query will have to read 98,304 out of a total of 98,384 rows. It rightly decides to do a full table scan instead. The results are shown in Listing 19-10.

Listing 19-10. The Execution Plan with an Index After Analyzing the Table

```sql
SQL> ANALYZE TABLE courses COMPUTE STATISTICS;
Table analyzed.
SQL> SELECT count(*) FROM courses
     2 WHERE course_subject='medicine'
     3 AND course_title = 'fundamentals of human anatomy';
COUNT(*)  
----------
     98304
Execution Plan
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE (Cost=74 Card=1 Bytes=39)
1 0    SORT (AGGREGATE)
2 1    TABLE ACCESS (FULL) OF 'COURSES' (Cost=74 Card=24596 Bytes=959244)
Statistics
----------------------------------------------------------
 290  recursive calls
    0  db block gets
 792  consistent gets
 334  physical reads
    0  redo size
 381  bytes sent via SQL*Net to client
 499  bytes received via SQL*Net from client
     2  SQL*Net roundtrips to/from client
   6  sorts (memory)
    0  sorts (disk)
    1  rows processed
SQL>
```

In this listing, the first item, recursive calls, refers to additional statements Oracle needs to make when it’s processing a user’s SQL statement. For example, Oracle issues recursive calls (or recursive SQL statements) to make space allocations or to query the data dictionary tables on disk. In our example, Oracle made 290 internal calls during the SQL Trace period.

Using SQL Trace and TKPROF

SQL Trace is an Oracle utility that helps you trace the execution of SQL statements. TKPROF is another Oracle utility that helps you format the trace files output by SQL Trace into a readable form. Although the EXPLAIN PLAN facility gives you the expected execution plan, the SQL Trace tool gives you the actual execution results of a SQL query. Sometimes, you may not be able to identify the exact code, say, for dynamically generated SQL. SQL Trace files can capture the SQL for dynamic SQL. Among other things, SQL Trace enables you to track the following variables:
• CPU and elapsed times
• Parsed and executed counts for each SQL statement
• Number of physical and logical reads
• Execution plan for all the SQL statements
• Library cache hit ratios

Tip If your application has a lot of dynamically generated SQL, the SQL Trace utility is ideal for tuning the SQL statements.

Although the EXPLAIN PLAN tool is important for determining the access path that the optimizer will use, SQL Trace gives you a lot of hard information on resource use and the efficacy of the statements. You’ll get a good idea of whether your statement is being parsed excessively. The statement’s execute and fetch counts illustrate its efficiency. You get a good sense of how much CPU time is consumed by your queries and how much I/O is being performed during the execution phase. This helps you identify the resource-guzzling SQL statements in your application and tune them. The EXPLAIN PLAN, which is an optional part of SQL Trace, gives the row counts for the individual steps of the EXPLAIN PLAN, helping you pinpoint at what step the most work is being done. By comparing resource use with the number of rows fetched, you can easily determine how productive a particular statement is.

In the next sections you’ll use SQL Trace to trace a simple SQL statement and interpret it with the TKPROF utility. You start by setting a few initialization parameters to ensure tracing.

Setting the Trace Initialization Parameters

Collecting trace statistics imposes a performance penalty, and consequently the database doesn’t automatically trace all sessions. Tracing is purely an optional process that you turn on for a limited duration to capture metrics about the performance of critical SQL statements. You need to look at four initialization parameters to set up Oracle correctly for SQL tracing, and you have to restart the database after checking that the following parameters are correctly configured. Three of these parameters are dynamic session parameters, and you can change them at the session level.

STATISTICS_LEVEL

The STATISTICS_LEVEL parameter can take three values. The value of this parameter has a bearing on the TIMED_STATISTICS parameter. You can see this dependency clearly in the following summary:

• If the STATISTICS_LEVEL parameter is set to TYPICAL or ALL, timed statistics are collected automatically for the database.
• If STATISTICS_LEVEL is set to BASIC, then TIMED_STATISTICS must be set to TRUE for statistics collection.
• Even if STATISTICS_LEVEL is set to TYPICAL or ALL, you can keep the database from tracing by using the ALTER SESSION statement to set TIMED_STATISTICS to FALSE.

TIMED_STATISTICS

The TIMED_STATISTICS parameter is FALSE by default, if the STATISTICS_LEVEL parameter is set to BASIC. In a case like this, to collect performance statistics such as CPU and execution time, set the value of the TIMED_STATISTICS parameter to TRUE in the init.ora file or SPFILE, or use the ALTER
SYSTEM_SET_TIMED_STATISTICS=TRUE statement to turn timed statistics on instance-wide. You can also do this at the session level by using the ALTER SESSION statement as follows:

```sql
SQL> ALTER SESSION SET timed_statistics = true;
Session altered.
SQL>
```

**USER_DUMP_DEST**

USER_DUMP_DEST is the directory on your server where your SQL Trace files will be sent. By default you use the `$ORACLE_HOME/admin/database_name/udump` directory as your directory for dumping SQL trace files. If you want non-DBAs to be able to read this file, make sure the directory permissions authorize reading by others. Alternatively, you can set the parameter `TRACE_FILES_PUBLIC=TRUE` to let others read the trace files on UNIX systems. Make sure the destination points to a directory that has plenty of free space to accommodate large trace files. USER_DUMP_DEST is a dynamic parameter, so you can also change it using the ALTER SYSTEM command, as follows:

```sql
SQL> ALTER SYSTEM SET user_dump_dest='c:\oracletnt\oradata';
System altered.
SQL>
```

---

**Note** In Oracle Database 11g, if you set the new DIAGNOSTIC_DEST initialization parameter, the database ignores the USER_DUMP_DEST setting. The directory you set for the DIAGNOSTIC_DEST parameter determines where the database will place the trace files.

---

**MAX_DUMP_FILE_SIZE**

Some traces could result in large trace files in a big hurry, so make sure your MAX_DUMP_FILE_SIZE initialization parameter is set to a high number. The default size of this parameter may be too small for some traces. If the trace fills the dump file, it won’t terminate, but the information in the file will be truncated.

**Enabling SQL Trace**

To use SQL Trace and TKPROF, first you need to enable the Trace facility. You can do this at the instance level by using the ALTER SESSION statement or the DBMS_SESSION package. You can trace the entire instance by either including the line `SQL_TRACE=TRUE` in your `init.ora` file or `SPFILE` or by using the ALTER SYSTEM command to set `SQL_TRACE` to `TRUE`. Tracing the entire instance isn’t recommended, because it generates a huge amount of tracing information, most of which is useless for your purpose. The statement that follows shows how to turn tracing on from your session using the ALTER SESSION statement:

```sql
SQL> ALTER SESSION SET sql_trace=true;
Session altered.
SQL>
```

The following example shows how you set `SQL_TRACE` to `TRUE` using the DBMS_SESSION package:

```sql
SQL> EXECUTE sys.dbms_session.set_sql_trace(true);
PL/SQL procedure successfully completed.
SQL>
```
Often, users request the DBA to help them trace their SQL statements. You can use the DBMS_ SYSTEM.SET_SQL_TRACE_IN_SESSION procedure to set tracing on in another user's session. Note that usage of the DBMS_SYSTEM package has never actually been supported by Oracle. The recommended way is to use the DBMS_MONITOR package to trace a session. Regardless of the method you use, once you start tracing a session, all statements are traced until you use the ALTER_SESSION statement or the DBMS_SESSION package to turn tracing off (replace true with false in either of the preceding statements). Alternatively, when the user logs off, tracing is automatically stopped for that user.

Interpreting the Trace Files with TKPROF

Once you set tracing on for a session, any SQL statement that is issued during that session is traced and the output stored in the directory (udump) specified by the USER_DUMP_DEST parameter in your init.ora file or SPFILE. The filename has the format db_name_ora_nnnnn.trc, where nnnnn is usually a four- or five-digit number. For example, the sample trace file in our example is named pasx_ora_16340.trc. If you go to the user dump destination directory immediately after a trace session is completed, the most recent file is usually the session trace file you just created.

You can also differentiate the trace file output by a SQL Trace execution from the other files in the dump directory, by its size—these trace files are much larger in general than the other files output to the directory. These trace files are detailed and complex. Fortunately, the easy-to-run TKPROF utility formats the output into a readable format. The TKPROF utility uses the trace file as the input, along with several parameters you can specify.

Table 19-1 shows the main TKPROF parameters you can choose to produce the format that suits you. If you type tkprof at the command prompt, you'll see a complete listing of all the parameters that you can specify when you invoke TKPROF.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME</td>
<td>The input trace file produced by SQL Trace</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>The EXPLAIN PLAN for the SQL statements</td>
</tr>
<tr>
<td>RECORD</td>
<td>Creates a SQL script with all the nonrecursive SQL statements</td>
</tr>
<tr>
<td>WAITS</td>
<td>Records a summary of wait events</td>
</tr>
<tr>
<td>SORT</td>
<td>Presents sort data based on one or more items, such as PRSCPU (CPU time parsing), PRSELA (elapsed time parsing), and so on</td>
</tr>
<tr>
<td>TABLE</td>
<td>Defines the name of the tables into which the TKPROF utility temporarily puts the execution plans</td>
</tr>
<tr>
<td>SYS</td>
<td>Enables and disables listing of SQL statements issued by SYS</td>
</tr>
<tr>
<td>PRINT</td>
<td>Lists only a specified number of SQL statements instead of all statements</td>
</tr>
<tr>
<td>INSERT</td>
<td>Creates a script that stores the trace information in the database</td>
</tr>
</tbody>
</table>

Let’s trace a session by a user who is executing two SELECT statements, one using tables with indexes and the other using tables without any indexes. In this example, you’re using only a few parameters, choosing to run TKPROF with default sort options. The first parameter is the name of the output file and the second is the name for the TKPROF-formatted output. You’re specifying that you don’t want any analysis of statements issued by the user SYS. You’re also specifying that the EXPLAIN PLAN for the statement be shown in addition to the other statistics.
Tip By just typing `tkprof` at the operating system prompt, you can get a quick help guide to the usage of the TKPROF utility.

```bash
$ tkprof finance_ora_16340.trc test.txt sys=no explain=y
```


Copyright (c) 1982, 2007, Oracle. All rights reserved.

The `test.txt` file contains the output of the SQL trace, now nicely formatted for you by the TKPROF utility.

Examining the Formatted Output File

Listing 19-11 shows the top portion of the `test.txt` file, which explains the key terms used by the utility.

**Listing 19-11. The Top Part of the TKPROF-Formatted Trace File**


Copyright (c) 1982, 2007, Oracle. All rights reserved.
Trace file: finance_ora_16340.trc
Sort options: default

| 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 |

Each TKPROF report shows the following information for each SQL statement issued during the time the user’s session was traced:

- The SQL statement
- Counts of parse, execute, and fetch (for SELECT statements) calls
- Count of rows processed
- CPU seconds used
- I/O used
- Library cache misses
- Optional execution plan
- Row-source operation listing
- A report summary analyzing how many similar and distinct statements were found in the trace file
Let’s analyze the formatted output created by TKPROF. Listing 19-12 shows the parts of the TKPROF output showing the parse, execute, and fetch counts.

**Listing 19-12. The Parse, Execute, and Fetch Counts**

```sql
SQL> select e.last_name,e.first_name,d.department_name
  2  from teste e,testd d
  3  where e.department_id=d.department_id;
```

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>17322</td>
<td>1.82</td>
<td>1.85</td>
<td>3</td>
<td>136</td>
<td>5</td>
<td>259806</td>
</tr>
</tbody>
</table>

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 53

In Listing 19-12

- CPU stands for total CPU time in seconds.
- Elapsed is the total time elapsed in seconds.
- Disk denotes total physical reads.
- Query is the number of consistent buffer gets.
- Current is the number of database block gets.
- Rows is the total number of rows processed for each type of call.

From Listing 19-12, you can draw the following conclusions:

- The SQL statement shown previously was parsed once, so a parsed version wasn’t available in the shared pool before execution. The Parse column shows that this operation took less than 0.01 seconds. Note that the lack of disk I/Os and buffer gets indicates that there were no data dictionary cache misses during the parse operation. If the Parse column showed a large number for the same statement, it would be an indicator that bind variables weren’t being used.
- The statement was executed once and execution took less than 0.01 seconds. Again, there were no disk I/Os or buffer gets during the execution phase.
- It took me a lot longer than 0.01 seconds to get the results of the SELECT statement back. The Fetch column answers this question of why that should be: it shows that the operation was performed 17,324 times and took up 1.82 seconds of CPU time.
- The Fetch operation was performed 17,324 times and fetched 259,806 rows. Because the number of rows is far greater than the number of fetches, you can deduce that Oracle used array fetch operations.
- There were three physical reads during the fetch operation. If there’s a large difference between CPU time and elapsed time, it can be attributed to time taken up by disk reads. In this case, the physical I/O has a value of only 3, and it matches the insignificant gap between CPU time and elapsed time. The fetch required 136 buffer gets in the consistent mode and only 5 DB block gets.
- The CBO was being used, because the optimizer goal is shown as CHOOSE.
The following output shows the execution plan that was explicitly requested when TKPROF was invoked. Note that instead of the cost estimates that you get when you use the EXPLAIN PLAN tool, you get the number of rows output by each step of the execution.

<table>
<thead>
<tr>
<th>Rows</th>
<th>Row Source Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>259806</td>
<td>MERGE JOIN</td>
</tr>
<tr>
<td>1161</td>
<td>SORT JOIN</td>
</tr>
<tr>
<td>1161</td>
<td>TABLE ACCESS FULL TESTD</td>
</tr>
<tr>
<td>259806</td>
<td>SORT JOIN</td>
</tr>
</tbody>
</table>

Finally, TKPROF summarizes the report, stating how many SQL statements were traced. Here's the summary portion of the TKPROF-formatted output:

```
Trace file: ORA02344.TRC
Trace file compatibility: 9.00.01
Sort options: default
  2  sessions in trace file.
  18 user  SQL statements in trace file.
  104 internal SQL statements in trace file.
  72  SQL statements in trace file.
  33 unique SQL statements in trace file.
 18182 lines in trace file.
```

The TKPROF output makes it easy to identify inefficient SQL statements. TKPROF can order the SQL statements by elapsed time (time taken for execution), which tells you which of the SQL statements you should focus on for optimization.

The SQL Trace utility is a powerful tool in tuning SQL, because it goes far beyond the information produced by using EXPLAIN PLAN. It provides you with hard information about the number of the various types of calls made to Oracle during statement execution, and how the resource use was allocated to the various stages of execution.

**Note** It’s easy to trace individual user sessions using the OEM Database Control. I explain how you can trace and view user sessions using the Database Control in the section “Using the Database Control for End-to-End Tracing.” You can trace a session as well as read the output file directly from the Database Control.

## End-to-End Tracing

In multitier environments, the middle tier passes a client’s request through several database sessions. It’s hard to keep track of the client across all these database sessions. Similarly, when you use shared server architecture, it’s hard to identify the user session that you’re tracing at any given time. Because multiple sessions may use the same shared server connection, when you trace the connection, you can’t be sure who the user is exactly at any given time—the active sessions using the shared server connection keep changing throughout.

In the cases I described earlier, tracing a single session becomes impossible. Oracle Database 10g introduced *end-to-end tracing*, with which you can uniquely identify and track the same client through multiple sessions. The attribute `CLIENT_IDENTIFIER` uniquely identifies a client and remains the same through all the tiers. You can use the DBMS_MONITOR package to perform end-to-end
tracing. You can also use the OEM Database Control to set up end-to-end tracing easily. Let’s look at both approaches in the following sections.

### Using the DBMS_MONITOR Package

You use the Oracle PL/SQL package DBMS_MONITOR to set up end-to-end tracing. You can trace a user session through multiple tiers and generate trace files using the following three attributes:

- Client identifier
- Service name
- Combination of service name, module name, and action name

You can specify a combination of service name, module name, and action name. You can also specify service name alone, or a combination of service name and module name. However, you can’t specify an action name alone. Your application must use the DBMS_APPLICATION_INFO package to set module and action names. The service name is determined by the connect string you use to connect to a service. If a user’s session isn’t associated with a service specifically, the sys$users service handles it.

Let’s use two procedures belonging to the DBMS_MONITOR package. The first one, SERV_MOD_ACT_TRACE_ENABLE, sets the service name, module name, and action name attributes. The second, CLIENT_ID_TRACE_ENABLE, sets the client ID attribute. Here’s an example:

```sql
SQL> EXECUTE dbms_monitor.serv_mod_act_trace_enable
   2   (service_name=>'myservice', module_name=>'batch_job');
PL/SQL procedure successfully completed.
SQL> EXECUTE dbms_monitor.client_id_trace_enable
   2   (client_id=>'salapati');
PL/SQL procedure successfully completed.
```

You can use the SET_IDENTIFIER procedure of the DBMS_SESSION package to get a client’s session ID. Here’s an example showing how you can use a logon trigger and the SET_IDENTIFIER procedure together to capture the user’s session ID immediately upon the user’s logging into the system:

```sql
SQL> CREATE OR REPLACE TRIGGER logon_trigger
   2  AFTER LOGON
   3  ON DATABASE
   4  DECLARE
   5     user_id VARCHAR2(64);
   6  BEGIN
   7     SELECT ora_login_user ||':'||SYS_CONTEXT('USERENV','OS_USER')
   8        INTO user_id
   9        FROM dual;
  10     dbms_session.set_identifier(user_id);
  11  END;
```

Using the value for the client_id attribute, you can get the values for the SID and SERIAL# columns in the V$SESSION view for any user and set up tracing for that client_id. Here’s an example:
SQL> EXECUTE dbms_monitor.session_trace_enable
  (session_id=>111, serial_num=>23, waits=>true, binds=>false);

You can now ask the user to run the problem SQL and collect the trace files so you can use the
TKPROF utility to analyze them. In a shared server environment especially, there may be multiple
trace files. By using the trcsess command-line tool, you can consolidate information from multiple
trace files into one single file. Here’s an example (first navigate to your user dump or udump directory):

$ trcsess output="salapati.trc" service="myservice
  "module="batch job" action="batch insert"

You can then run your usual TKPROF command against the consolidated trace file, as shown
here:

$ tkprof salapati.trc output=salapati_report SORT=(EXEELA, PRSELA, FCHELA)

---

**Note** In this chapter, you saw how to enable SQL tracing using the SQL Trace facility, the DBMS_SESSION
package, and the DBMS_MONITOR package. You should use one of the two packages, rather than SQL Trace, to
trace SQL statements. You can use any one of these three methods to set up a session-level or instance-wide trace.
Be careful about tracing the entire instance, because it’ll lead to excessive load on your instance, as well as produce
too many voluminous trace files.

---

**Using the Database Control for End-to-End Tracing**

The best approach, as well as the recommended one, to end-to-end tracing is to use the OEM Data-
base Control. This way, you don’t have to bother with manual runs of the DBMS_MONITOR
package. Here are the steps:

1. From the Database Control home page, click the Performance link.
2. In the Performance page, click the Top Consumers link under the Additional Management
   Links section.
3. In the Top Consumers page, you’ll see the tabs for Top Services, Top Modules, Top Actions,
   Top Clients, and Top Sessions, as shown in Figure 19-2. Click the Top Clients tab.
4. To enable aggregation for a client, select the client and click Enable Aggregation.

If you wish, you can use the Database Control to trace a normal SQL session instead of using the
SET_TRACE command and the TKPROF utility. To trace a user command, in step 3 of the preceding
sequence, click the Top Sessions tab. You then click the Enable SQL Trace button. You can then use
the Disable SQL Trace button to stop the session tracing and view the output by clicking the View
SQL Trace File button.

---

**Note** You can view all outstanding trace information in your instance by examining the DBA_ENABLED_TRACES
view, or use a trace report generated through the Database Control.
Using the V$SQL View to Find Inefficient SQL

The V$SQL view is an invaluable tool in tracking down wasteful SQL code in your application. The V$SQL view gathers information from the shared pool area on every statement’s disk reads and memory reads, in addition to other important information. The view holds all the SQL statements executed since instance startup, but there’s no guarantee that it will hold every statement until you shut down the instance. For space reasons, the older statements are aged out of the V$SQL view. It’s a good idea for you to grant your developers select rights on this view directly if you haven’t already granted them the “select any catalog” role. You can use the V$SQL view to perform ad hoc queries on disk and CPU usage, but remember that the AWR report includes summaries of these kinds of information.

The V$SQL view includes, among other things, the following columns, which help in assessing how many resources a SQL statement consumes:

- **rows_processed** gives you the total number of rows processed by the statement.
- **sql_text** is the text of the SQL statement (first 1,000 characters).
- **sql_fulltext** is a CLOB column that shows the full text of a SQL statement.
- **buffer_gets** gives you the total number of logical reads (indicates high CPU use).
- **disk_reads** tells you the total number of disk reads (indicates high I/O).
- **sorts** gives the number of sorts for the statement (indicates high sort ratios).
- **cpu_time** is the total parse and execution time.
- **elapsed_time** is the elapsed time for parsing and execution.
• `parse_calls` is the combined soft and hard parse calls for the statement.
• `executions` is the number of times a statement was executed.
• `loads` is the number of times the statement was reloaded into the shared pool after being flushed out.
• `sharable_memory` is the total shared memory used by the cursor.
• `persistent_memory` is the total persistent memory used by the cursor.
• `runtime_memory` is the total runtime memory used by the cursor.

**Note** In previous versions, DBAs used the V$SQLAREA view to gather information shown earlier. However, the V$SQL view supplants the V$SQLAREA view by providing all information in that view, plus other important tuning-related information as well.

### Finding SQL That Uses Most of Your Resources

You can query the V$SQL view to find high-resource-using SQL. You can determine resource-intensive SQL on the basis of the number of logical reads or buffer gets, or high disk reads, high parse calls, large number of executions, or combinations of these factors. It’s obvious that a high number of disk reads is inefficient because a high amount of physical I/O slows query performance. However, a high number of memory reads (buffer gets) is also expensive because they consume CPU resources. You normally have high buffer gets because you’re using the wrong index, the wrong driving table in a join, or a similar SQL-related error. One of the primary goals of SQL tuning should be to lower the number of unnecessary logical reads. If buffer gets and disk reads are at identical levels, it could indicate a missing index. The reasoning is this: if you don’t have an index, Oracle is forced to do a full table scan. However, full table scans can’t be kept in the SGA for too long because they might force a lot of other data to be cleared out. Consequently, the full table won’t get to stay in the SGA for too long unless it’s a small table.

The following simple query shows how the V$SQL view can pinpoint problem SQL statements; both high disk reads and high logical reads are used as the criteria for flagging down poor SQL statements captured by the V$SQL view. The SQL_TEXT column shows the exact SQL statement that’s responsible for the high disk reads and logical reads:

```sql
SQL> SELECT sql_text, executions, buffer_gets, disk_reads,
FROM V$SQL
WHERE buffer_gets > 100000
OR disk_reads > 100000
ORDER BY buffer_gets + 100*disk_reads DESC;
```

<table>
<thead>
<tr>
<th>SQL_TEXT</th>
<th>EXECUTIONS</th>
<th>BUFFER_GETS</th>
<th>DISK_READS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN dbms_job.run(1009133);</td>
<td>726216</td>
<td>1615283234</td>
<td>125828</td>
</tr>
<tr>
<td>BEGIN label_sc.pkg.launch_sc;</td>
<td>34665</td>
<td>1211625422</td>
<td>3680242</td>
</tr>
<tr>
<td>SELECT COUNT(*) AV_YOUTHS...</td>
<td>70564</td>
<td>152737737</td>
<td>7186125</td>
</tr>
<tr>
<td>SELECT UC.CHART_ID...</td>
<td>37849</td>
<td>96590083</td>
<td>5547319</td>
</tr>
<tr>
<td>SELECT MAX(REC_NUM) FROM...</td>
<td>5163242</td>
<td>33272842</td>
<td>6034715</td>
</tr>
</tbody>
</table>

The following query is a slight variation on the preceding query. It seeks to find out the number of rows processed for each statement:

```sql
SQL> SELECT sql_text, executions, buffer_gets, disk_reads,
FROM V$SQL
WHERE buffer_gets > 100000
OR disk_reads > 100000
ORDER BY buffer_gets + 100*disk_reads DESC;
```
SQL> SELECT sql_text, rows_processed,  
  2  buffer_gets, disk_reads, parse_calls  
  3 FROM V$SQL  
  4 WHERE buffer_gets > 100000  
  5 OR disk_reads > 100000  
  6*ORDER BY buffer_gets + 100*disk_reads DESC;

<table>
<thead>
<tr>
<th>SQL_TEXT</th>
<th>ROWS_PROCESSED</th>
<th>BUFFER_GETS</th>
<th>DISK_READS</th>
<th>PARSE_CALLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN dbms_job.run(1009133);</td>
<td>9659</td>
<td>1615322749</td>
<td>125830</td>
<td>2078</td>
</tr>
<tr>
<td>BEGIN label_sc_pkg.launch_sc;</td>
<td>3928</td>
<td>1214405479</td>
<td>3680515</td>
<td>4</td>
</tr>
<tr>
<td>SELECT COUNT(*) AV_YOUTHS...</td>
<td>70660</td>
<td>152737737</td>
<td>7186125</td>
<td>3863</td>
</tr>
<tr>
<td>SELECT UC.CHART_ID...</td>
<td>37848</td>
<td>96590083</td>
<td>5547319</td>
<td>5476</td>
</tr>
<tr>
<td>SELECT MAX(REC_NUM) FROM...</td>
<td>5163236</td>
<td>33272842</td>
<td>6034715</td>
<td>606</td>
</tr>
</tbody>
</table>

The V$SQL view helps you find out which of your queries have high logical I/O (LIO) and high physical I/O (PIO). By also providing the number of rows processed by each statement, it tells you whether a statement is efficient or not. By providing the disk reads and the number of executions per statement, the view helps you determine whether the number of disk reads per execution is reasonable. If CPU usage is your concern, look at the statements that have a high number of buffer gets. If I/O is your primary concern, focus on the statements that perform the most disk reads. Once you settle on the statement you want to investigate further, examine the complete SQL statement and see whether you (or the developers) can improve it.

One of the best ways to find poorly performing SQL queries is by using the Oracle wait interface, which I explain in detail in Chapter 20.

Here’s a query that uses the V$SQL view to sort the top five queries that are taking the most CPU time and the most elapsed time to complete:

SQL> SELECT sql_text, executions,  
  2  ROUND(elapsed_time/1000000, 2) elapsed_seconds,  
  3  ROUND(cpu_time/1000000, 2) cpu_secs from  
  4  (select * from v$sql order by elapsed_time desc)  
  5* WHERE rownum <6;

<table>
<thead>
<tr>
<th>SQL_TEXT</th>
<th>EXECUTIONS</th>
<th>ELAPSED_SECONDS</th>
<th>CPU_SECS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE MS_DASH_TRANLOGS...</td>
<td>2283</td>
<td>44.57</td>
<td>43.04</td>
</tr>
<tr>
<td>UPDATE PERSONS SET...</td>
<td>14132</td>
<td>19.74</td>
<td>20.52</td>
</tr>
<tr>
<td>SELECT /*+ INDEX(ud)...</td>
<td>9132</td>
<td>9.95</td>
<td>9</td>
</tr>
<tr>
<td>SELECT PROG_ID FROM UNITS ...</td>
<td>14132</td>
<td>5.26</td>
<td>5.81</td>
</tr>
<tr>
<td>SELECT NVL(SUM(RECHART),0)...</td>
<td>2284</td>
<td>4.13</td>
<td>4.43</td>
</tr>
</tbody>
</table>

Using Other Dictionary Views for SQL Tuning

The V$SQL_PLAN and V$SQL_PLAN_STATISTICS views are highly useful for tracking the efficiency of execution plans. You should be wary of quick changes in code to fix even the worst-performing query in the system. Let’s say you create an extra index on a table or change the order of columns in a composite key to fix this problem query. How do you know these aren’t going to impact other queries in the application adversely? This happens more often than you think, and therefore you must do your due diligence to rule out unintended consequences of your fixes.
The SQL Tuning Advisor

You can use the SQL Tuning Advisor to improve poorly performing SQL statements. The SQL Tuning Advisor provides the following to help you tune bad SQL statements:

- Advice on improving the execution plan
- Reasons for the SQL improvement recommendations
- Benefits you can expect by following the Advisor’s advice
- Details of the commands to tune the misbehaving SQL statements

Using the SQL Tuning Advisor

The SQL Tuning Advisor can use the following sources:

- New SQL statements. When working with a development database, this may be your best source of SQL statements.
- High-load SQL statements.
- SQL statements from the AWR.
- SQL statements from the database cursor cache.

The Advisor can tune sets of SQL statements called SQL Tuning Sets. An STS is a set of SQL statements combined with execution information, which includes the average elapsed time. An STS has the advantage of capturing the information about a database’s workload as well as allowing you to tune several large SQL statements at once.

How the SQL Tuning Advisor Works

As mentioned previously, the optimizer will try to find the optimal execution plan for each statement you provide. However, this process happens under production conditions, so the optimizer can only devote a short amount of time to working out a solution. The optimizer uses heuristics to generate an estimate of the best solution. This is called the normal mode of the optimizer.

You can also run the optimizer in tuning mode, which means that the optimizer carries out in-depth analysis to come up with ways to optimize execution plans. While in this mode, the optimizer can take several minutes and produces recommendations instead of the best SQL execution plan. You, in turn, use these recommendations to optimize the SQL statements’ execution plans. You get the added advantage of advice that details the rationale behind the recommendations and what you will gain from implementing them. The Oracle optimizer running in tuning mode is called the Automatic Tuning Optimizer (ATO). The ATO does the following tasks:

- Statistics analysis
- SQL profiling
- Access path analysis
- SQL structure analysis

I describe each of these tasks in the following sections, along with the types of recommendations that the SQL Tuning Advisor makes.
Statistics Analysis

The ATO makes sure that there are representative, up-to-date statistics for all the objects in the SQL statement, which you need for efficient execution plans. If the ATO finds any statistics that are missing or stale, it suggests that you collect new statistics for the objects in question. During this process, the ATO collects other information that it can use to fill in any missing statistics. It can also correct stale statistics.

SQL Profiling

At this stage the ATO tries to verify the validity of its estimates of factors such as column selectivity and cardinality of database objects. It can use three methods to verify its estimates:

- *Dynamic data sampling*: The ATO can use a data sample to check its estimates. The ATO can apply correction factors if the data-sampling process shows its estimates to be significantly wrong.
- *Partial execution*: The ATO can carry out the partial execution of a SQL statement. This process allows it to check whether its estimates are close to what really happens. It does not check whether its estimates are correct, but rather it checks whether a plan derived from those statistics is the best possible plan.
- *Past execution history statistics*: The ATO can use the SQL statement’s execution history to help with its work.

If there’s enough information from statistics analysis or SQL profiling, the ATO suggests you create a SQL profile, which is supplementary information about a SQL statement.

If you accept this advice and are running the optimizer in tuning mode, Oracle will store the SQL profile in the data dictionary. Once you have done this, the optimizer uses it to produce optimal execution plans, even when it is running in normal mode.

**Tip** Remember that a SQL profile is not the same thing as a stored execution plan.

The SQL profile will continue to apply if you make small changes to your database and allow your objects to grow normally. One of the big advantages of SQL profiles is the ability to tune packaged applications. These are hard to tune because you can’t easily access and modify the code. Because SQL profiles are saved in the data dictionary, you can use them to tune packaged applications.

Analyzing Access Paths

The ATO analyzes how using an improved access method, such as working with an index, will affect queries. These are important considerations, because adding an index can substantially increase the speed of a query. However, adding new indexes can adversely affect other SQL statements; the SQL Advisor knows this and makes its recommendations as follows:

- If an index is effective, it will advise you to create it.
- It can advise you to run the SQL Access Advisor (see Chapter 7 for details) to analyze the wisdom of adding the new index.
SQL Structure Analysis

The ATO can make recommendations to modify the structure (both the syntax and semantics) of poorly performing SQL statements. The ATO considers issues such as the following:

- Design mistakes; for example, performing full table scans because you didn't create indexes.
- Using inefficient SQL; for example, the `NOT IN` construct, which is known to be much slower than the `NOT EXISTS` construct in general.

**Note** The ATO only identifies poorly written SQL, but it won’t rewrite it for you. You will know your application better than the ATO, so Oracle only provides advice, which you can implement or not.

Recommendations

Here are some recommendations that the SQL Tuning Advisor will give you:

- Creating indexes will speed up access paths.
- Using SQL profiles will allow you to generate a better execution plan.
- Gathering optimizer statistics for objects that do not have any, or renewing stale statistics, will be of benefit.
- Rewriting SQL as advised will improve its performance.

The SQL Tuning Advisor in Practice

You can use the SQL Tuning Advisor through packages or through the web interface of the OEM Database Control.

Using the DBMS_SQLTUNE Package to Run the SQL Tuning Advisor

The main SQL package for tuning SQL statements is DBMS_SQLTUNE. The first example will be creating and managing tasks that tune SQL statements.

**Note** You must have the ADVISOR privilege to use the DBMS_SQLTUNE package. Ensure that you do before running any of the following examples.

Performing Automatic SQL Tuning

Here’s how to tune SQL statements using the DBMS_SQLTUNE package:

1. Create a task: The CREATE_TUNING_TASK procedure creates a task to tune a single statement or several statements (a SQL tuning set or STS). You can also use a SQL statement (using the SQL identifier) from the AWR or from the cursor cache. In the following example, I show how to create a task using a single SQL statement as input. First, I pass the SQL statement as a CLOB argument, as shown here:
DECLARE
  my_task_name VARCHAR2(30);
  my_sqltext   CLOB;
BEGIN
  my_sqltext := 'SELECT /*+ ORDERED */ *
                 FROM employees e, locations l, departments d
                 WHERE e.department_id = d.department_id AND
                       l.location_id = d.Location_id AND
                       e.employee_id < :bnd';

  Next, I create the following tuning task:

  my_task_name := DBMS_SQLTUNE.CREATE_TUNING_TASK(
                  sql_text => my_sqltext,
                  bind_list => sql_binds(anydata.ConvertNumber(90)),
                  user_name   => 'HR',
                  scope       => 'COMPREHENSIVE',
                  time_limit  => 60,
                  task_name   => 'my_sql_tuning_task',
                  description => 'Task to tune a query on a specified employee');
END;
/

In the preceding task, sql_text refers to the single SQL statement that I'm tuning. The bind_list shows that 90 is the value of the bind variable bnd. The tuning task's scope is comprehensive, meaning that it analyzes the SQL Profile, and the task_limit parameter sets a limit of 60 seconds on the total time for analysis.

2. Execute the task: To execute the task, run the EXECUTE_TUNING_TASK procedure:

  BEGIN
    DBMS_SQLTUNE.EXECUTE_TUNING_TASK( task_name => 'my_sql_tuning_task' );
  END;
/

3. Get the tuning report: You can view the tuning process with the REPORT_TUNING_TASK procedure:

  SQL> SET LONG 1000
  SQL> SET LONGCHUNKSIZE 1000
  SQL> SET LINESIZE 100
  SQL> SELECT DBMS_SQLTUNE.REPORT_TUNING_TASK( 'my_sql_tuning_task')
       FROM DUAL;

  The report consists of findings and recommendations. The Tuning Advisor provides the rationale and the expected benefits for each recommendation. It also provides you with the SQL to implement the recommendations.

You can use the following views to manage your automatic SQL tuning efforts:

- DBA_ADVISOR_TASKS
- DBA_ADVISOR_FINDINGS
- DBA_ADVISOR_RECOMMENDATIONS
- DBA_ADVISOR_RATIONALE
- DBA_SQLTUNE_STATISTICS
- DBA_SQLTUNE_PLANS
Managing SQL Profiles

Once the ATO has made its recommendations, you can accept its findings and run the DBMS_SQLTUNE.ACCEPT_SQL_PROFILE procedure to create an appropriate SQL profile, though you must ensure you have the CREATE_ANY_PROFILE privilege first.

The preceding may seem to say that a SQL profile is an inevitable consequence of an ATO process, but it will only recommend that you create a SQL profile if it has built one as a result of its scan. However, it will only do this if it collected auxiliary information while analyzing statistics and profiling SQL (as detailed previously). Oracle will apply the new profile to the SQL statement when you execute it.

Managing SQL Tuning Categories

You may find that you have a number of different SQL profiles for a single SQL statement. Oracle has to manage them in some way, so it assigns each one to a SQL tuning category. The same process occurs when a user logs in, meaning that Oracle will assign a user to a tuning category. The category is selected according to the SQLTUNE_CATEGORY initialization parameter.

If you do not change it, SQLTUNE_CATEGORY takes the value DEFAULT. This means that any SQL profiles belonging to the default category apply to everyone who logs in. You can alter the SQL tuning category for every user with the ALTER SYSTEM command. You can also alter a session’s tuning category with the ALTER SESSION command. For example, take the PROD and DEV categories. To change the SQL tuning category for every user, do the following:

```
SQL> ALTER SYSTEM SET SQLTUNE_CATEGORY = PROD;
```

If you wanted to change a session’s tuning category, you could do this:

```
SQL> ALTER SESSION SET SQLTUNE_CATEGORY = DEV;
```

Note You may also use the DBMS_SQLTUNE.ALTER_SQL_PROFILE procedure to alter the SQL tuning category.

Using the OEM to Run the SQL Tuning Advisor

To use the OEM to run the Advisor, click Related Links ➤ Advisor Central ➤ SQL Tuning Advisor. This is the SQL Tuning Advisor page. Here you can specify the SQL statements that the SQL Advisor will analyze, which can be one of two kinds:

- **Top SQL**: These SQL statements could be top SQL from the cursor cache or saved high-load SQL statements that have come from the AWR.
- **SQL Tuning Sets**: You can create an STS from any set of SQL statements.

Choosing one of the four links on the SQL Tuning Advisor page will take you to your selected data source. You can now launch the SQL Tuning Advisor if you wish.

The Automatic SQL Tuning Advisor

I explained the Automatic Tuning Optimizer earlier in this chapter. This is what Oracle calls the optimizer when it runs in tuning mode. The Automatic Tuning Optimizer performs the following types of analysis for high-load SQL statements, with a goal of isolating poorly written SQL statements and making recommendations to improve them: statistics analysis, SQL profiling, access path analysis, and SQL structure analysis. When you execute a SQL Tuning Advisor session, it invokes the Automatic Tuning Optimizer to tune the SQL statements. The SQL Tuning Advisor provides recommendations, but can’t implement them for you.
The Automatic Tuning Optimizer also runs regularly as an automated maintenance task called the (Automatic) SQL Tuning Advisor task. The advisor can identify poorly running SQL statements by picking them from the AWR, make recommendations to improve them, and also implement any recommendations that invoke the use of SQL profiles. The SQL Tuning Advisor task conducts statistics analysis, SQL profiling, access path analysis, and SQL structure analysis.

The Automatic SQL Tuning process consists of the identification of candidates for tuning, tuning the statements and making recommendations, testing the recommendations, and automatic implementation of the SQL profile recommendations. I describe each of the steps in the following sections.

### Identifying SQL Tuning Candidates

The Automatic SQL Tuning Advisor uses the AWR Top SQL identification process for selecting candidates for automatic tuning. The database takes into account the CPU time and I/O time utilized by SQL statements to select these candidates. The goal is to select statements that offer a large potential for improvement. The advisor prepares a list of candidate statements by organizing the top SQL queries in the past week into the following “buckets”:

- Top for the week
- Top for any day in the week
- Top for any hour during the week
- Highest average single execution

By assigning weights to each of the buckets, the SQL Tuning Advisor combines the four buckets into a single group of statements and ranks the statements according to their impact on performance. Subsequently, during the maintenance window, the advisor automatically tunes each of the candidate SQL statements selected by it.

A SQL profile consists of addition statistics beyond those collected by the optimizer, to help evolve better execution plans. The additional information gathered via SQL profiles may include customized optimizer settings, adjustments to compensate for missing or stale statistics, and adjustments for estimation errors in optimization statistics. Since you don’t need to change the SQL query when you implement a SQL profile, they are ideal for use in packaged applications. Implementation of a SQL profile would normally lead to the generation of more efficient execution plans for SQL statements.

### Tuning and Making Recommendations

The SQL Tuning Advisor tunes statements in the order of their performance impact. If the advisor finds stale or missing statistics, it lets the `GATHER_STATS_JOB` know about this fact, so it can collect statistics for it when the database collects statistics the next time.

The advisor makes different types of recommendations to improve the poorly performing SQL statements, including the creation of indexes, refreshing of the optimizer statistics, restructuring SQL statements, and creation of SQL profiles. The advisor can automatically implement only the SQL profile creation recommendations. The advisor creates and tests the SQL profiles it recommends before implementing them. You can decide whether to retain the new SQL profiles that are automatically implemented by the advisor or not, based on an analysis of the SQL Tuning Advisor report.

### Testing the Recommendations for New SQL Profiles

For any SQL profile recommendation it makes, the SQL Tuning Advisor runs the statement with and without the profile and compares the performance. The advisor will recommend adopting a profile only if implementing the profile leads to at least a threefold increase in improvement in performance, as shown by a reduction in the sum of the CPU and I/O usage.
Implementing the SQL Profiles

The setting of the ACCEPT_SQL_PROFILES attribute of the SET_TUNING_TASK_PARAMETERS view determines whether the database automatically accepts the SQL profile recommendations made by the Automatic SQL Tuning Advisor. The DBA_SQL_PROFILES view shows all the automatically implemented SQL profiles. If a SQL profile was automatically implemented, it'll have a value of AUTO in the TYPE column.

Limitations

You can’t tune the following types of statements with the Automatic SQL Tuning Advisor:

- Parallel queries
- Ad hoc queries
- Recursive statements
- SQL statements that use the INSERT and DELETE statements
- SQL statements that use DDL statements such as CREATE TABLE AS SELECT

If a query takes a long time to execute after implementing a SQL profile, the advisor will reject the implementation of the SQL profile, since it can’t test-execute the query. Note that with the exception of ad hoc statements, you can manually tune all the preceding types of statements with a manual invocation of the SQL Tuning Advisor.

Configuring Automatic SQL Tuning

Use the DBMS_SQLTUNE package to configure and manage the automatic SQL tuning task. Manage the SYS_AUTO_TUNING_TASK, which controls the automatic SQL tuning job, with the following procedures:

- SET_TUNING_TASK_PARAMETERS: Use this procedure to test task parameters controlling items such as whether to automatically implement SQL profiles.
- EXECUTE_TUNING_TASK: Use this procedure to run the tuning task in the foreground.
- EXPORT_TUNING_TASK: This procedure helps produce a task execution report.

The Automatic SQL Tuning Advisor job runs for a maximum of one hour by default, but you can change the execution time limit by executing the SET_TUNING_TASK_PARAMETERS procedure, as shown here:

```
SQL> exec dbms_sqltune.set_tuning_task_parameter
('SYS_AUTO_SQL_TUNING_TASK', 'TIME_LIMIT', 14400);
```

The previous example shows how to raise the maximum run time for the SQL tuning task to four hours, from its default value of one hour.

The SET_TUNING_TASK_PARAMETERS procedure enables you to configure the tuning task by specifying the following parameters:

- ACCEPT_SQL_PROFILES determines whether the database must automatically accept a SQL profile.
- REPLACE_USER_SQL_PROFILES determines whether the task should replace the SQL profiles created by the user.
- MAX_SQL_PROFILES_PER_EXEC specifies the maximum number of SQL profiles that can be accepted for a single automatic SQL tuning task.
- **MAX_AUTO_SQL_PROFILES** determines the total number of SQL profiles that are accepted by the database.
- **EXECUTION_DAYS_TO_EXPIRE** specifies the maximum number of days for which the database saves the task history. The default is 30 days.

The following example shows how to configure a SQL tuning task that'll automatically accept all SQL profile recommendations:

```sql
SQL> begin
  2  dbms_sqltune.set_tuning_task_parameters(
  3    task_name => 'SYS_AUTO_SQL_TUNING_PROG',
  4    parameter => 'accept_sql_profiles', value => 'true');
  5* end;
SQL> /
```

The previous example sets the value for the **ACCEPT_SQL_PROFILES** parameter to **TRUE**, which makes the advisor automatically accept SQL profile recommendations.

The **SYS_AUTO_SQL_TUNING_TASK** procedure runs the automatic SQL tuning job every night during the maintenance window of the Oracle Scheduler. It tunes SQL statements according to the priority ranking in the SQL candidates. It creates necessary SQL profiles for a statement and tests them before tuning the next statement in the candidate list.

### Managing Automatic SQL Tuning

Use the **DBMS_AUTO_TASK_ADMIN** package to enable and disable the Automatic SQL Tuning Advisor job during the Scheduler maintenance window. The **ENABLE** procedure helps you enable the Automatic SQL Tuning Advisor task:

```sql
begin
  dbms_auto_task_admin.enable (
    client_name => 'sql tuning advisor',
    operation => 'NULL',
    window_name='NULL');
end;
```

The value **NULL** for the **WINDOW_NAME** parameter will enable the task in all maintenance windows. To specify the task in a specific maintenance window, specify a window name, as shown here:

```sql
begin
  dbms_auto_task_admin.enable (
    client_name => 'sql tuning advisor',
    operation => 'NULL',
    window_name='monday_night_window');
end;
```

To disable the Automatic SQL Tuning Advisor task, execute the **DISABLE** procedure, as shown here:

```sql
begin
  dbms_auto_task_admin.disable (
    client_name => 'sql tuning advisor',
    operation => 'NULL',
    window_name='NULL');
end;
```
The previous code will disable the automatic SQL tuning tasks in all maintenance windows, since I didn’t specify a value for the WINDOW_NAME parameter.

**Tip** By setting the TEST_EXECUTE parameter when you execute the SET_TUNING_TASK_PARAMETER procedure, you can run the SQL Tuning Advisor in test execute mode to save time.

You can also configure all Automatic SQL Tuning parameters easily through the Database control (or the Grid Control). Go to the Automatic SQL Tuning Settings page, accessible by clicking the Configure button in the Automated Maintenance Tasks page. You can configure all automated tasks from the Automated Maintenance Tasks configuration page. Here’s a simple example that shows how to get recommendations for fixing a SQL statement:

1. Click the finding with the highest impact on database time in the Database Home page.
2. Click Schedule SQL Tuning Advisor on the SQL Details page.
3. Click Submit on the Scheduler Advisor page.
4. Click Implement if you want to adopt the advisor’s recommendations.
5. Click Yes on the Confirmation page, and the database creates a new SQL profile.
6. View the tuning benefits by going to the Performance page after the database executes the tuned statement again.

Go to the Automated Maintenance Task page to view information about the latest executions of the Automatic SQL Tuning Advisor. Click the Server tab in the Database Control home page first. Click the Automated Maintenance Tasks link under the Tasks section in the Server page, and then click the most recent execution icon or the Automatic SQL Tuning task link to view the Automatic SQL Tuning Result Summary page.

**Interpreting Automatic SQL Tuning Reports**

You can get a report of the Automatic SQL Tuning Advisor tasks by executing the REPORT_AUTO_TUNING_TASK function, as shown here:

```sql
SQL> begin
  2  :test_report := dbms_sqltune.report_auto_tuning_task (
  3     type         => 'text',
  4     level        => 'typical',
  5     section      => 'all');
6* end;
SQL> /
PL/SQL procedure successfully completed.
SQL>
print :test_report
```

The report produced by the previous code contains information about all the statements analyzed by the advisor in its most recent execution and includes both the implemented and unimplemented advisor recommendations. The report also contains EXPLAIN PLANS before and after implementing the tuning recommendations.
You can use the following views to get information about the Automatic SQL Tuning Advisor jobs:

- `DBA_ADVISOR_EXECUTIONS`: Shows metadata information for each task
- `DBA_ADVISOR_SQLSTATS`: Shows a list of all SQL compilation and execution statistics
- `DBA_ADVISOR_SQLPLANS`: Shows a list of all SQL execution plans

The following code, for example, provides information about all Automatic SQL Tuning Advisor tasks:

```sql
SQL> SELECT execution_name, status, execution_start, execution_end
    2       FROM dba_advisor_executions
    3     WHERE task_name='SYS_AUTO_SQL_TUNING_TASK';
```

### Using Other GUI Tools

The EXPLAIN PLAN and SQL Trace utilities aren’t the only tools you have available to tune SQL statements. Several GUI-based tools provide the same information much more quickly. Just make sure that statistics collection is turned on in the initialization file before you use these tools. One of the well-known third-party tools is the free version of TOAD software, which is marketed by Quest Software (http://www.quest.com). From this tool you get not only the execution plan, but also memory usage, parse calls, I/O usage, and a lot of other useful information, which will help you tune your queries. The use of GUI tools helps you avoid most of the drudgery involved in producing and reading EXPLAIN PLANS. Note that whether you use GUI tools or manual methods, the dynamic performance views that you use are the same. How you access them and use the data makes the difference in the kind of tool you use.

### Using the Result Cache

You can improve the response times of frequently executed SQL queries by using the result cache. The result cache stores results of SQL queries and PL/SQL functions in a new component of the SGA called the Result Cache Memory. The first time a repeatable query executes, the database caches its results. On subsequent executions, the database simply fetches the results from the result cache instead of executing the query again. The database manages the result cache. You can turn result caching on only at the database level. If any of the objects that are part of a query are modified, the database invalidates the cached query results. Ideal candidates for result caching are queries that access many rows to return a few rows, as in many data warehousing solutions.

The result cache consists of two components, the SQL Query Result Cache that stores SQL query results and the PL/SQL Function Result Cache that stores the values returned by PL/SQL functions, with both components sharing the same infrastructure. I discuss the two components of the result cache in the following sections.

### Managing the Result Cache

The result cache is always enabled by default, and its size depends on the memory the database allocates to the shared pool. If you specify the `MEMORY_TARGET` parameter for allocating memory, Oracle allocates 0.25% of the `MEMORY_TARGET` parameter value to the result cache. If you specify the `SGA_TARGET` parameter instead, Oracle allocates 0.5% of the `SGA_TARGET` value to the result cache.

You can change the memory allocated to the result cache by setting the `RESULT_CACHE_MAX_SIZE` initialization parameter. This parameter can range from a value of zero to a system-dependent maximum. You disable result caching by setting the parameter to zero, as shown here:
SQL> ALTER SYSTEM SET result_cache_max_size=0;

Since result caching is enabled by default, it means that the RESULT_CACHE_MAX_SIZE parameter has a positive default value as well, based on the size of the MEMORY_TARGET parameter (or the SGA_TARGET parameter if you have that parameter instead).

In addition to the RESULT_CACHE_MAX_SIZE parameter, two other initialization parameters have a bearing on the functioning of the result cache: the RESULT_CACHE_MAX_RESULT parameter specifies the maximum amount of the result cache a single result can use. By default, a single cached result can occupy up to 5 percent of the result cache, and you can specify a percentage between 1 and 100. The RESULT_CACHE_REMOTE_EXPIRATION parameter determines the length of time for which a cached result that depends on remote objects is valid. By default, this parameter is set to zero, meaning you aren’t supposed to use the result cache for queries involving remote objects. The reason for this is over time remote objects could be modified, leading to invalid results in the cache.

### Setting the RESULT_CACHE_MODE Parameter

Whether the database caches a query result or not depends on the value of the RESULT_CACHE_MODE initialization parameter, which can take two values: MANUAL or FORCE. Here’s how the two values affect result caching behavior in the database:

- If you set the parameter to FORCE, the database will try to use the cache for all results, wherever it’s possible to do so. You can, however, skip the cache by specifying NO_RESULT_CACHE hint within a query.
- If you set the parameter to MANUAL, the database caches the results of a query only if you include the RESULT_CACHE hint in the query.

By default, the RESULT_CACHE_MODE parameter is set to MANUAL and you can change the value dynamically as shown here:

SQL> alter session set result_cache_mode=force scope=spfile;

### Using the RESULT_CACHE and NO_RESULT_CACHE Hints

Using the RESULT_CACHE hint as a part of a query adds the ResultCache operator to a query’s execution plan. The ResultCache operator will search the result cache to see whether there’s a stored result in there for the query. It retrieves the result if it’s already in the cache; otherwise, the ResultCache operator will execute the query and store its results in the result cache. The no_result_cache operator works the opposite way. If you add this hint to a query, it’ll lead the ResultCache operator to bypass the result cache and reexecute the query to get the results.

The following example shows how to incorporate the RESULT_CACHE hint in a SQL query:

```sql
SQL> select /*+ result_cache */
2  department_id, avg(salary)
3  from hr.employees
4* group by department_id;

SQL>
```

The RESULT_CACHE hint in line 1 of the query adds the ResultCache operator, which looks in the result cache for the cached results and, if they aren’t there already, executes the query and stores its results in the result cache. The EXPLAIN PLAN for the query shows that the query will utilize the result cache:
Chapter 19  ■  Improving Database Performance: SQL Query Optimization

SQL> EXPLAIN PLAN FOR select /*+ result_cache +*/
    2  department_id, avg(salary)
    3  from hr.employees
    4  group by department_id
SQL> /
Explained.

SQL>
SQL> SELECT plan_table_output FROM table(DBMS_XPLAN.DISPLAY());
PLAN_TABLE_OUTPUT
----------------------------------------------------------------
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>11</td>
<td>77</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(25)</td>
</tr>
<tr>
<td>1</td>
<td>RESULT CACHE</td>
<td>8nk7a7rfhymzyosob89ksn9bfz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HASH GROUP BY</td>
<td></td>
<td>11</td>
<td>77</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(25)</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>EMPLOYEES</td>
<td>107</td>
<td>749</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0)</td>
</tr>
</tbody>
</table>
----------------------------------------------------------------

PLAN_TABLE_OUTPUT

Result Cache Information (identified by operation id):
-----------------------------------------------
1 - column-count=2; dependencies=(HR.EMPLOYEES);
name="select /*+ result_cache +*/
department_id, avg(salary)
from hr.employees
group by department_id"
15 rows selected.

SQL>

Tip  The RESULT_CACHE and the NO_RESULT_CACHE hints always take precedence over the value you set for the RESULT_CACHE_MODE initialization parameter.

The EXPLAIN PLAN output reveals the use of the result cache by the query in our example. Since I used the RESULT_CACHE hint to use the result cache, the RESULT_CACHE_MODE parameter is set to MANUAL. If it is set to FORCE, I don’t have to set the RESULT_CACHE hint inside the queries. The database will simply cache results for all repeatable SQL statements, unless I specify the NO_RESULT_CACHE hint in a query.
Managing the Result Cache

Use the DBMS_RESULT_CACHE package to manage the result cache, such as checking the status of the cache and flushing the cache. The following example shows how to check the memory allocation to the cache by executing the MEMORY_REPORT function:

```sql
SQL> set serveroutput on
SQL> exec dbms_result_cache.memory_report
Result Cache Memory Report
[Parameters]
Block Size = 1K bytes
Maximum Cache Size = 672K bytes (672 blocks)
Maximum Result Size = 33K bytes (33 blocks)
[Memory]
Total Memory = 5132 bytes [0.005% of the Shared Pool]
... Fixed Memory = 5132 bytes [0.005% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]

PL/SQL procedure successfully completed.
SQL>
```

Execute the STATUS function to check the current status of the result cache, which could be ENABLED or DISABLED. You can purge the contents of the result cache by executing the FLUSH procedure or the FLUSH function. You may have to purge the result cache if the database ends up filling up the result cache, as the result cache doesn’t automatically flush its contents. If you load a new version of a function, for example, you can get rid of the older function’s results from the result cache by purging the results with the FLUSH procedure or function. Before you execute the FLUSH procedure or FLUSH function, you must first put the result cache in bypass mode by executing the BYPASS procedure with the TRUE value. Once you purge the result cache, execute the BYPASS procedure again, now with the FALSE value, as shown here:

```sql
BEGIN
EXEC dbms_result_cache.bypass (FALSE);
END;
/
PL/SQL procedure successfully completed.
SQL>
```

You can use the following views to manage the result cache:

- **V$RESULT_CACHE_STATISTICS**: Lists cache settings and memory usage statistics
- **V$RESULT_CACHE_OBJECTS**: Lists all cached objects and their attributes
- **V$RESULT_CACHE_DEPENDENCY**: Lists the dependency information between the cached results and dependencies
- **V$RESULT_CACHE_MEMORY**: Lists all memory blocks and their statistics
- **V$RESULT_CACHE_OBJECTS**: Lists both cached results and all dependencies

For example, you can use the following query on the V$RESULT_CACHE_OBJECTS view to find out which results are part of the result cache:
The previous query shows there are currently two results in the result cache.

Restrictions on Using the SQL Query Result Cache

You can’t cache results in the SQL Query Result Cache for the following objects:

- Temporary tables
- Dictionary tables
- Nondeterministic PL/SQL functions
- The curval and nextval pseudo functions
- The SYSDATE, SYS_TIMESTAMP, CURRENT_DATE, CURRENT_TIMESTAMP, LOCAL_TIMESTAMP, USERENV, SYS_CONTEXT, and SYS_QUID functions

You also won’t be able to cache subqueries, but you can use the RESULT_CACHE hint in an inline view.

The PL/SQL Function Result Cache

The SQL Query Result Cache shares the result cache infrastructure with the PL/SQL Function Result Cache, which caches the results of PL/SQL functions. Candidates for PL/SQL function caching are those functions that the database uses frequently that depend on fairly static information. You can choose to specify that the database invalidate the cached results of a PL/SQL function when there’s a change in any of the objects the functions depends on.

Creating a Cacheable Function

Include the RESULT_CACHE clause in a PL/SQL function definition to make the database cache the function results in the PL/SQL Function Result Cache. Here’s an example:

```sql
SQL> CREATE OR REPLACE function get_dept_info (dept_id number) RETURN dept_info_record result_cache relies_on (employees)
   IS
   rec dept_info_record;
   BEGIN
       SELECT AVG(salary), COUNT(*) INTO rec
       FROM employees
       WHERE department_id = dept_id;
   RETURN rec;
END get_dept_info;
/
```

The RELIES ON clause is optional. The clause specifies that the database must invalidate the function results if any of the tables or other objects that the function depends on undergoes a modification.
The first time the database executes the `GET_DEPT_INFO` function, the function will execute as usual. On subsequent executions of the function, the database fetches the function values directly from the PL/SQL Function Result Cache instead of reexecuting the function. The database reexecutes the function only when

- You bypass the result cache by not specifying the `RESULT_CACHE` hint.
- You execute the `DBMS_RESULT_CACHE_BYPASS` procedure to make functions and queries bypass the result cache, regardless of the setting of the `RESULT_CACHE_MODE` parameter or the specification or the `RESULT_CACHE` hint.
- Any of the objects underlying a function change and you’ve specified the `RELIERS_ON` clause in the function definition.
- The database ages out cached results because the system needs additional memory.

**Restrictions**

A PL/SQL function must satisfy the following requirements in order for the database to cache its results. A function cannot

- Have any IN/OUT parameters
- Be an anonymous block
- Be defined in a module that has invoker’s rights
- Have parameters that belong to the collection, object, ref cursor, or LOB types
- Be a pipelined table function

Besides meeting these requirements, the function must not depend on session-specific settings or application contexts and must also not have any side effects.

**The Client Query Result Cache**

If you are using any OCI applications and drivers such as JDBC and ODP.NET, you can also use Oracle’s client-side caching of SQL result sets in the Client Query Result Cache that’s located on the server. The database keeps the result sets consistent with changes in session attributes. If you’ve frequently repeated statements in your applications, client-side caching could offer tremendous improvement in query performance benefits. Since the database caches results on the clients, server round-trips are minimized and scalability improves as a result, with lower I/O and CPU load.

Unlike server-side caching, client-side caching isn’t enabled by default. If your applications produce small result sets that are static over a period of time, client-side caching may be a good thing to implement. Frequently executed queries and queries involving lookup tables are also good candidates for client-side caching.

**Enabling and Disabling the Client Query Result Cache**

As with server-side caching, you use the `RESULT_CACHE_MODE` initialization parameter to enable and disable client-side caching. The `RESULT_CACHE` and the `NO_RESULT_CACHE` hints work the same way as they do for server-side caching. If you choose to specify the `MANUAL` setting for the `RESULT_CACHE_MODE` parameter, you must use the `RESULT_CACHE` hint in a query for the query’s results to be cached. Also, the two hints override the setting of the `RESULT_CACHE_MODE` parameter, as in the case of server-side caching. You pass the `RESULT_CACHE` and the `NO_RESULT_CACHE` hints to SQL statements by using the `OCIStatementPrepare()` and the `OCIStatementPrepare2()` calls.
Managing the Client Result Cache

There are two initialization parameters that control how the Client Query Result Cache works. Here’s a brief description of these parameters:

- **CLIENT_RESULT_CACHE_SIZE**: Determines the maximum client per-process result set cache size (in bytes). If you set this parameter to zero, you disable the Client Query Result Cache. The database allocates the maximum-size memory to every OCI client process by default.

  **Tip** You can override the setting of the `CLIENT_RESULT_CACHE_SIZE` parameter with the server-side parameter `OCI_RESULTCACHE_MAX_SIZE`. By setting the latter to zero, you can disable the Client Query Result Cache.

- **CLIENT_RESULT_CACHE_LAG**: Determines the Client Query Result Cache lag time. A low value means more round-trips to the database from the OCI client library. Set this parameter to a low value if your application accesses the database infrequently.

  An optional client configuration file overrides any client caching configuration parameters you set. You can set the client-side configuration parameters in the `sqlnet.ora` file on the client. You can set the following client-side configuration parameters:

  - **OCI_RESULTCACHE_MAX_SIZE**: Specifies the maximum size of the query cache for a single process
  - **OCI_RESULTCACHE_MAX_RSET_SIZE**: Enables you to specify the maximum size of a single result in bytes for a process
  - **OCI_RESULTCACHE_MAX_RST_ROWS**: Sets the maximum size of a query result in rows for a single process

  You can also insert the `RESULT_CACHE` and `NO_RESULT_CACHE` hints in OCI applications. Use the `CLIENT_RESULTCACHE` view to see the settings of the result cache and the usage statistics for the Client Query Result Cache.

Restrictions

You can’t cache queries that use the following types of objects, even though you may be able to cache them in a server-side result cache:

- Views
- Remote objects
- Complex types in the select list
- Flashback queries
- Queries that include PL/SQL functions
- Queries that reference VPD policies on the tables

A Simple Approach to Tuning SQL Statements

Whether you use manual methods such as EXPLAIN PLAN, SQL Trace, and TKPROF, or more sophisticated methods such as the SQL Tuning Advisor, you need to understand that optimizing SQL statements can improve performance significantly. In the following sections, I summarize a simple methodology you can follow to tune your SQL statements.
Identify Problem Statements

This chapter has shown you many ways you can identify your slow-running or most resource-intensive SQL statements. For instance, you can use dynamic performance views such as V$SQL to find out your worst SQL statements, as shown earlier. Statements with high buffer gets are the CPU-intensive statements and those with high disk reads are the high I/O statements. Alternatively, you can rely on the AWR report and the ADDM analysis to figure out which of your SQL statements need to be written more efficiently. Obviously, you want to start (and maybe end) with tuning these problem statements.

Locate the Source of the Inefficiency

The next step is to locate the inefficiency in the SQL statements. To do this, you need to collect information on how the optimizer is executing the statement. That is, you must first walk through the EXPLAIN PLAN for the statement. This step helps you find out if there are any obvious problems, such as full table scans due to missing indexes.

In addition to analyzing the EXPLAIN PLAN output or using the V$SQL_PLAN view, collect the performance information, if you can, by using the SQL Trace and TKPROF utilities.

Review each EXPLAIN PLAN carefully to see that the access and join methods and the join order are optimal. Specifically, check the plans with the following questions in mind:

- Are there any inefficient full table scans?
- Are there any unselective range scans?
- Are the indexes appropriate for your queries?
- Are the indexes selective enough?
- If there are indexes, are all of them being used?
- Are there any later filter operations?
- Does the driving table in the join have the best filter?
- Are you using the right join method and the right join order?
- Do your SQL statements follow basic guidelines for writing good SQL statements (see the section “Writing Efficient SQL” in this chapter)?

In most cases, a structured analysis of the query will reveal the source of the inefficiency.

Tune the Statement

Use the Database Control’s SQL Access Advisor to get index and materialized view recommendations. Review the access path for the tables in the statement and the join order. Consider the use of hints to force the optimizer to use a better execution plan. You can also use the SQL Tuning Advisor to get recommendations for more efficient SQL statements.

Compare Performance

Once you generate alternative SQL, it’s time to go through the first three steps again. Use the EXPLAIN PLAN facility and performance statistics to compare the new statement with the older one. After you ensure that your new statements perform better, it’s time to replace the inefficient SQL. Oracle Database 11g has a much wider array of automatic SQL tuning capabilities than ever before. Once you get familiar with the various automatic tuning tools, such as the SQL Tuning Advisor and the ADDM, you should be able to harness the database’s capabilities to tune your recalcitrant SQL statements.
CHAPTER 20

Performance Tuning: Tuning the Instance

In the previous chapter, you learned how to write efficient SQL to maximize an application’s performance. The use of optimal SQL and efficient design of the layout of the database objects are parts of a planned or proactive tuning effort. This chapter focuses on the efficient use of the resources Oracle works with: memory, CPU, and storage disks.

The chapter discusses how to monitor and optimize memory allocation for the Oracle instance. In this context, you’ll learn about the traditional database hit ratios, such as the buffer cache hit ratios. However, focusing on the hit ratios isn’t necessarily the smartest way to maintain efficient Oracle databases because you need to focus on the user’s response time. Investigating factors that are causing processes to spend excessive time waiting for resources is a better approach to performance tuning. This chapter provides you with a solid introduction to Oracle wait events and tells you how to interpret them and reduce the incidence of these wait events in your system.

A fairly common problem in many production systems is that of a database hang, when things seem to come to a standstill for some reason. This chapter shows you what to do during such events.

The chapter explains the key dynamic performance tables that you need to be familiar with to understand instance performance issues. Although you’ve encountered the Automatic Database Diagnostic Monitor (ADDM) and Automatic Workload Repository (AWR) in earlier chapters, this chapter reviews their role in instance tuning. You can also use the Active Session History (ASH) feature to understand recent session history. Analyzing ASH information helps solve numerous performance issues in a running instance.

Although it’s nice to be able to design a system proactively for high performance, more often than not, the DBA has to deal with reactive tuning when performance is unsatisfactory and a fix needs to be found right away. The final part of this chapter deals with a simple methodology to follow when your system performance deteriorates and you need to fine-tune the Oracle instance.

I begin this chapter with a short introduction to instance tuning and then turn to cover in detail the tuning of crucial resources such as memory, disk, and CPU usage. Later on in the chapter, I review the important Oracle wait events, which will help you get a handle on several kinds of database performance issues.

An Introduction to Instance Tuning

Oracle doesn’t give anything but minimal and casual advice regarding the appropriate settings of key resources, such as total memory allocation or the sizes of the components of memory. Oracle has some general guidelines about the correct settings for several key initialization parameters that have a bearing on performance. However, beyond specifying wide ranges for the parameters, the company’s guidelines aren’t helpful to DBAs deciding on the optimal levels for these parameters.
Oracle says this is because all these parameters are heavily application dependent. All of this means that you as a DBA have to find out the optimal sizes of resource allocations and the ideal settings of key initialization parameters through trial and error. As a DBA, you’re often called in to tune the instance when users perceive slow response caused by a bottleneck somewhere in the system. This bottleneck can be the result of either an excessive use of or insufficient provision of some resource. In addition, database locks and latches may cause a slowdown. You have to remember, though, that in most cases, the solution isn’t simply to increase the resource that seems to be getting hit hard—that may be the symptom, not the cause of a problem. If you address the performance slowdown by fixing the symptoms, the root causes will remain potential troublemakers.

Performance tuning an Oracle database instance involves tuning memory and I/O as well as operating system resources such as CPU, the operating system kernel, and the operating system memory allocation. When you receive calls from the help desk or other users of the system complaining that the system is running slowly, you can only change what’s under your direct control—mainly, the allocation of memory and its components and some dynamic initialization parameters that have a bearing on instance performance. Depending on what the various indicators tell you, you may adjust the shared pool and other components of memory to improve performance. You can also change the operating system priority of some processes, or quickly add some disk drives to your system.

One of the main reasons for a slow response time in a production system is due to user processes waiting for a resource. Oracle provides several ways of monitoring waits, but you need to understand their significance in your system. Long wait times aren’t the problem themselves; they’re symptoms of deep-seated problems. The DBA should be able to connect different types of waits with possible causes in the application or in the instance.

Although some manuals tell you that you should do performance tuning before application tuning—before you proceed to tuning areas such as memory, I/O, and contention—real life isn’t so orderly. Most of the time, you don’t have the opportunity to have the code revised, even if there are indications that it isn’t optimal. Instead of being an orderly process, tuning databases is an iterative process, where you may have to go back and forth between stages.

More often than not, DBAs are forced to do what they can to fix the performance problem that’s besetting them at that moment. In this sense, most performance tuning is a reactive kind of tuning. Nevertheless, DBAs should endeavor to understand the innards of wait issues and seek to be proactive in their outlooks.

There are two big advantages to being in a proactive mode of tuning. First, you have fewer sudden performance problems that force hurried reactions. Second, as your understanding of your system increases, so does your familiarity with the various indicators of poor performance and the likely causes for them, so you can resolve problems that do occur much more quickly.

If you’re fortunate enough to be associated with an application during its design stages, you can improve performance by performing several steps, including choosing automatic space management and setting correct storage options for your tables and indexes. Sizing the table and indexes correctly doesn’t hurt, either. However, if you’re stuck with a database that has a poor design, all is not lost. You can still tune the instance using techniques that I show later in this chapter to improve performance.

When response time is slower than usual, or when throughput falls, you’ll notice that the Oracle instance isn’t performing at its usual level. If response times are higher, obviously there’s a problem somewhere in one of the critical resources Oracle uses. If you can rule out any network slowdowns, that leaves you with memory (Oracle’s memory and the system’s memory), the I/O system, and CPUs. One of these resources is usually the bottleneck that’s slowing down your system.

In the next few sections, you’ll learn how to tune key system resources such as memory, I/O, and CPU to improve performance. You’ll also see how to measure performance, detect inefficient waits in the system, and resolve various types of contention in an Oracle database. The next section presents a discussion of how tuning Oracle’s memory can help improve database performance.
Oracle Corp., like the other software vendors, releases periodic *patches* or *patch sets*, which are a set of fixes for bugs discovered by either Oracle or its customers. When you get in touch with Oracle technical support, one of the things the technical support representative will commonly ask you to do is make sure you have applied the latest patch set to your Oracle software. Similarly, UNIX operating systems may have their own patch sets that you may have to apply to fix certain bugs.

Each of Oracle’s patch sets could cover fixes for literally hundreds of bugs. My recommendation is to apply a patch set as soon as it’s available. One of the primary reasons for this is to see whether your bug is unique to your database or if a general solution has already been found for the problem. When you ask Oracle technical support to resolve a major problem caused by a bug, Oracle usually provides you with a workaround. Oracle recommends that you upgrade your database to the latest versions and patch sets because some Oracle bugs may not have any workarounds or fixes. Oracle will continue to support older versions of its server software throughout their support life cycle, which is usually about two to three years after the next major release. Many organizations see no urgency to move to newer versions, as Oracle continues to support the older versions after the release of the new versions.

The question regarding how quickly you should convert to a new version is somewhat tricky to answer. Traditionally, people have shied away from being early adopters of new Oracle software versions. Oracle, like most other software companies, has a reputation for buggy initial releases of its major software versions. DBAs and managers in general prefer to wait a while until a “stable version” comes out. Although the logic behind this approach is understandable, you must also figure in the cost of not being able to use the many powerful features Oracle introduces in each of its major releases.

Because nobody likes to jeopardize the performance of a production system, the ideal solution is to maintain a test server where the new software is tested thoroughly before being moved into production as early as possible. However, don’t wait forever to move to a new version—by the time some companies move to the new version, an even newer Oracle version is already out!

Some of your good SQL statements may not be so good after you migrate to a new version, due to the way a hint might behave in the new version, for example. That’s why it’s extremely important to test the whole system on the new version before cutting over production systems. A smart strategy is to collect a set of performance statistics that can serve as a baseline before you make any major changes in the system. These system changes may include the following:

- Migrating or upgrading a database
- Applying a new database or operating system patch set
- Adding a new application to your database
- Substantially increasing the user population

### Automatic Performance Tuning vs. Dynamic Performance Views

Traditionally, Oracle DBAs relied heavily on the use of dynamic performance views (V$ views) to gather performance statistics and diagnose instance performance problems. You have access to all the traditional views in Oracle Database 11g. However, you now also have powerful automatic performance tuning features that provide a faster and more painless way to approach instance performance tuning. Most of these tools use the same V$ dynamic performance views that you use in manual performance tuning. Although I provide several examples of manual performance tuning in this chapter, I must emphasize the importance of understanding and using the powerful set of
automatic performance features that are already a part of your database. Here’s a brief summary of the automatic performance tuning features:

- The AWR collects all the performance data necessary for tuning as well as diagnosing instance problems.
- The ADDM automatically diagnoses database performance by analyzing the AWR data.
- The Automatic SQL Tuning Advisor provides SQL tuning recommendations.
- The database automatically runs the statistics collection job, thus keeping all statistics up to date.
- The Segment Advisor runs automatically during the maintenance interval and makes recommendations about which segments to shrink and which to reorganize (for example, due to excessive row chaining).
- The SQL Access Advisor provides recommendations about the ideal indexes and materialized views to create.
- The Memory Advisor, MTTR Advisor, and Undo Advisor help you tune memory, redo logs, and undo segments, respectively.

In this chapter, I present the major dynamic performance views that you can use to diagnose instance performance. Traditionally, Oracle DBAs relied heavily on scripts using these views to monitor and tune instance performance. However, the best way to diagnose and tune Oracle performance issues is through the OEM Database Control (or Grid Control). I thus show you a simple approach to tuning using the OEM Database Control.

---

**Note**  The AWR and ADDM are Oracle products that need special licensing through the purchase of the Diagnostic Pack. If you haven’t purchased this licensing, you aren’t supposed to use these features.

---

**Tuning Oracle Memory**

A well-known fact of system performance is that fetching data that’s stored in memory is a lot faster than retrieving data from disk storage. Given this, Oracle tries to keep as much of the recently accessed data as possible in its SGA. In addition to data, shared parsed SQL code and necessary data dictionary information are cached in memory for quick access. You can easily adjust the memory allocation of Oracle, by simply changing a single initialization parameter—`MEMORY_TARGET`.

There’s a two-way relationship between memory configuration and the application’s use of that memory. The correct memory allocation size depends on the nature of your application, the number of users, and the size of transactions. If there isn’t enough memory, the application will have to perform time-consuming disk I/Os. However, the application itself might be using memory unnecessarily, and throwing more memory at it may not be the right strategy. As a DBA, you must not view memory and its sizing in isolation. This can lead to some poor choices, as you address the symptoms instead of the causes for what seems like insufficient memory. The tendency on a DBA’s part is to allocate as much memory as possible to the shared pool, hoping that doing so will resolve the problem. However, sometimes this only exacerbates the problem. It’s wise to manage the database with as little memory as necessary, and no more. The system can always use the free memory to ensure there’s no swapping or paging. Performance slowdowns caused by paging outweigh the benefits of a larger SGA under most operating systems.
Tuning the Shared Pool

In a production database, the shared pool is going to command most of your attention because of its direct bearing on application performance. The shared pool is a part of the SGA that holds almost all the necessary elements for execution of the SQL statements and PL/SQL programs. In addition to caching program code, the shared pool caches the data dictionary information that Oracle needs to refer to often during the course of program execution.

Proper shared pool configuration leads to dramatic improvements in performance. An improperly tuned shared pool leads to problems such as the following:

- Increased latch contention with the resulting demand for more CPU resources
- Greater I/O because executable forms of SQL aren’t present in the shared pool
- Higher CPU usage because of unnecessary parsing of SQL code

The general increase in shared pool waits and other waits observed during a severe slowdown of the production database is the result of SQL code that fails to use bind variables (I explain the important concept of bind variables in the following section).

As the number of users increases, so does the demand on shared pool memory and latches, which are internal locks for memory areas. If there are excessive latches, the result might be a higher wait time and a slower response time. Sometimes the entire database seems to hang.

The shared pool consists of two major areas: the library cache and the data dictionary cache. You can’t allocate or decrease memory specifically for one of these components. If you increase the total shared pool memory size, both components will increase in some ratio that Oracle determines. Similarly, when you decrease the total shared pool memory, both components will decrease in size. Let’s look at these two important components of the shared pool in detail.

The Library Cache

The library cache holds the parsed and executable versions of SQL and PL/SQL code. As you may recall from Chapter 19, all SQL statements undergo the following steps during their processing:

- **Parsing**, which includes syntactic and semantic verification of SQL statements and checking of object privileges to perform the actions.
- **Optimization**, where the Oracle optimizer evaluates how to process the statement with the least cost, after it evaluates several alternatives.
- **Execution**, where Oracle uses the optimized physical execution plan to perform the action stated in the SQL statement.
- **Fetching**, which only applies to SELECT statements where Oracle has to return rows to you. This step isn’t necessary in any nonquery-type statements.

Parsing is a resource-intensive operation, and if your application needs to execute the same SQL statement repeatedly, having a parsed version in memory will reduce contention for latches, CPU, I/O, and memory usage. The first time Oracle parses a statement, it creates a parse tree. The optimization step is necessary only for the first execution of a SQL statement. Once the statement is optimized, the best access path is encapsulated in the access plan. Both the parse tree and the access plan are stored in the library cache before the statement is executed for the first time. Future invocation of the same statement will need to go through only the last stage, execution, which avoids the overhead of parsing and optimizing as long as Oracle can find the parse tree and access plan in the library cache. Of course, if the statement is a SQL query, the last step will be the fetch operation.
The library cache, being limited in size, discards old SQL statements when there’s no more room for new SQL statements. The only way you can use a parsed statement repeatedly for multiple executions is if a SQL statement is identical to the parsed statement. Two SQL statements are identical if they use exactly the same code, including case and spaces. The reason for this is that when Oracle compares a new statement to existing statements in the library cache, it uses simple string comparisons. In addition, any bind variables used must be similar in data type and size. Here are a couple of examples that show you how picky Oracle is when it comes to considering whether two SQL statements are identical.

In the following example, the statements aren’t considered identical because of an extra space in the second statement:

```sql
SELECT * FROM employees;
SELECT * FROM employees;
```

In the next example, the statements aren’t considered identical because of the different case used for the table Employees in the second statement. The two versions of employees are termed literals because they’re literally different from each other.

```sql
SELECT * FROM employees;
SELECT * FROM Employees;
```

Let’s say users in the database issue the following three SQL statements:

```sql
SELECT * FROM persons WHERE person_id = 10
SELECT * FROM persons WHERE person_id = 999
SELECT * FROM persons WHERE person_id = 6666
```

Oracle uses a different execution plan for the preceding three statements, even though they seem to be identical in every respect, except for the value of `person_id`. Each of these statements has to be parsed and executed separately, as if they were entirely different. Because all three are essentially the same, this is inherently inefficient. As you can imagine, if hundreds of thousands of such statements are issued during the day, you’re wasting database resources and the query performance will be slow. Bind variables allow you to reuse SQL statements by making them “identical,” and thus eligible to share the same execution plan.

In our example, you can use a bind variable, which I’ll call `:var`, to help Oracle view the three statements as identical, thus requiring a single execution instead of multiple ones. The `person_id` values 10, 99, and 6666 are “bound” to the bind variable, `:var`. Your replacement SQL statement using a bind variable, then, would be this:

```sql
SELECT * FROM persons WHERE person_id = :var
```

Using bind variables can dramatically increase query performance, and I explain in the section “Using the CURSOR_SHARING (Literal Replacement) Parameter” how you can “force” Oracle to use bind variables, even if an application doesn’t use them.

The Dictionary Cache

The dictionary cache, as mentioned earlier, caches data dictionary information. This cache is much smaller than the library cache, and to increase or decrease it you modify the shared pool accordingly. If your library cache is satisfactorily configured, chances are that the dictionary cache is going to be fine too. You can get an idea about the efficiency of the dictionary cache by using the following query:

```sql
SQL> SELECT (sum(gets - getmisses - fixed)) / SUM(gets) 2 "data dictionary hit ratio" from v$rowcache;
  2 -------------------------
  2 .936781093
SQL>
```
Usually, it’s a good idea to shoot for a dictionary hit ratio as high as 95 to 99 percent, although Oracle itself sometimes seems to refer to a figure of 85 percent as being adequate. To increase the library cache ratio, you simply increase the shared pool size for the instance.

**Hard Parsing and Soft Parsing**

You may recall from the last chapter that all SQL code goes through the parse, optimize, and execute phases. When an application issues a statement, Oracle first sees whether a parsed version of the statement already exists. If it does, the result is a so-called soft parse and is considered a library cache hit. If, during a parse phase or the execution phase, Oracle isn’t able to find the parsed version or the executable version of the code in the shared pool, it will perform a *hard parse*, which means that the SQL statement has to be reloaded into the shared pool and parsed completely.

During a hard parse, Oracle performs syntactic and semantic checking, checks the object and system privileges, builds the optimal execution plan, and finally loads it into the library cache. A hard parse involves a lot more CPU usage and is inefficient compared to a soft parse, which depends on reusing previously parsed statements. Hard parsing involves building all parse information from scratch, and therefore it’s more resource intensive. Besides involving a higher CPU usage, hard parsing involves a large number of latch gets, which may increase the response time of the query. The ideal situation is where you parse once and execute many times. Otherwise, Oracle has to perform a hard parse.

---

**Caution** High hard parse rates lead to severe performance problems, so it’s critical that you reduce hard parse counts in your database.

---

A soft parse simply involves checking the library cache for an identical statement and reusing it. The major step of optimizing the SQL statement is completely omitted during a soft parse. There’s no parsing (as done during a hard parse) during a soft parse, because the new statement is hashed and its hash value is compared with the hash values of similar statements in the library cache. During a soft parse, Oracle only checks for the necessary privileges. For example, even if there’s an identical statement in the library cache, your statement may not be executed if Oracle determines during the (soft) parsing stage that you don’t have the necessary privileges. Oracle recommends that you treat a hard parse rate of more than 100 per second as excessive.

**Using SQL Trace and TKPROF to Examine Parse Information**

In Chapter 19, you learned how to use the SQL Trace and TKPROF utilities to trace SQL statement execution. One of the most useful pieces of information the SQL Trace utility provides concerns the hard and soft parsing information for a query. The following simple example demonstrates how you can derive the parse information for any query:

1. Enable tracing in the session by using the following command:

   ```sql
   SQL> ALTER SESSION SET SQL_TRACE=TRUE;
   Session altered.
   SQL>
   ```

   To make sure none of your queries were parsed before, flush the shared pool, which removes all SQL statements from the library cache:

   ```sql
   SQL> ALTER SYSTEM FLUSH SHARED_POOL;
   System altered.
   SQL>
   ```
2. Use the following query to create a trace in the user dump directory:

   SQL> SELECT * FROM comp_orgs WHERE created_date > SYSDATE-5;

   The SQL Trace output shows the following in the output file:

   PARSING IN CURSOR #1 len=63 dep=0 uid=21 oct=3
   lid=21 tim=1326831345 hv=71548308
   SELECT * FROM comp_orgs WHERE created_date > SYSDATE-:"SYS_B_0"
   END OF STMT
   PARSE #1:c=4,e=4,p=0,cr=57,cu=3,mis=1,r=0,dep=0,og=0,tim=1326831345
   Note that mis=1 indicates a hard parse because this SQL isn't present in the library cache.

3. Use a slightly different version of the previous query next. The output is the same, but Oracle won't use the previously parsed version, because the statements in steps 2 and 3 aren't identical.

   SQL> SELECT * FROM comp_orgs WHERE created_date > (SYSDATE -5);

   Here's the associated SQL Trace output:

   PARSING IN CURSOR #1 len=77 dep=0 uid=21 oct=3
   lid=21 tim=1326833972
   SELECT /* A Hint */ * FROM comp_orgs WHERE
   created_date > SYSDATE-:"SYS_B_0"
   END OF STMT
   PARSE #1:c=1,e=1,p=0,cr=57,cu=3,mis=1,r=0,dep=0,og=0,tim=1326833972
   Again, a hard parse, indicated by mis=1, shows a library cache miss. This isn't a surprise, as this statement isn't identical to the one before, so it has to be parsed from scratch.

4. Use the original query again. Now Oracle performs only a soft parse, because the statements here and in the first step are the same. Here's the SQL Trace output:

   PARSING IN CURSOR #1 len=63 dep=0 uid=21 oct=3
   lid=21 tim=1326834357
   SELECT * FROM comp_orgs WHERE created_date > SYSDATE-:"SYS_B_0"
   END OF STMT
   PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1326834357
   The statement in step 4 is identical in all respects to the statement in step 1, so Oracle reuses the parsed version. Hence mis=0 indicates there wasn’t a hard parse but merely a soft parse, which is a lot cheaper in terms of resource usage.

   If you now look at the TKPROF output, you'll see the following section for the SQL statements in step 2 and step 4 (identical statements):

   *********************************************************
   SELECT * FROM comp_orgs WHERE created_date > SYSDATE - 5
   call count cpu elapsed disk query current rows
   ------- ------ -------- --------- --------- ------- ---------- ---------- ----------
   Parse    2  0.03  0.01     0        1       3       0
   Execute   2  0.00  0.00     0        0       0       0
   Fetch    4  0.07  0.10  156  166    24     10
   total    8  0.10  0.11  156  167    27     10
   Misses in library cache during parse: 1
   *********************************************************

   As you can see, there was one miss in the library cache when you first executed the statement. The second time around, there was no hard parse and hence no library cache miss.
Measuring Library Cache Efficiency

You can use simple ratios to see if your library cache is sized correctly. The V$LIBRARYCACHE data dictionary view provides you with all the information you need to see whether the library cache is efficiently sized. Listing 20-1 shows the structure of the V$LIBRARYCACHE view.

Listing 20-1. The V$LIBRARYCACHE View

```
SQL> DESC V$LIBRARYCACHE
Name                                      Null?    Type
----------------------------------------- -------- ------------
NAMESPACE                                          VARCHAR2(15)
GETS                                               NUMBER
GETHITS                                            NUMBER
GETHITRATIO                                        NUMBER
PINS                                               NUMBER
PINHITS                                            NUMBER
PINHITRATIO                                        NUMBER
RELOADS                                            NUMBER
INVALIDATIONS                                      NUMBER
DLM_LOCK_REQUESTS                                  NUMBER
DLM_PIN_REQUESTS                                   NUMBER
DLM_PIN_RELEASES                                   NUMBER
DLM_INVALIDATION_REQUESTS                          NUMBER
DLM_INVALIDATIONS                                  NUMBER
SQL> 
```

The following formula provides you with the library cache hit ratio:

```
SQL> SELECT SUM(pinhits)/sum(pins) Library_cache_hit_ratio 
   2  FROM V$LIBRARYCACHE;
```

```
LIBRARY_CACHE_HIT_RATIO
------------------------
.993928013
SQL> 
```

The formula indicates that the library cache currently has a higher than 99 percent hit ratio, which is considered good. However, be cautious about relying exclusively on high hit ratios for the library cache and the buffer caches, such as the one shown here. You may have a hit ratio such as 99.99 percent, but if significant waits are caused by events such as excessive parsing, you’re going to have a slow database. Always keep an eye on the wait events in your system, and don’t rely blindly on high hit ratios such as these.

Listing 20-2 shows how to determine the number of reloads and pinhits of various statements in your library cache.

Listing 20-2. Determining the Efficiency of the Library Cache

```
SQL> SELECT namespace, pins, pinhits, reloads 
   2  FROM V$LIBRARYCACHE 
   3  ORDER BY namespace;
```
<table>
<thead>
<tr>
<th>NAMESPACE</th>
<th>PINS</th>
<th>PINHITS</th>
<th>RELOADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BODY</td>
<td>25</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>CLUSTER</td>
<td>248</td>
<td>239</td>
<td>0</td>
</tr>
<tr>
<td>INDEX</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JAVA DATA</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>JAVA RESOURCE</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>JAVA SOURCE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OBJECT</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PIPE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SQL AREA</td>
<td>390039</td>
<td>389465</td>
<td>14</td>
</tr>
<tr>
<td>TABLE/PROCEDURE</td>
<td>3532</td>
<td>1992</td>
<td>0</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

If the RELOADS column of the V$LIBRARYCACHE view shows large values, it means that many SQL statements are being reloaded into the library pool after they've been aged out. You might want to increase your shared pool, but this still may not do the trick if the application is large, the number of executions is large, or the application doesn’t use bind variables. If the SQL statements aren’t exactly identical and/or if they use constants instead of bind variables, more hard parses will be performed, and hard parses are inherently expensive in terms of resource usage. You can force the executable SQL statements to remain in the library cache component of the shared pool by using the Oracle-provided DBMS_SHARED_POOL package. The package has the KEEP and UNKEEP procedures; using these you can retain and release objects in the shared pool.

You can use the V$LIBRARY_CACHE_MEMORY view to determine the number of library cache memory objects currently in use in the shared pool and to determine the number of freeable library cache memory objects in the shared pool. The V$SHARED_POOL_ADVICE view provides you with information about the parse time savings you can expect for various sizes of the shared pool.

Optimizing the Library Cache

You can configure some important initialization parameters so the library cache areas are used efficiently. You’ll look at some of these initialization parameters in the following sections.

Using the CURSOR_SHARING (Literal Replacement) Parameter

The key idea behind optimizing the use of the library cache is to reuse previously parsed or executed code. One of the easiest ways to do this is to use bind variables rather than literal statements in the SQL code. Bind variables are like placeholders: they allow binding of application data to the SQL statement. Using bind variables enables Oracle to reuse statements when the only things changing in the statements are the values of the input variables. Bind variables enable you to reuse the cached, parsed versions of queries and thus speed up your application. Here’s an example of the use of bind variables. The following code sets up a bind variable as a number type:

```sql
SQL> VARIABLE bindvar NUMBER;
SQL> BEGIN
  2  :bindvar :=7900;
  3  END;
  4 /
PL/SQL procedure successfully completed.
SQL>
```
You can now issue the following SQL statement that makes use of the bind variable you just created:

```
SQL> SELECT ename FROM scott.emp WHERE empid = :bindvar;
ENAME
JAMES
```

You can execute this statement multiple times with different values for the bind variable. The statement is parsed only once and executes many times. Unlike when you use a literal value for the `emp_id` column (7499, for example), Oracle reuses the execution plan it created the first time, instead of creating a separate execution plan for each such statement. This cuts hard parsing (and high latch activity) and the attendant CPU usage drastically, and dramatically reduces the time taken to retrieve data. For example, all the following statements can use the parsed version of the query that uses the bind variable:

```
SELECT ename FROM scott.emp WHERE empid = 7499;
SELECT ename FROM scott.emp WHERE empid = 7788;
SELECT ename FROM scott.emp WHERE empid = 7902;
```

Unfortunately, in too many applications, literal values rather than bind values are used. You can alleviate this problem to some extent by setting up the following initialization parameter:

```
CURSOR_SHARING=FORCE
```

Or you could use the following parameter:

```
CURSOR_SHARING=SIMILAR
```

By default, the `CURSOR_SHARING` initialization parameter is set to `EXACT`, meaning that only statements that are identical in all respects will be shared among different executions of the statement. Either of the alternative values for the `CURSOR_SHARING` parameter, `FORCE` or `SIMILAR`, ensures Oracle will reuse statements even if they aren’t identical in all respects.

For example, if two statements are identical in all respects and differ only in literal values for some variables, using `CURSOR_SHARING=FORCE` will enable Oracle to reuse the parsed SQL statements in its library cache. Oracle replaces the literal values with bind values to make the statements identical. The `CURSOR_SHARING=FORCE` option forces the use of bind variables under all circumstances, whereas the `CURSOR_SHARING=SIMILAR` option does so only when Oracle thinks doing so won’t adversely affect optimization. Oracle recommends the use of `CURSOR_SHARING=SIMILAR` rather than `CURSOR_SHARING=FORCE` because of possible deterioration in the execution plans. However, in reality, the benefits provided by the `CURSOR_SHARING=FORCE` parameter far outweigh any possible damage to the execution plans. You can improve the performance of your database dramatically when you notice a high degree of hard parsing due to failing to use bind variables by moving from the default `CURSOR_SHARING=EXACT` option to the `CURSOR_SHARING=FORCE` option. You can change the value of this parameter in the `init.ora` file or `SPFILE`, or you can do so dynamically by using the `ALTER SYSTEM` (instance-wide) statement or the `ALTER SESSION` (session-level) statement.

By allowing users to share statements that differ only in the value of the constants, the `CURSOR_SHARING` parameter enables the Oracle database to scale easily to a large number of users who are using similar, but not identical, SQL statements. This major innovation started in the Oracle8i version.

**Sessions with a High Number of Hard Parses**

The query in Listing 20-3 enables you to find out how the hard parses compare with the number of executions since the instance was started. It also tells you the session ID for the user using the SQL statements.
Listing 20-3. Determining Sessions with a High Number of Parses

SQL> SELECT s.sid, s.value "Hard Parses", t.value "Executions Count"
2  FROM v$sesstat s, v$sesstat t
3  WHERE s.sid=t.sid
4  AND s.statistic#=(select statistic#
5  FROM v$statname where name='parse count (hard)')
6  AND t.statistic#=(select statistic#
7  FROM v$statname where name='execute count')
8  AND s.value>0
9  ORDER BY 2 desc;

<table>
<thead>
<tr>
<th>SID</th>
<th>Hard Parses</th>
<th>Executions Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1696</td>
<td>70750</td>
<td>3638104</td>
</tr>
<tr>
<td>1750</td>
<td>12188</td>
<td>262881</td>
</tr>
<tr>
<td>1759</td>
<td>3555</td>
<td>5895488</td>
</tr>
<tr>
<td>1757</td>
<td>3265</td>
<td>2758185</td>
</tr>
<tr>
<td>1694</td>
<td>1579</td>
<td>2389953</td>
</tr>
</tbody>
</table>

Using the CURSOR_SPACE_FOR_TIME Parameter

By default, cursors can be deallocated even when the application cursors aren’t closed. This forces an increase in Oracle’s overhead because of the need to check whether the cursor is flushed from the library cache. The parameter that controls whether this deallocation of cursors takes place is the CURSOR_SPACE_FOR_TIME initialization parameter, whose default value is FALSE. If you set this parameter to TRUE, you ensure that the cursors for the application cannot be deallocated while the application cursors are still open. The initialization parameter in the init.ora file should be as follows:

CURSOR_SPACE_FOR_TIME=TRUE

---

**Tip** If you want to set this parameter, make sure that you have plenty of free shared pool memory available, because this parameter will use more shared pool memory for saving the cursors in the library cache.

---

Using the SESSION_CACHED_CURSORS Parameter

Ideally, an application should have all the parsed statements available in separate cursors, so that if it has to execute a new statement, all it has to do is pick the parsed statement and change the value of the variables. If the application reuses a single cursor with different SQL statements, it still has to pay the cost of a soft parse. After opening a cursor for the first time, Oracle will parse the statement, and then it can reuse this parsed version in the future. This is a much better strategy than re-creating the cursor each time the database executes the same SQL statement. If you can cache all the cursors, you’ll retain the server-side context, even when clients close the cursors or reuse them for new SQL statements.

You’ll appreciate the usefulness of the SESSION_CACHED_CURSORS parameter in a situation where users repeatedly parse the same statements, as happens in an Oracle Forms-based application when users switch among various forms. Using the SESSION_CACHED_CURSORS parameter ensures that for
any cursor for which more than three parse requests are made, the parse requests are automatically cached in the session cursor cache. Thus new calls to parse the same statement avoid the parsing overhead. Using the initialization parameter SESSION_CACHED_CURSORS and setting it to a high number makes the query processing more efficient. Although soft parses are cheaper than hard parses, you can reduce even soft parsing by using the SESSION_CACHED_CURSORS parameter and setting it to a high number.

You can enforce session caching of cursors by setting the SESSION_CACHED_CURSORS in your initialization parameter file, or dynamically by using the following ALTER SESSION command:

```
SQL> ALTER SESSION SET SESSION_CACHED_CURSORS = value;
```

You can check how good your SESSION_CACHED_CURSORS parameter value is by using the V$SYSSTAT view. If the value of session cursor cache hits is low compared to the total parse count for a session, then the SESSION_CACHED_CURSORS parameter value should be bumped up.

The perfect situation is where a SQL statement is soft parsed once in a session and executed multiple times. For a good explanation of bind variables, cursor sharing, and related issues, please read the Oracle white paper “Efficient use of bind variables, cursor_sharing and related cursor parameters” (http://otn.oracle.com/deploy/performance/pdf/cursor.pdf).

### Parsing and Scaling Applications

When the number of users keeps increasing, some systems have trouble coping. Performance slows down dramatically in many systems as a result of trying to scale to increased user populations. When your user counts are increasing, focus on unnecessary parsing in your system. A high level of parsing leads to latch contention, which slows down the system. Here are some guidelines that help summarize the previous discussion about the library cache, parsing, and the use of special initialization parameters:

- A standard rule is to put as much of the code as possible in the form of stored code—packages, procedures, and functions—so you don’t have the problems caused by ad hoc SQL. Use of ad hoc SQL could wreak havoc with your library cache, and it’s an inefficient way to run a large application with many users. Using stored code guarantees that code is identical and thus reused, thereby enhancing scalability.

- Lower the number of hard parses, as they could be expensive. One way to convert a hard parse to a soft parse is to use bind variables, as you saw earlier in this chapter. Reducing hard parsing reduces shared-pool latch contention.

- If bind variables aren’t being used in your system, you can use the CURSOR_SHARING=FORCE parameter to force the sharing of SQL statements that differ only in the value of literals.

- Pay attention to the amount of soft parsing, not the per unit cost, which is much lower than that of a hard parse. A high amount of soft parsing increases contention for the library cache latch and could lead to a slow-performing database. The point to note here is to avoid any unnecessary soft parsing, which will end up costing you.

- Use the SESSION_CACHED_CURSORS initialization parameter to reuse the open cursors in a session. If repeated parse calls are used for a SQL statement, Oracle moves the session cursor for that statement into the session cursor cache. This, as you’ve seen, reduces the amount of soft parsing. Set the value of this parameter to somewhere between the value of the OPEN_CURSORS initialization parameter and the number of cursors that are being used in the session.

- Use the CURSOR_SPACE_FOR_TIME initialization parameter (set it to TRUE) to prevent the early deallocation of cursors. If you don’t mind the extra cost of using more memory, this feature will enhance your application’s scalability level.
• Reduce the amount of session logging on/off activity by users. This may reduce scalability due to the increased amount of overhead involved in authenticating the user, verifying privileges, and so on, leading to a waste of time and resources. Furthermore, the users may be spending more time trying to log into the system than executing their SQL statements. Frequent logging off and logging back on might also cause contention for the web server and other resources, and increase the time it takes to log into your system.

• To increase scalability, you must also ensure that applications share sessions. If you only have shared SQL, your hard parses will go down, but your soft parses might still be high. If an application program can maintain a persistent connection to the Oracle server, it doesn’t have to perform repeated soft parsing to reuse code.

Sizing the Shared Pool

The best way to set the size of the shared pool in Oracle Database 11g is to let Oracle do all the work for you by using the MEMORY_TARGET initialization parameter, thus automating the management of SGA. You can initially set the SGA_TARGET parameter at something close to the total SGA you would have allocated under a manual management mode. Review the material in Chapter 17 for guidance on setting your initial MEMORY_TARGET value.

Pinning Objects in the Shared Pool

As I have discussed, if code objects have to be repeatedly hard-parsed and executed, database performance will deteriorate eventually. Your goal should be to see that as much of the executed code remains in memory as possible so compiled code can be reexecuted. You can avoid repeated reloading of objects in your library cache by pinning objects using the DBMS_SHARED_POOL package. (The library cache is a component of the shared pool, as you’ve seen earlier.) Listing 20-4 shows how you can determine the objects that should be pinned in your library cache (shared pool).

Listing 20-4. Determining the Objects to Be Pinned in the Shared Pool

```sql
SQL> SELECT type, COUNT(*) OBJECTS,
2  SUM(DECODE(KEPT,'YES',1,0)) KEPT,
3  SUM(loads) - COUNT(*) reloads
4  FROM V$DB_OBJECT_CACHE
5  GROUP BY type
6* ORDER BY objects DESC;
```

<table>
<thead>
<tr>
<th>TYPE</th>
<th>OBJECTS</th>
<th>KEPT</th>
<th>RELOADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR</td>
<td>41143</td>
<td>0</td>
<td>136621</td>
</tr>
<tr>
<td>NOT LOADED</td>
<td>37522</td>
<td>0</td>
<td>54213</td>
</tr>
<tr>
<td>TABLE</td>
<td>758</td>
<td>24</td>
<td>133742</td>
</tr>
<tr>
<td>PUB_SUB</td>
<td>404</td>
<td>0</td>
<td>135</td>
</tr>
<tr>
<td>SYNONYM</td>
<td>381</td>
<td>0</td>
<td>7704</td>
</tr>
<tr>
<td>JAVA CLASS</td>
<td>297</td>
<td>296</td>
<td>317</td>
</tr>
<tr>
<td>VIEW</td>
<td>181</td>
<td>0</td>
<td>11586</td>
</tr>
<tr>
<td>INVALID TYPE</td>
<td>139</td>
<td>48</td>
<td>11</td>
</tr>
<tr>
<td>PACKAGE</td>
<td>137</td>
<td>0</td>
<td>8352</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>136</td>
<td>0</td>
<td>8515</td>
</tr>
<tr>
<td>PACKAGE BODY</td>
<td>121</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>81</td>
<td>0</td>
<td>3015</td>
</tr>
<tr>
<td>INDEX</td>
<td>61</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>41</td>
<td>0</td>
<td>219</td>
</tr>
</tbody>
</table>
If the number of reloads in the output shown in Listing 20-4 is high, you need to make sure that the objects are pinned using the following command:

```sql
EXECUTE SYS.DBMS_SHARED_POOL.KEEP(object_name,object_type);
```

You can use the following statements to pin a package first in the shared pool and then remove it, if necessary:

```sql
EXECUTE SYS.DBMS_SHARED_POOL.KEEP(NEW_EMP.PKG, PACKAGE);
EXECUTE SYS.DBMS_SHARED_POOL.UNKEEP(NEW_EMP.PKG, PACKAGE);
```

Of course, if you shut down and restart your database, the shared pool won’t retain the pinned objects. That’s why most DBAs use scripts with all the objects they want to pin in the shared pool and schedule them to run right after every database start. Most of the objects usually are small, so there’s no reason to be too conservative about how many you pin. For example, I pin all my packages, including Oracle-supplied PL/SQL packages.

Look at the following example, which gives you an idea about the total memory taken up by a large number of packages. This query shows the total number of packages in my database:

```sql
SELECT COUNT(*) FROM V$DB_OBJECT_CACHE WHERE type='PACKAGE';
```

```
COUNT(*)
-------
167
```

The following query shows the total amount of memory needed to pin all my packages in the shared pool:

```sql
SELECT SUM(sharable_mem) FROM V$DB_OBJECT_CACHE WHERE type='PACKAGE';
```

```
SUM(SHARABLE_MEM)
------------------
4771127
```

As you can see, pinning every single package in my database takes up less than 5MB of a total of several hundred megabytes of memory allocated to the shared pool.
Tuning the Buffer Cache

When users request data, Oracle reads the data from the disks (in terms of Oracle blocks) and stores it in the buffer cache so it may access the data easily if necessary. As the need for the data diminishes, eventually Oracle removes the data from the buffer cache to make room for newer data. Note that some operations don’t use the buffer cache (SGA); rather, they read directly into the PGA area. Direct sort operations and parallel reads are examples of such operations.

How to Size the Buffer Cache

As with the shared pool component, the best way to manage the buffer cache is to choose automatic SGA management. However, if you choose to manage the SGA manually, you can use a process of trial and error to set the buffer cache size. You assign an initial amount of memory to the pool and watch the buffer cache hit ratios to see how often the application can retrieve the data from memory, as opposed to going to disk. The terminology used for calculating the buffer hit ratio can be somewhat confusing on occasion. Here are the key terms you need to understand:

- **Physical reads**: These are the data blocks that Oracle reads from disk. Reading data from disk is much more expensive than reading data that’s already in Oracle’s memory. When you issue a query, Oracle always first tries to retrieve the data from memory—the database buffer cache—and not disk.

- **DB block gets**: This is a read of the buffer cache, to retrieve a block in current mode. This most often happens during data modification when Oracle has to be sure that it’s updating the most recent version of the block. So, when Oracle finds the required data in the database buffer cache, it checks whether the data in the blocks is up to date. If a user changes the data in the buffer cache but hasn’t committed those changes yet, new requests for the same data can’t show these interim changes. If the data in the buffer blocks is up to date, each such data block retrieved is counted as a DB block get.

- **Consistent gets**: This is a read of the buffer cache, to retrieve a block in consistent mode. This may include a read of undo segments to maintain the read consistency principle (see Chapter 8 for more information about read consistency). If Oracle finds that another session has updated the data in that block since the read began, then it will apply the new information from the undo segments.

- **Logical reads**: Every time Oracle is able to satisfy a request for data by reading it from the database buffer cache, you get a logical read. Thus logical reads include both DB block gets and consistent gets.

- **Buffer gets**: This term refers to the number of database cache buffers retrieved. This value is the same as the logical reads described earlier.

The following formula gives you the buffer cache hit ratio:

\[
1 - \left( \frac{\text{'physical reads cache'}}{\text{'consistent gets from cache'} + \text{'db block gets from cache'}} \right)
\]

You can use the following query to get the current values for all three necessary buffer cache statistics:

```
SQL> SELECT name, value FROM v$sysstat
WHERE name IN ('physical reads cache',
               'consistent gets from cache',
               'db block gets from cache');
```
The following calculation, based on the statistics I derived in the preceding code from the V$SYSSTAT view, show that the buffer cache hit ratio for my database is a little over 91 percent:

\[
1 - \frac{(505726180)}{(103264732 + 5924585494)} = .916101734
\]

As you can see from the formula for the buffer cache hit ratio, the lower the ratio of physical reads to the total logical reads, the higher the buffer cache hit ratio.

You can use the V$BUFFER_POOL_STATISTICS view, which lists all buffer pools for the instance, to derive the hit ratio for the buffer cache:

```
SQL> SELECT NAME, PHYSICAL_READS, DB_BLOCK_GETS, CONSISTENT_GETS,
  1 - (PHYSICAL_READS/(DB_BLOCK_GETS + CONSISTENT_GETS)) "HitRatio"
  FROM V$BUFFER_POOL_STATISTICS;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>PHYSICAL_READS</th>
<th>DB_BLOCK_GETS</th>
<th>CONSISTENT_GETS</th>
<th>HitRatio</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>50587859</td>
<td>103275634</td>
<td>5924671178</td>
<td>.991607779</td>
</tr>
</tbody>
</table>

SQL>

In addition, you can use the Database Control's Memory Advisor to get advice regarding the optimal buffer cache size. The advice is presented in a graphical format, showing the trade-off between increasing the SGA and the reduction in DB time. You can use the V$DB_CACHE_ADVICE view (use V$SGA_TARGET_ADVICE to size the SGA_TARGET size) to see how much you need to increase the buffer cache to lower the physical I/O by a certain amount. Essentially, the output of the V$DB_CACHE_ADVICE view shows you how much you can increase your buffer cache memory before the gains in terms of a reduction in the amount of physical reads (estimated) are insignificant. The Memory Advisor simulates the miss rates in the buffer cache for caches of different sizes. In this sense, the Memory Advisor can keep you from throwing excess memory in a vain attempt at lowering the amount of physical reads in your system.

Oracle blocks used during a full table scan involving a large table are aged out of the buffer cache faster than Oracle blocks from small-table full scans or indexed access. Oracle may decide to keep only part of the large table in the buffer cache to avoid having to flush out its entire buffer cache. Thus, your buffer cache hit ratio would be artificially low if you were using several large-table full scans. If your application involves many full table scans for some reason, increasing the buffer cache size isn’t going to improve performance. Some DBAs are obsessed about achieving a high cache hit ratio, such as 99 percent or so. A high buffer cache hit ratio is no guarantee that your application response time and throughput will also be high. If you have a large number of full table scans or if your database is more of a data warehouse than an OLTP system, your buffer cache may be well below 100 percent, and that’s not a bad thing. If your database consists of inefficient SQL, there will be an inordinately high number of logical reads, making the buffer cache hit ratio look good (say 99.99 percent), but this may not mean your database is performing efficiently. Please read the interesting article by Cary Millsap titled "Why a 99%+ Database Buffer Cache Hit Ratio Is Not Ok" (http://www.hotsos.com/e-library/abstract.php?id=6).
Using Multiple Pools for the Buffer Cache

You don’t have to allocate all the buffer cache memory to a single pool. As Chapter 10 showed you, you can use three separate pools: the keep buffer pool, the recycle buffer pool, and the default buffer pool. Although you don’t have to use the keep and default buffer pools, it’s a good idea to configure all three pools so you can assign objects to them based on their access patterns. In general, you follow these rules of thumb when you use the multiple buffer pools:

- Use the recycle cache for large objects that are infrequently accessed. You don’t want these objects to occupy a large amount of space unnecessarily in the default pool.
- Use the keep cache for small objects that you want in memory at all times.
- Oracle automatically uses the default pool for all objects not assigned to either the recycle or keep cache.

Since version 8.1, Oracle has used a concept called touch count to measure how many times an object is accessed in the buffer cache. This algorithm of using touch counts for managing the buffer cache is somewhat different from the traditional modified LRU algorithm that Oracle used to employ for managing the cache. Each time a buffer is accessed, the touch count is incremented. A low touch count means that the block isn’t being reused frequently, and therefore is wasting database buffer cache space. If you have large objects that have a low touch count but occupy a significant proportion of the buffer cache, you can consider them ideal candidates for the recycle pool. Listing 20-5 contains a query that shows you how to find out which objects have a low touch count. The TCH column in the x$bh table owned by the user SYS indicates the touch count.

Listing 20-5. Determining Candidates for the Recycle Buffer Pool

```sql
SELECT obj object, COUNT(1) buffers, (COUNT(1)/totsize) * 100 percent_cache FROM x$bh,
(SELECT value totsize FROM v$parameter WHERE name = 'db_block_buffers') WHERE tch=1 OR (tch = 0 and lru_flag < 10) GROUP BY obj, totsize HAVING (COUNT(1)/totsize) * 100 > 5
```

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>BUFFERS</th>
<th>PERCENT_CACHE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1386</td>
<td>14288</td>
<td>5.95333333</td>
</tr>
<tr>
<td>1412</td>
<td>12616</td>
<td>5.25666667</td>
</tr>
<tr>
<td>613114</td>
<td>22459</td>
<td>9.35791667</td>
</tr>
</tbody>
</table>

The preceding query shows you that three objects, each with a low touch count, are taking up about 20 percent of the total buffer cache. Obviously, they’re good candidates for the recycle buffer pool. In effect, you’re limiting the number of buffers the infrequently used blocks from these three tables can use up in the buffer cache.

The following query on the DBA_OBJECTS view gives you the names of the objects:
SQL> SELECT object_name FROM DBA_OBJECTS
2  WHERE object_id IN (1386,1412,613114);

OBJECT_NAME
-------------
EMPLOYEES
EMPLOYEE_HISTORY
FINANCE_RECS
SQL>

You can then assign these three objects to the reserved buffer cache pool. You can use a similar
criterion to decide which objects should be part of your keep buffer pool. Say you want to pin all
objects in the keep pool that occupy at least 25 buffers and have an average touch count of more than
5. Listing 20-6 shows the query that you should run as the user SYS.

Listing 20-6. Determining Candidates for the Keep Buffer Cache

SQL> SELECT obj object,
2  count(1) buffers,
3  AVG(tch) average_touch_count
4  FROM x$bh
5  WHERE lru_flag = 8
6  GROUP BY obj
7  HAVING avg(tch) > 5
8* AND count(1) > 25;

OBJECT     BUFFERS      AVERAGE_TOUCH_COUNT
----------   ----------    --------------------
1349785         36                        67
4294967295         87                 57.137931
SQL>

Again, querying the DBA_OBJECTS view provides you with the names of the objects that are
candidates for the keep buffer cache pool.

Here’s a simple example to show how you can assign objects to specific buffer caches (keep and
recycle). First, make sure you configure the keep and recycle pools in your database by using the
following set of initialization parameters:

DB_CACHE_SIZE=256MB
DB_KEEP_CACHE_SIZE=16MB
DB_RECYCLE_CACHE_SIZE=16MB

In this example, the keep and recycle caches are 16MB each. Once you create the keep and
recycle pools, it’s easy to assign objects to these pools. All tables are originally in the default buffer
cache, where all tables are cached automatically unless specified otherwise in the object creation
statement.

You can use the ALTER TABLE statement to assign any table or index to a particular type of buffer
cache. For example, you can assign the following two tables to the keep and recycle buffer caches:

SQL> ALTER TABLE test1 STORAGE (buffer_pool keep);
Table altered.

SQL> ALTER TABLE test2 STORAGE (buffer_pool recycle);
Table altered.
SQL>
Tuning the Large Pool, Streams Pool, and Java Pool

You mainly use the large pool, an optional component of the SGA, in shared server systems for session memory, for facilitating parallel execution for message buffers, and for backup processes for disk I/O buffers. Oracle recommends the use of the large pool if you’re using shared server processes so you can keep the shared pool fragmentation low. If you’re using shared server configurations, you should configure the large pool. The streams pool is relevant only if you’re using the Oracle Streams feature. You don’t have to bother with tuning the Java pool allocation unless you’re using heavy Java applications.

Note You size the large pool based on the number of active simultaneous sessions in a shared server environment. Remember that if you’re using the shared server configuration and you don’t specify a large pool, Oracle will allocate memory to the shared sessions out of your shared pool.

Tuning PGA Memory

Each server process serving a client is allocated a private memory area, the PGA, most of which is dedicated to memory-intensive tasks such as group by, order by, rollup, and hash joins. The PGA area is a nonshared area of memory created by Oracle when a server process is started, and it’s automatically deallocated upon the end of that session. Operations such as in-memory sorting and building hash tables need specialized work areas. The memory you allocate to the PGA determines the size of these work areas for specialized tasks, such as sorting, and determines how fast the system can finish them. In the following sections you’ll examine how you can decide on the optimal amount of PGA for your system.

Automatic PGA Memory Management

The management of the PGA memory allocation is easy from a DBA’s point of view. You can set a couple of basic parameters and let Oracle automatically manage the allocation of memory to the individual work areas. You need to do a couple things before Oracle can automatically manage the PGA. You need to use the PGA_AGGREGATE_TARGET parameter to set the memory limit, and you need to use the V$PGA_TARGET_ADVICE view to tune the target’s value. In the next sections I discuss those tasks.

Using the PGA_AGGREGATE_TARGET Parameter

The PGA_AGGREGATE_TARGET parameter in the init.ora file sets the maximum limit on the total memory allocated to the PGA. Oracle offers the following guidelines on sizing the PGA_AGGREGATE_TARGET parameter:
• For an OLTP database, the target should be 16 to 20 percent of the total memory allocated to Oracle.
• For a DSS database, the target should be 40 to 70 percent of the total memory allocated to Oracle.

The preceding guidelines are just that—guidelines. The best way to determine the ideal size of the PGA_AGGREGATE_TARGET parameter is to use the V$PGA_TARGET_ADVICE or V$PGASTAT view, which I explain in the following sections.

**Using the V$PGA_TARGET_ADVICE View**

Once you’ve set the initial allocation for the PGA memory area, you can use the V$PGA_TARGET_ADVICE view to tune the target’s value. Oracle populates this view with the results of its simulations of different workloads for various PGA target levels. You can then query the view as follows:

```sql
SELECT ROUND(pga_target_for_estimate/1024/1024) target_mb,
     estd_pga_cache_hit_percentage cache_hit_perc,
     estd_overalloc_count
FROM V$PGA_TARGET_ADVICE;
```

Using the estimates from the V$PGA_TARGET_ADVICE view, you can then set the optimal level for PGA memory.

**Setting the Value of the PGA_AGGREGATE_TARGET Parameter**

Remember that the memory you provide through setting the PGA_AGGREGATE_TARGET parameter is what determines the efficiency of sorting and hashing operations in your database. If you have a large number of users who perform heavy-duty sort or hash operations, your PGA_AGGREGATE_TARGET must be set at a high level. When you set the SGA_TARGET at, say 2GB, the instance takes the 2GB from the total OS memory as soon as you start it. However, the PGA_AGGREGATE_TARGET is merely a target. Oracle doesn’t take all the memory you assign to the PGA_AGGREGATE_TARGET when the instance starts. The PGA_AGGREGATE_TARGET only serves as the upper bound on the total private or work-area memory the instance can allocate to all the sessions combined.

The ideal way to perform sorts is by doing the entire job in memory. A sort job that Oracle performs entirely in memory is said to be an optimal sort. If you set the PGA_AGGREGATE_TARGET too low, some of the sort data is written out directly to disk (temporary tablespace) because the sorts are too large to fit in memory. If only part of a sort job spills over to disk, it’s called a one-pass sort. If the instance performs most of the sort on disk instead of in memory, the response time will be high. Luckily, as long as you have enough memory available, you can monitor and avoid problems due to the undersizing of the PGA memory (PGA_TARGET).

You can examine the PGA usage within your database by using the following query. The value column shows, in bytes, the amount of memory currently allocated to the various users:

```sql
SELECT s.value, s.sid, a.username
FROM V$SESSTAT S, V$STATNAME N, V$SESSION A
WHERE n.STATISTIC# = s.STATISTIC# and
     name = 'session pga memory'
     AND s.sid=a.sid
ORDER BY s.value;
```
An important indicator of the efficiency of the PGA_TARGET parameter is the PGA “hit ratio,” shown in the last row of the following query, which uses the V$PGASTAT view:

```sql
SELECT * FROM V$PGASTAT;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate PGA target parameter</td>
<td>49999872</td>
<td>bytes</td>
</tr>
<tr>
<td>aggregate PGA auto target</td>
<td>4194304</td>
<td>bytes</td>
</tr>
<tr>
<td>global memory bound</td>
<td>2499584</td>
<td>bytes</td>
</tr>
<tr>
<td>total PGA inuse</td>
<td>67717120</td>
<td>bytes</td>
</tr>
<tr>
<td>total PGA allocated</td>
<td>161992704</td>
<td>bytes</td>
</tr>
<tr>
<td>maximum PGA allocated</td>
<td>244343808</td>
<td>bytes</td>
</tr>
<tr>
<td>total freeable PGA memory</td>
<td>16121856</td>
<td>bytes</td>
</tr>
<tr>
<td>PGA memory freed back to OS</td>
<td>6269370368</td>
<td>bytes</td>
</tr>
<tr>
<td>total PGA used for auto workareas</td>
<td>0</td>
<td>bytes</td>
</tr>
<tr>
<td>maximum PGA used for auto</td>
<td>6843392</td>
<td>bytes</td>
</tr>
<tr>
<td>total PGA used for manual workareas</td>
<td>0</td>
<td>bytes</td>
</tr>
<tr>
<td>maximum PGA used for manual workareas</td>
<td>530432</td>
<td>bytes</td>
</tr>
<tr>
<td>over allocation count</td>
<td>1146281</td>
<td>bytes</td>
</tr>
<tr>
<td>processed</td>
<td>4.4043E+10</td>
<td>bytes</td>
</tr>
<tr>
<td>extra bytes read/written</td>
<td>7744561152</td>
<td>bytes</td>
</tr>
<tr>
<td>cache hit percentage</td>
<td>85.04</td>
<td>percent</td>
</tr>
</tbody>
</table>

16 rows selected.

In this example, the cache hit percentage (PGA) is more than 85 percent, which is good enough for an OLTP or data warehouse application. In fact, if you have a large data-warehousing type of database, you may even have to be content with a much smaller PGA cache hit ratio.

Another way to look at PGA efficiency is by using the following query, which involves the V$SQL_WORKAREA_HISTOGRAM view. The view contains information about the number of work areas executed with optimal, one-pass, and multipass memory size. The work areas are divided into groups, whose optimal requirement varies from 0KB to 1KB, 1KB to 2KB, 2KB to 4KB—and so on.

Listing 20-7 shows the results of a query using the V$SQL_WORKAREA_HISTOGRAM view.

Listing 20-7. Using the V$SQL_WORKAREA_HISTOGRAM View

```sql
SELECT 2 low_optimal_size/1024 "Low (K)", 3 (high_optimal_size + 1)/1024 "High (K)", 4 optimal_executions "Optimal", 5 onepass_executions "1-Pass", 6 multipasses_executions ">1 Pass" 7 FROM v$sql_workarea_histogram 8* WHERE total_executions <> 0;
```
<table>
<thead>
<tr>
<th>Low (K)</th>
<th>High (K)</th>
<th>Optimal</th>
<th>1-Pass</th>
<th>&gt;1 Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>7820241</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>9011</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>4064</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>512</td>
<td>3782</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1024</td>
<td>18479</td>
<td>58</td>
<td>4</td>
</tr>
<tr>
<td>1024</td>
<td>2048</td>
<td>3818</td>
<td>53</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>4096</td>
<td>79</td>
<td>241</td>
<td>67</td>
</tr>
<tr>
<td>4096</td>
<td>8192</td>
<td>1</td>
<td>457</td>
<td>26</td>
</tr>
<tr>
<td>8192</td>
<td>16384</td>
<td>0</td>
<td>11</td>
<td>44</td>
</tr>
<tr>
<td>16384</td>
<td>32768</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>32768</td>
<td>65536</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>65536</td>
<td>131072</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>131072</td>
<td>262144</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

14 rows selected.

SQL>

An overwhelming number of the sorts in this instance were done optimally, with only a few sorts using the one-pass approach. This is why you have the 85 percent PGA hit ratio in the previous example. Here’s an instance that’s in trouble, as shown by the significant number of sorts in the one-pass and the multipass (> 1 Pass) group. Right now, most of your customers will be complaining that the database is slow.

Note that the query is the same as in the previous example. Here’s the output:

<table>
<thead>
<tr>
<th>Low (K)</th>
<th>High (K)</th>
<th>Optimal</th>
<th>1-Pass</th>
<th>&gt;1 Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>2</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>129866</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>1288</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>2</td>
<td>180</td>
<td>61</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>6</td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>256</td>
<td>512</td>
<td>44</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>512</td>
<td>1024</td>
<td>1063</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>1024</td>
<td>2048</td>
<td>31069</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2048</td>
<td>4096</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>8192</td>
<td>16384</td>
<td>986</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>32768</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

As you can see, there are significant multiple pass sorts in this example, and you can bet that the cache hit ratio is going to be low, somewhere in the 70 percent range. Fortunately, all you have to do to speed up the instance is to increase the value of the `PGA_AGGREGATE_TARGET` parameter in the following manner:

```sql
ALTER SYSTEM SET pga_aggregate_target=500000000;
```

The new `V$PROCESS_MEMORY` view lets you view dynamic PGA memory usage for each Oracle process, and shows the PGA usage by each process for categories such as Java, PL/SQL, OLAP, and SQL. Here’s a simple query on that view:
You can also use the V$PROCESS view to monitor PGA usage by individual processes. If you’re running out of memory on your server, it’s a good idea to see whether you can release some PGA memory for other uses. Here’s a query that shows you the allocated, used, and freeable PGA memory for each process currently connected to the instance:

```sql
SELECT program, pga_used_mem, pga_alloc_mem, pga_freeable_mem, pga_max_mem V$PROCESS;
```

You can use the following SQL statement to estimate quickly the proportion of work areas since you started the Oracle instance, using optimal, one-pass, and multipass PGA memory sizes:

```sql
SELECT name PROFILE, cnt COUNT, 
      DECODE(total, 0, 0, ROUND(cnt*100/total)) PERCENTAGE 
FROM (SELECT name, value cnt, (sum(value) over ()) total 
      FROM V$SYSSTAT 
      WHERE name like 'workarea exec%');
```

In the preceding example, the PGA cache hit percentage for optimal executions is 100 percent, which, of course, is excellent. Oracle DBAs have traditionally paid a whole lot more attention to tuning the SGA memory component because the PGA memory tuning in its present format is relatively new. DBAs in charge of applications requiring heavy-duty hashing and sorting requirements are well advised to pay close attention to the performance of the PGA. It’s easy to tune the PGA, and the results of a well-tuned PGA show up in dramatic improvements in performance.

## Evaluating System Performance

The instance-tuning efforts that you undertake from within Oracle will have only a limited impact (they may even have a negative impact) if you don’t pay attention to the system performance as a whole. System performance includes the CPU performance, memory usage, and disk I/O. In the following sections you’ll look at each of these important resources in more detail.
CPU Performance

You can use operating system utilities such as System Activity Reporter (sar) or vmstat to find out how the CPU is performing. Don’t panic if your processors seem busy during peak periods—that’s what they’re there for, so you can use them when necessary. If the processors are showing a heavy load during low usage times, you do need to investigate further. Listing 20-8 shows a sar command output indicating how hard your system is using the CPU resources right now.

Listing 20-8. sar Command Output Showing CPU Usage

$ sar -u 10 5
HP-UX finance1 B.11.00 A 9000/800 07/03/05
13:39:17 %usr %sys %wio %idle
13:39:27 34 23 7 36
13:39:37 37 17 8 38
13:39:47 34 18 6 41
13:39:57 31 16 9 44
13:40:07 38 19 11 32
Average 35 19 8 38

In the preceding listing, the four columns report on the following CPU usage patterns:

- %usr shows the proportion of total CPU time taken up by the various users of the system.
- %sys shows the proportion of time the system itself was using the CPU.
- %wio indicates the percentage of time the system was waiting for I/O.
- %idle is the proportion of time the CPU was idle.

If the %wio or %idle percentages are near zero during nonpeak times, it indicates a CPU-bound system.

Remember that an intensive CPU usage level may mean that an operating-system process is hogging CPU, or an Oracle process may be doing the damage. If it is Oracle, a background process such as PMON may be the culprit, or an Oracle user process may be running some extraordinarily bad ad hoc SQL query on the production box. You may sometimes track down such a user and inform the person that you’re killing the process in the interest of the welfare of the entire system. Imagine your surprise when you find that the user’s Oracle process is hale and hearty, while merrily continuing to devastate your system in the middle of a busy day. This could happen because a child process or a bequeath process continued to run even after you killed this user. It pays to double-check that the user is gone—lock, stock, and barrel—instead of assuming that the job has been done.

That said, let’s look at some of the common events that could cause CPU-related slowdowns on your system.

The Run Queue Length

One of the main indicators of a heavily loaded CPU system is the length of the run queue. A longer run queue means that more processes are lined up, waiting for CPU processing time. Occasional blips in the run-queue length aren’t bad, but prolonged high run-queue lengths indicate that the system is CPU bound.
CHAPTER 20 ■ PERFORMANCE TUNING: TUNING THE INSTANCE

CPU Units Used by Processes

You can determine the number of CPU units a UNIX process is using by using the simple process (ps) command, as shown here:

```bash
$ ps -ef | grep f60
   UID   PID   PPID  C   STIME  TTY  TIME   CMD
oracle 20108  4768   0 09:11:49  ?   0:28 f60webm
oracle 883  4768   5 17:12:21  ?   0:06 f60webm
oracle 7090  4768  16 09:18:46  ?   1:08 f60webm
oracle 15292  4768 101 15:49:21  ?   1:53 f60webm
oracle 18654  4768 101 15:49:21  ?   1:53 f60webm
oracle 24316  4768 101 15:49:21  ?   1:53 f60webm
```

The key column to watch is the fourth one from the left, which indicates the CPU units of processing that each process is using. If each CPU on a server has 100 units, the Oracle process with PID 15292 (the fourth in the preceding list) is occupying more than an entire CPU’s processing power. If you have only two processors altogether, you should worry about this process and why it’s so CPU intensive.

Finding High CPU Users

If the CPU usage levels are high, you need to find out which of your users are among the top CPU consumers. Listing 20-9 shows how you can easily identify those users.

Listing 20-9. Identifying High CPU Users

```sql
SQL> SELECT n.username, 
2   s.sid, 
3   s.value 
4  FROM v$sesstat s,v$statname t, v$session n 
5  WHERE s.statistic# = t.statistic# 
6  AND n.sid = s.sid 
7  AND t.name='CPU used by this session' 
8  ORDER BY s.value desc;
```

<table>
<thead>
<tr>
<th>USERNAME</th>
<th>SID</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHLMAN</td>
<td>152</td>
<td>20745</td>
</tr>
<tr>
<td>NROBERTS</td>
<td>103</td>
<td>4944</td>
</tr>
<tr>
<td>JOHLMAN</td>
<td>167</td>
<td>4330</td>
</tr>
<tr>
<td>LROLLINS</td>
<td>87</td>
<td>3699</td>
</tr>
<tr>
<td>JENGMAN</td>
<td>130</td>
<td>3694</td>
</tr>
<tr>
<td>JPATEL</td>
<td>66</td>
<td>3344</td>
</tr>
<tr>
<td>NALAPATI</td>
<td>73</td>
<td>3286</td>
</tr>
</tbody>
</table>

Listing 20-9 shows that CPU usage isn’t uniformly spread across the users. You need to investigate why one user is using such a significant quantity of resources. If you need to, you can control CPU usage by a single user or a group of users by using the Database Resource Manager. You can also find out session-level CPU usage information by using the V$SESSTAT view, as shown in Listing 20-10.
Listing 20-10. Determining Session-Level CPU Usage

SQL> SELECT sid, s.value "Total CPU Used by this Session"
2  FROM V$SESSTAT S
3  WHERE S.statistic# = 12
4* ORDER BY S.value DESC;

<table>
<thead>
<tr>
<th>SID</th>
<th>Total CPU Used by this Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>496</td>
<td>27623</td>
</tr>
<tr>
<td>542</td>
<td>21325</td>
</tr>
<tr>
<td>111</td>
<td>20814</td>
</tr>
<tr>
<td>731</td>
<td>17089</td>
</tr>
<tr>
<td>424</td>
<td>15228</td>
</tr>
</tbody>
</table>

SQL>

What Is the CPU Time Used For?

It would be a mistake to treat all CPU time as equal. CPU time is generally understood as the processor time taken to perform various tasks, such as the following:

- Loading SQL statements into the library cache
- Searching the shared pool for parsed versions of SQL statements
- Parsing the SQL statements
- Querying the data dictionary
- Reading data from the buffer cache
- Traversing index trees to fetch index keys

The total CPU time used by an instance (or a session) can be viewed as the sum of the following components:

total CPU time = parsing CPU usage + recursive CPU usage + other CPU usage

Ideally, your total CPU usage numbers should show a small proportion of the first two categories of CPU usage—parsing and recursive CPU usage. For example, for a session-wide estimate of CPU usage, you can run the query shown in Listing 20-11.

Listing 20-11. Decomposition of Total CPU Usage

SQL> SELECT name,value FROM V$SYSSTAT
2  WHERE name IN ('CPU used by this session',
3  'recursive cpu usage',
4  'parse time cpu');

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive cpu usage</td>
<td>4713085</td>
</tr>
<tr>
<td>CPU used by this session</td>
<td>98196187</td>
</tr>
<tr>
<td>parse time cpu</td>
<td>132947</td>
</tr>
</tbody>
</table>

3 rows selected.
SQL>
In this example, the sum of recursive CPU usage and parse time CPU usage is a small proportion of total CPU usage. You need to be concerned if the parsing or recursive CPU usage is a significant part of total CPU usage. Let’s see how you can go about reducing the CPU usage attributable to these various components.

Note In the following examples, you can examine CPU usage at the instance level by using the V$SYSSTAT view or at an individual session level by using the V$SESSTAT view. Just remember that the column “total CPU used by this session” in the V$SYSSTAT view refers to the sum of the CPU used by all the sessions combined.

Parse Time CPU Usage

As you learned at the beginning of this chapter, parsing is an expensive operation that you should reduce to a minimum. In the following example, the parse time CPU usage is quite low as a percentage of total CPU usage. The first query tells you that the total CPU usage in your instance is 49159124:

```
SQL> SELECT name, value FROM V$SYSSTAT WHERE name LIKE '%CPU%';
NAME                                 VALUE
---------------------------         --------
CPU used when call started          13220745
CPU used by this session            49159124
2 rows selected.
SQL>
```

The next query shows that the parse time CPU usage is 96431, which is an insignificant proportion of total CPU usage in your database:

```
SQL> SELECT name, value FROM V$SYSSTAT WHERE name LIKE '%parse%';
NAME                            VALUE
--------------------          --------
parse time cpu                  96431
parse time elapsed             295451
parse count (total)           3147900
parse count (hard)              29139
4 rows selected.
SQL>
```

Listing 20-12 shows an example of a session whose CPU usage is predominantly due to high parse time.

Listing 20-12. Determining Parse Time CPU Usage

```
SQL> SELECT a.value "Tot_CPU_Used_This_Session", b.value "Total_Parse_Count", c.value "Hard_Parse_Count", d.value "Parse_Time_CPU" FROM v$sysstat a, v$sysstat b, v$sysstat c, v$sysstat d
```
Parse time CPU in the preceding example is fully two-thirds of the total CPU usage. Obviously, you need to be concerned about the high rates of parsing, even though most of the parses are soft parses. The next section shows you what you can do to reduce the amount of parsing in your database.

Reducing Parse Time CPU Usage

If parse time CPU is the major part of total CPU usage, you need to reduce this by performing the following steps:

1. Use bind variables and remove hard-coded literal values from code, as explained in the “Optimizing the Library Cache” section earlier in this chapter.

2. Make sure you aren’t allocating too much memory for the shared pool. Remember that even if you have an exact copy of a new SQL statement in your library cache, Oracle has to find it by scanning all the statements in the cache. If you have a zillion relatively useless statements sitting in the cache, all they’re doing is slowing down the instance by increasing the parse time.

3. Make sure you don’t have latch contention on the library cache, which could result in increased parse time CPU usage.

4. If your TKPROF output or one of the queries shown previously indicates that total parse time CPU is as high as 90 percent or more, check to make sure all the tables in the queries have been analyzed recently. If you don’t have statistics on some of the tables, the parsing process generates the statistics, but the parse CPU usage time goes up dramatically.

Recursive CPU Usage

Recursive CPU usage is mostly for data dictionary lookups and for executing PL/SQL programs. Thus, if your application uses a high number of packages and procedures, you’ll see a significant amount of recursive CPU usage.

In the following example, there’s no need for alarm, because the percentage of recursive CPU usage is only about 5 percent of total CPU usage:

```
SQL> SELECT name, value FROM V$SYSSTAT
   WHERE name IN ('CPU used by this session', 'recursive cpu usage');
NAME                        VALUE
-------------------------------
recursive cpu usage          4286925
CPU used by this session    84219625
2 rows selected.
SQL>
```
If the recursive CPU usage percentage is a large proportion of total CPU usage, you may want to make sure the shared pool memory allocation is adequate. However, a PL/SQL-based application will always have a significant amount of recursive CPU usage.

**Note** A high number of recursive SQL statements may also indicate that Oracle is busy with space management activities, such as allocating extents. This has a detrimental effect on performance. You can avoid this problem by increasing the extent sizes for your database objects. This is another good reason to choose locally managed tablespaces, which cut down on the number of recursive SQL statements.

### Memory

Operating system physical memory holds all the data and programs by loading them from disk. System CPU executes programs only if they’re loaded into the physical memory. If excessive memory usage occurs, the operating system will use virtual memory, which is storage space on secondary storage media such as disks, to hold temporarily some of the data and/or programs being used. The space for the virtual memory is called swap space. When the system needs room in the physical or main memory, it “swaps out” some programs to the swap area, thus freeing up additional physical memory for an executing program.

The operating system swaps out data in units called pages, which are the smallest units of memory that can be used in transferring memory back and forth between physical memory and the swap area. When the operating system needs a page that has been swapped out to the swap area, a page fault is said to occur. Page faults are commonly referred to as simply “paging,” and involve the transfer of data from virtual memory back to the physical memory. An excessive amount of paging results in degradation of operating system performance, and thus affects Oracle instance performance as well.

One of the best ways to check operating system memory performance is by using the `vmstat` utility, which was explained in Chapter 3.

### Disk I/O

The way you configure your disk system has a profound impact on your I/O rates. You have to address several issues when you’re planning your disk system. Important factors that have a bearing on your I/O are as follows:

- **Choice of RAID configuration**: Chapter 3 covered RAID system configuration in detail. Just remember that a RAID 5 configuration doesn’t give you ideal I/O performance if your application involves a large number of writes. For faster performance, make sure you use a configuration that involves striping your disks, preferably according to the Oracle guidelines.

- **Raw devices or operating system file systems**: Under some circumstances, you can benefit by using raw devices, which bypass the operating system buffer cache. Raw devices have their own drawbacks, though, including limited backup features, and you want to be sure the benefits outweigh the drawbacks. Raw devices in general provide faster I/O capabilities and give better performance for a write-intensive application. You might also want to consider alternative file systems such as VERITAS’s VXFSS, which helps large I/O operations through its direct I/O option.

- **I/O size**: I/O size is in terms of the Oracle block size. The minimum size of I/O depends on your block size, and the maximum size depends on the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter. If your application is OLTP based, the I/O size needs to be small, and if your application is oriented toward a DSS, the I/O size needs to be much larger. As of Oracle Database 10.2, the database automatically tunes this parameter, if you don’t set it.
• **Logical volume stripe sizes:** Stripe size (or stripe width) is a function of the stripe depth and the number of drives in the striped set. If you stripe across multiple disks, your database’s I/O performance will be higher and its load balancing will be enhanced. Make sure that the stripe size is larger than the average I/O request; otherwise, you’ll be making multiple I/Os for a single I/O request by Oracle. If you have multiple concurrent I/O requests, your stripe size should be much larger than the I/O size. Most modern LVMs can dynamically reconfigure the stripe size.

• **Number of controllers and disks:** The number of spindles and the number of controllers are both important variables in determining disk performance. Even if you have a large number of spindles, you could conceivably run into contention at the controller level.

• **Distribution of I/O:** Your goal should be to avoid a lopsided distribution of I/O in your disk system. If you’re using an LVM or using striping at the hardware level, you don’t have a whole lot to worry about in this area. However, if you aren’t using an LVM or using striping at the hardware level, you should manually arrange your datafiles on the disks such that the I/O rate is fairly even across the system. Note that your tables and indexes are usually required to be in different tablespaces, but there is no rule that they have to be placed on different disks. Because the index is read before the table, they can coexist on the same disk.

### Measuring I/O Performance

You have a choice of several excellent tools to measure I/O performance. Several operating system utilities are easy to use and give you information about how busy your disks are. Iostat and sar are two of the popular operating system utilities that measure disk performance. I explained how to use both these tools in Chapter 3.

### Is the I/O Optimally Distributed?

From the sar output, you can figure out whether you’re using the storage subsystem heavily. If the number of waits is higher than the number of CPUs, or if the service times are high (say, greater than 20 milliseconds), then your system is facing contention at the I/O level. One of the most useful pieces of information you can get is by using the `sar -d` command to find out if you’re using any of your disks excessively compared to other disks in the system. Once you identify such hot spots, you can move the datafiles to less busy drives, thereby spreading the load more evenly.

The following is the output of a `sar -d` command that shows extremely high queue values. Even at peak levels, the `avque` column value should be less than 2. Here, it is 61.4. Obviously, something is happening on the file system named `c2t6d0` that’s showing up as a high queue value:

```bash
$ sar -d 10 5
HP-UX finance1 B.11.00 A 9000/8000 07/03/08
17:27:13 device %busy avque r+w/s blks/s avwait avserv
17:27:23 c2t6d0 100 61.40 37 245 4.71 10.43
c5t6d0 20.38 0.50 28 208 4.92 9.54
c2t6d0 100 61.40 38 273 4.55 9.49
c5t6d0 18.28 0.50 27 233 4.46 7.93
c0t1d0 0.10 0.50 4 33 4.99 0.81
```

You can obtain an idea about the I/O distribution in your system by using the query in Listing 20-13.
Listing 20-13. Determining I/O Distribution in the Database

```
SQL> SELECT d.name,
       2  f.phyrds reads,
       3  f.phywrts wrts,
       4  (f.readtim / decode(f.phyrds,0,-1,f.phyrds)) readtime,
       5  (f.writetim / decode(f.phywrts,0,-1,phywrts)) writetime
       6  FROM
       7  v$datafile d,
       8  v$filestat f
       9  WHERE
      10  d.file# = f.file#
      11  ORDER BY
      12* d.name;

NAME                          READS   WRTS   READTIME     WRITETIME
----------------------------- -----   ----  ----------    ----------
/pa01/oradata/pa/lol_i_17.dbf    23      9  .608695652    .222222222
/pa01/oradata/pa/lol_i_18.dbf    18      7  .277777778             0

. . .
```

Caution  Excessive reads and writes on some disks indicate that there might be disk contention in your I/O system.

Reducing Disk Contention

If there’s severe I/O contention in your system, you can undertake some of the following steps, depending on your present database configuration:

- Increase the number of disks in the storage system.
- Separate the database and the redo log files.
- For a large table, use partitions to reduce I/O.
- Stripe the data either manually or by using a RAID disk-striping system.
- Invest in cutting-edge technology, such as file caching, to avoid I/O bottlenecks.
- Consider using the new Automatic Storage Management system, which is discussed in Chapter 17.

The Oracle SAME Guidelines for Optimal Disk Usage

Oracle provides you with the Stripe and Mirror Everything (SAME) guidelines for optimal disk usage. This methodology advocates striping all files across all disks and mirroring all data to achieve a simple, efficient, and highly available disk configuration. Striping across all the available disks aims to spread the load evenly and avoid hot spots. The SAME methodology also recommends placing frequently accessed data on the outer half of the disks. The goal of the SAME disk storage strategy is to eliminate I/O hot spots and maximize I/O bandwidth.

Network Performance

You may want to rule out the network as the culprit during a poor performance period by checking whether it’s overloaded and exhibiting excessive latency. You can use the operating system tool
netstat to check your network performance, as I explained in Chapter 3. Excessive network round-trips necessitated by client messages could clog your network and increase the latency, thus indirectly affecting the CPU load on your system. In cases such as this, you must try and reduce the network round-trips by using array inserts and array fetches.

**Measuring Instance Performance**

One of the trickiest parts of the DBA’s job is to judge the performance of the Oracle instance accurately. Trainers and the manuals advise you to perform diligent proactive tuning, but in reality most tuning efforts are reactive—they’re intensive attempts to fix problems that perceptibly slow down a database and cause user complaints to increase. You look at the same things whether you’re doing proactive or reactive tuning, but proactive tuning gives you the luxury of making decisions in an unhurried and low-stress environment. Ideally, you should spend more than two-thirds of your total tuning time on proactive planning. As you do so, you’ll find that you’re reacting less and less over time to sudden emergencies.

Oracle Database 11g uses the concept of DB time (discussed in detail in Chapter 17) to determine how well the instance is performing. You can look at some statistics to see how well the database is performing. These statistics fall into two groups: database hit ratios and database wait statistics. If you’re consistently seeing numbers in the high 90s for the various hit ratios you saw earlier in this chapter, you’re usually doing well, according to this approach.

However, the big question is this: Do high hit ratios automatically imply a perfectly tuned and efficient database? The surprising answer is no. To understand this confusing fact, you need to look at what hit ratios indicate. The following sections examine the two main groups of performance statistics.

**Database Hit Ratios**

Database hit ratios are the most commonly used measures of performance. These include the buffer cache hit ratio, the library cache and dictionary cache hit ratios, the latch hit ratio, and the disk sort ratios. These hit ratios don’t indicate how well your system is performing. They’re broad indicators of proper SGA allocation, and they may be high even when the system as a whole is performing poorly. The thing to remember is that the hit ratios only measure such things as how physical reads compare with logical reads, and how much of the time a parsed version of a statement is found in memory. As to whether the statements themselves are efficient or not, the hit ratios can’t tell you anything. When your system is slow due to bottlenecks, the hit ratios are of little help, and you should turn to a careful study of wait statistics instead.

---

**Caution** Even if you have a 99.99 percent buffer cache hit ratio, you may still have major inefficiencies in your application. What if you have an extremely high number of “unnecessary” logical reads? This makes your buffer cache hit ratio look good, as that hit ratio is defined as physical reads over the sum of logical reads. Although you may think your application should run faster because you’re doing most of your reads from memory instead of disk, this may well not happen. The reason is that even if you’re doing logical reads, you’re still burning up the CPU units to do the unnecessary logical reads. In essence, by focusing zealously on the buffer cache hit ratio to relieve the I/O subsystem, you could be an unwitting party to a CPU usage problem. Please read Cary Millsap’s interesting article, “Why You Should Focus on LIOs Instead of PIOs” ([http://www.hotsos.com/e-library/abstract.php?id=7](http://www.hotsos.com/e-library/abstract.php?id=7)), which explains why a high logical I/O level could be a major problem.

---

When faced with a slow-performing database or a demand for shorter response times, Oracle DBAs have traditionally looked to increase their database hit ratios and tune the database by adjusting
a host of initialization parameters (such as SGA allocations). More recently, there’s been awareness that the key area to focus on is clearing up database bottlenecks that contribute to a higher response time.

The total response time for a query is the time Oracle takes to execute it, plus the time the process spends waiting for resources such as latches, data buffers, and so on. For a database instance to perform well, ideally your application should spend little time waiting for access to critical resources.

Let’s now turn to examining the critical wait events in your database, which can be real show-stoppers on a busy day in a production instance.

### Database Wait Statistics

When your users complain that the database is crawling and they can’t get their queries returned fast enough, there’s no use in your protesting that your database is showing high hit ratios for the shared pool and the buffer cache (and the large pool and redo log buffer as well). If the users are waiting for long periods of time to complete their tasks, then the response time will be slow, and you can’t say that the database is performing well, the high hit ratios notwithstanding.

---

**Note** For an interesting review of the Oracle wait analysis (the wait interface), please read one of the early papers in this area, titled “Yet Another Performance Profiling Method (or YAPP-Method),” by Anjo Kolk, Shari Yamaguchi, and Jim Viscusi. It’s available at the OraPerf web site at [http://www.oraperf.com](http://www.oraperf.com) (a free registration is required).

---

Once it starts executing a SQL statement, an Oracle process doesn’t always get to work on the execution of the statement without any interruptions. Often, the process has to pause or wait for some resource to be released before it can continue its execution. Thus, an active Oracle process is doing one of the following at any given time:

- The process is executing the SQL statement.
- The process is waiting for something (for example, a resource such as a database buffer or a latch). It could be waiting for an action such as a write to the buffer cache to complete.

That’s why the response time—the total time taken by Oracle to finish work—is correctly defined as follows:

\[
\text{response time} = \text{service time} + \text{wait time}
\]

When you track the total time taken by a transaction to complete, you may find that only part of that time was taken up by the Oracle server to actually “do” something. The rest of the time, the server may have been waiting for some resource to be freed up or waiting for a request to do something. This busy resource may be a slow log writer or a database writer process. The wait event may also be due to unavailable buffers or latches. The wait events in the V$SYSTEM_EVENT view (instance-level waits) and the V$SESSION_EVENT view (session-level waits) tell you what the wait time is due to (full table scans, high number of library cache latches, and so on). Not only do the wait events tell you what the wait time in the database instance is due to, but they also tell you a lot about bottlenecks in the network and the application.

---

**Note** It’s important to understand that the wait events are only the symptoms of problems, most likely within the application code. The wait events show you what’s slowing down performance, but not why a certain wait event is showing up in large numbers. It’s up to you to investigate the SQL code to find out the real cause of the performance problems.
Four dynamic performance views contain wait information: V$SESSION, V$SYSTEM_EVENT, V$SESSION_EVENT, and V$SESSION_WAIT. These four views list just about all the events the instance was waiting for and the duration of these waits. Understanding these wait events is essential for resolving performance issues.

Let’s look at the common wait events in detail in the following sections. Remember that the four views show similar information but focus on different aspects of the database, as you can see from the following summary. The wait events are most useful when you have timed statistics turned on. Otherwise, the wait events only have the number of times they occurred, not the length of time they consumed. Without timing the events, you can’t tell if a wait event was indeed a contributing factor in a system slowdown.

Tip Use the wait event views (wait interface) for examining current and recent performance issues in your instance. For comprehensive analysis of most performance problems, you need to use the ADDM, which analyzes the AWR hourly snapshots.

Oracle wait interface analysis has garnered quite a bit of attention in the last few years. There are entire books dedicated to Oracle waits. I discuss the important performance topic of Oracle wait analysis later in this chapter, in the section “Analyzing Instance Performance.” Ideally, all sessions should be on the CPU, with zero time spent waiting for resources such as I/O. However, remember that every working instance will have some kind of wait. It’s unrealistic to expect to work toward a zero wait system. The key question should not be whether you have any Oracle wait events occurring, but rather if there are excessive waits.

Wait Events and Wait Classes

Any time a server process waits for an event to complete, it’s classified as a wait event. There are more than 950 Oracle wait events in Oracle Database 11g. The most common wait events are those caused by resource contention such as latch contention, buffer contention, and I/O contention.

A wait class is a grouping of related wait events, and every wait event belongs to a wait class. Important wait classes include Administrative, Application, Concurrency, Configuration, Idle, Network, System I/O, and User I/O. For example, the Administrative wait class includes lock waits caused by row-level locking. The User I/O class of waits refers to waits for blocks to be read off a disk. Using wait classes helps you move quickly to the root cause of a problem in your database by limiting the focus of further analysis. Here’s a summary of the main wait classes in Oracle Database 11g:

- **Administrative**: Waits caused by administrative commands, such as rebuilding an index, for example.
- **Application**: Waits due to the application code.
- **Cluster**: Waits related to Real Application Cluster management.
- **Commit**: The single wait event log file sync, which is a wait caused by commits in the database.
- **Concurrency**: Waits for database resources that are used for locking, for example, latches.
- **Idle**: Waits that occur when a session isn’t active, for example, the 'SQL*Net message from client' wait event.
- **Network**: Waits incurred during network messaging.
- **Other**: Miscellaneous waits.
- **Scheduler**: Resource Manager–related waits.
• **System I/O**: Waits for background-process I/O, including the database writer background process’s wait for the `db file parallel write` event. Also included are archive-log–related waits and redo log read-and-write waits.

• **User I/O**: Waits for user I/O. Includes the `db file sequential read` and `db file scattered read` events.

### Analyzing Instance Performance

One of the first things you can do to measure instance performance efficiency is to determine the proportion of total time the database is spending working compared to the proportion of time it’s merely waiting for resources. The `V$SYSMETRIC` view displays the system metric values for the most current time interval. The following query using the `V$SYSMETRIC` view reveals a database instance where waits are taking more time than the instance CPU usage time:

```sql
SQL> SELECT METRIC_NAME, VALUE
       FROM V$SYSMETRIC
       WHERE METRIC_NAME IN ('Database CPU Time Ratio',
                               'Database Wait Time Ratio') AND
       INTSIZE_CSEC = (SELECT MAX(INTSIZE_CSEC) FROM V$SYSMETRIC);

METRIC_NAME                       VALUE
------------------------         -----
Database Wait Time Ratio          72
Database CPU Time Ratio           28

SQL>
```

Once you realize that the total instance wait time ratio is much higher than the CPU time ratio, you can explore things further. Wait classes provide a quick way to figure out why the database instance is performing poorly. In the example shown in Listing 20-14, you can easily see that user I/O waits are responsible for most of the wait time. You can establish this fact by looking at the `PCT_TIME` column, which gives you the percentage of time attributable to each wait class. Total waits are often misleading, as you can see by looking at the `NETWORK` wait class. In percentage terms, network waits are only 1 percent, although total network waits constitute more than 51 percent of total waits in this instance.

### Listing 20-14. Determining Total Waits and Percentage Waits by Wait Class

```sql
SQL> SELECT WAIT_CLASS,
       TOTAL_WAITS,
       ROUND(100 * (TOT_WAITS / SUM_WAITS),2) PCT_TOTWAITS,
       ROUND((TIME_WAITED / 100),2) TOT_TIME_WAITED,
       ROUND(100 * (TOT_TIME_WAITED / SUM_TIME),2) PCT_TIME
       FROM (SELECT WAIT_CLASS,
               TOT_WAITS,
               TOT_TIME_WAITED
       FROM V$SYSTEM_WAIT_CLASS
       WHERE WAIT_CLASS != 'Idle'),
       (SELECT SUM(TOT_WAITS) SUM_WAITS,
               SUM(TOT_TIME_WAITED) SUM_TIME
       FROM V$SYSTEM_WAIT_CLASS
       WHERE WAIT_CLASS != 'Idle')
       ORDER BY PCT_TIME DESC;
```
### Using V$ Tables for Wait Information

The key dynamic performance tables for finding wait information are the V$SYSTEM_EVENT, V$SESSION_EVENT, V$SESSION_WAIT, and the V$SESSION views. The first two views show the waiting time for different events.

The V$SYSTEM_EVENT view shows the total time waited for all the events for the entire system since the instance started up. The view doesn’t focus on the individual sessions experiencing waits, and therefore it gives you a high-level view of waits in the system. You can use this view to find out what the top instance-wide wait events are. You can calculate the top \( n \) waits in the system by dividing the event’s wait time by the total wait time for all events.

The three key columns of the V$SYSTEM_EVENT view are total_waits, which gives the total number of waits; time_waited, which is the total wait time per session since the instance started; and average_wait, which is the average wait time by all sessions per event.

The V$SESSION_EVENT view is similar to the V$SYSTEM_EVENT view, and it shows the total time waited per session. All the wait events for an individual session are recorded in this view for the duration of that session. By querying this view, you can find out the specific bottlenecks encountered by each session.

The third dynamic view is the V$SESSION_WAIT view, which shows the current waits or just-completed waits for sessions. The information on waits in this view changes continuously based on the types of waits that are occurring in the system. The real-time information in this view provides you with tremendous insight into what’s holding up things in the database right now. The V$SESSION_WAIT view provides detailed information on the wait event, including details such as file number, latch numbers, and block number. This detailed level of information provided by the V$SESSION_WAIT view enables you to probe into the exact bottleneck that’s slowing down the database. The low-level information helps you zoom in on the root cause of performance problems.

The following columns from the V$SESSION_WAIT view are important for troubleshooting performance issues:

- **EVENT**: These are the different wait events described in the next section (for example, latch free and buffer busy waits).
- **P1, P2, P3**: These are the additional parameters that represent different items, depending on the particular wait event. For example, if the wait event is **db file sequential read**, \( P1 \) stands for the file number, \( P2 \) stands for the block number, and \( P3 \) stands for the number of blocks. If the wait is due to a latch free event, \( P1 \) stands for the latch address, \( P2 \) stands for the latch number, and \( P3 \) stands for the number of attempts for the event.
- **WAIT_CLASS_ID**: This identifies the wait class.

<table>
<thead>
<tr>
<th>WAIT_CLASS</th>
<th>TOTAL_WAITS</th>
<th>PCT_TOT_WAITS</th>
<th>TOT_TIME_WAITED</th>
<th>PCT_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>User I/O</td>
<td>664935191</td>
<td>45.07</td>
<td>46305770.5</td>
<td>84.42</td>
</tr>
<tr>
<td>Other</td>
<td>394490128</td>
<td>2.67</td>
<td>5375324.17</td>
<td>9.8</td>
</tr>
<tr>
<td>Concurrency</td>
<td>78768788</td>
<td>.53</td>
<td>1626254.9</td>
<td>2.96</td>
</tr>
<tr>
<td>Network</td>
<td>754692506</td>
<td>51.15</td>
<td>547128.66</td>
<td>1</td>
</tr>
<tr>
<td>Application</td>
<td>2012092</td>
<td>.01</td>
<td>449945.5</td>
<td>.82</td>
</tr>
<tr>
<td>Commit</td>
<td>15526036</td>
<td>.11</td>
<td>351043.3</td>
<td>.64</td>
</tr>
<tr>
<td>Configuration</td>
<td>12898465</td>
<td>.09</td>
<td>116029.85</td>
<td>.21</td>
</tr>
<tr>
<td>System I/O</td>
<td>53005529</td>
<td>.36</td>
<td>78783.64</td>
<td>.14</td>
</tr>
<tr>
<td>Administrative</td>
<td>25</td>
<td>0</td>
<td>7.6</td>
<td>0</td>
</tr>
<tr>
<td>Scheduler</td>
<td>1925</td>
<td>0</td>
<td>.15</td>
<td>0</td>
</tr>
</tbody>
</table>

10 rows selected.

SQL>
• **WAIT_CLASS#:** This is the number of the wait class.
• **WAIT_CLASS:** This is the name of the wait class.
• **WAIT_TIME:** This is the wait time in seconds if the state is waited known time.
• **SECONDS_IN_WAIT:** This is the wait time in seconds if the state is waiting.
• **STATE:** The state could be waited short time, waited known time, or waiting, if the session is waiting for an event.

The fourth wait-related view is the V$SESSION view. Not only does this view provide many details about the session, it also provides significant wait information as well. The V$SESSION view contains all the columns of the V$SESSION_WAIT view, plus a number of other important session-related columns. Because of this overlap of wait information in the V$SESSION and the V$SESSION_WAIT views, you can use the V$SESSION view directly to look for most of the wait-related information, without recourse to the V$SESSION_WAIT view. You can start analyzing the wait events in your system by first querying the V$SYSTEM_EVENT view to see if any significant wait events are occurring in the database. You can do this by running the query shown in Listing 20-15.

**Listing 20-15. Using the V$SYSTEM_EVENT View to View Wait Events**

```sql
SQL> SELECT event, time_waited, average_wait
2  FROM V$SYSTEM_EVENT
3  GROUP BY event, time_waited, average_wait
4* ORDER BY time_waited DESC;
```

<table>
<thead>
<tr>
<th>EVENT</th>
<th>TIME_WAITED</th>
<th>AVERAGE_WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbms ipc message</td>
<td>24483121</td>
<td>216.71465</td>
</tr>
<tr>
<td>SQL*Net message from client</td>
<td>18622096</td>
<td>106.19049</td>
</tr>
<tr>
<td>PX Idle Wait</td>
<td>12485418</td>
<td>205.01844</td>
</tr>
<tr>
<td>pmon timer</td>
<td>3120909</td>
<td>306.93440</td>
</tr>
<tr>
<td>smon timer</td>
<td>3093214</td>
<td>29459.18100</td>
</tr>
<tr>
<td>PL/SQL lock timer</td>
<td>3024203</td>
<td>1536.68852</td>
</tr>
<tr>
<td>db file sequential read</td>
<td>831831</td>
<td>.25480</td>
</tr>
<tr>
<td>db file scattered read</td>
<td>107253</td>
<td>.90554</td>
</tr>
<tr>
<td>free buffer waits</td>
<td>52955</td>
<td>43.08787</td>
</tr>
<tr>
<td>log file parallel write</td>
<td>19958</td>
<td>2.02639</td>
</tr>
<tr>
<td>latch free</td>
<td>5884</td>
<td>1.47505</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58 rows selected.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example shows a simple system with hardly any waits other than the idle type of events and the SQL*Net wait events. There aren’t any significant I/O-related or latch-contention–related wait events in this database. The `db file sequential read` (caused by index reads) and the `db file scattered read` (caused by full table scans) wait events do seem somewhat substantial, but if you compare the total wait time contributed by these two events to the total wait time since the instance started, they don’t stand out. Furthermore, the `AVERAGE_WAIT` column shows that both these waits have a low average wait time (caused by index reads). I discuss both these events, along with several other Oracle wait events, later in this chapter, in the section “Important Oracle Wait Events.” However, if your query on a real-life production system shows significant numbers for any nonidle wait event, it’s probably a good idea to find out the SQL statements that are causing the waits. That’s where you have to focus your efforts to reduce the waits. You have different ways to obtain the associated SQL for the waits, as explained in the following section.
Obtaining Wait Information

Obtaining wait information is as easy as querying the related dynamic performance tables. For example, if you wish to find out quickly the types of waits different user sessions (session-level wait information) are facing and the SQL text of the statements they’re executing, you can use the following query:

```
SQL> SELECT s.username,
       2  t.sql_text, s.event
       3  FROM V$SESSION s, V$SQLTEXT t
       4  WHERE s.sql_hash_value = t.hash_value
       5  AND s.sql_address  = t.address
       6  AND s.type    <> 'BACKGROUND'
       7* ORDER BY s.sid,t.hash_value,t.piece;
```

**Note** You need to turn on statistics collection by either setting the initialization parameter TIMED_STATISTICS to TRUE or setting the initialization parameter STATISTICS_LEVEL to TYPICAL or ALL.

If you want a quick instance-wide wait event status, showing which events were the biggest contributors to total wait time, you can use the query shown in Listing 20-16 (several idle events are listed in the output, but I don’t show them here).

**Listing 20-16. Instance-Wide Waits Sorted by Total Wait Time**

```
SQL> SELECT event, total_waits,time_waited
       2  FROM V$SYSTEM_EVENT
       3  WHERE event NOT IN
       4  ('pmon timer','smon timer','rdbms ipc reply','parallel dequeue
       5  wait','virtual circuit','%SQL*Net%','client message','NULL    event')
       6* ORDER BY time_waited DESC;
```

```
EVENT                              TOTAL_WAITS     TIME_WAITED
------------------------           -----------     ------------
db file sequential read               35051309        15965640
latch free                             1373973         1913357
db file scattered read                 2958367         1840810
enqueue                                   2837          370871
buffer busy waits                       444743          252664
log file parallel write                 146221          123435
SQL>
```

The preceding query shows that waits due to the `db file scattered read` wait event account for most of the waits in this instance. The `db file sequential read` wait event, as you’ll learn shortly, is caused by full table scans. It’s somewhat confusing in the beginning when you’re trying to use all the wait-related V$ views, which all look similar. Here’s a quick summary of how you go about using the key wait-related Oracle Database 11g dynamic performance views.

First, look at the V$SYSTEM_EVENT view and rank the top wait events by the total amount of time waited, as well as the average wait time for that event. Start investigating the top waits in terms of the percentage of total wait time. You can also look at any AWR reports you may have, because the AWR also lists the top five wait events in the instance.

Next, find out more details about the specific wait event that’s at the top of the list. For example, if the top event is buffer busy waits, look in the V$WAITSTAT view to see which type of buffer block (data block, undo block, and so on) is causing the buffer busy waits (a simple `SELECT *` from V$WAITSTAT...}
gets you all the necessary information). For example, if the undo-block buffer waits make up most of
your buffer busy waits, then the undo segments are at fault, not the data blocks.

Finally, use the V$SESSION view to find out the exact objects that may be the source of a problem.
For example, if you have a high amount of \texttt{db file scattered read}-type waits, the V$SESSION view
will give you the file number and block number involved in the wait events. In the following example,
the V$SESSION view is used to find out who is doing the full table scans showing up as the most
important wait events right now. As explained earlier, the \texttt{db file scattered read} wait event is
caused by full table scans.

\begin{verbatim}
SQL> SELECT sid, sql_address, sql_hash_value 
   FROM V$SESSION WHERE event = 'db file scattered read';

Here’s an example that shows how to find out the current wait event for a given session:

SQL> SELECT sid, state, event, wait_time, seconds_in_wait 
   FROM v$session 
   WHERE sid=1418;

<table>
<thead>
<tr>
<th>SID</th>
<th>STATE</th>
<th>EVENT</th>
<th>WAIT_TIME</th>
<th>SECONDS_IN_WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1418</td>
<td>WAITING</td>
<td>db file sequential read</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The value of 0 under the \texttt{WAIT_TIME} column indicates that the wait event \texttt{db file sequential
read} is occurring for this session. When the wait event is over, you'll see values for the \texttt{WAIT_TIME}
and the \texttt{SECONDS_IN_WAIT} columns.

You can also use the V$SQLAREA view to find out which SQL statements are responsible for high
disk reads. If latch waits predominate, you should be looking at the V$LATCH view to gain more
information about the type of latch that's responsible for the high latch wait time:

\begin{verbatim}
SQL> SELECT sid, blocking_session, username, 
      event, seconds_in_wait siw 
   FROM V$SESSION 
   WHERE blocking_session_status = 'VALID';

<table>
<thead>
<tr>
<th>SID</th>
<th>BLOCKING_SESS</th>
<th>USERNAME</th>
<th>EVENT</th>
<th>SIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1218</td>
<td>1527</td>
<td>UCR_USER</td>
<td>enq: TX - row lock contention</td>
<td>23</td>
</tr>
<tr>
<td>1400</td>
<td>1400</td>
<td>APPOWNER</td>
<td>latch free</td>
<td>0</td>
</tr>
</tbody>
</table>

The V$SESSION_WAIT_HISTORY View

The V$SESSION_WAIT_HISTORY view holds information about the \texttt{last ten wait events} for each active
session. The other wait-related views, such as the V$SESSION and the V$SESSION_WAIT, show you
only the wait information for the most recent wait. This may be a short wait, thus escaping your scrutiny.
Here’s a sample query using the V$SESSION_WAIT_HISTORY view:

\begin{verbatim}
SQL> SELECT seq#, event, wait_time, p1, p2, p3 
   FROM V$SESSION_WAIT_HISTORY 
   WHERE sid = 988 
   ORDER BY seq#;
\end{verbatim}
### Analyzing Waits with Active Session History

The V$SESSION_WAIT view tells you what resource a session is waiting for. However, neither of these views provides you with historical information about the waits in your instance. Once the task is over, you can no longer view the wait information using the V$SESSION_WAIT view. The waits are so fleeting that by the time you query the views, the wait in most times is over. The new Active Session History feature, by recording session information, enables you to go back in time and review the history of a performance bottleneck in your database. Although the AWR provides hourly snapshots of the instance by default, you won’t be able to analyze events that occurred five or ten minutes ago, based on AWR data. This is where the ASH information comes in handy. ASH samples the V$SESSION view every second and collects the wait information for all active sessions. An active session is defined as a session that’s on the CPU or waiting for a resource. You can view the ASH session statistics through the view V$ACTIVE_SESSION_HISTORY, which contains a single row for each active session in your instance.

ASH is a rolling buffer in memory, with older information being overwritten by new session data. Every 60 minutes, the MMON background process flushes filtered ASH data to disk, as part of the hourly AWR snapshots. Once the ASH data is flushed to disk, you won’t be able to see it in the V$ACTIVE_SESSION_HISTORY view. You’ll now have to use the DBA_HIST_ACTIVE_SESS_HISTORY view to look at the historical data.

In the following sections, I show how you can query the V$ACTIVE_SESSION_HISTORY view to analyze current (recent) Active Session History.

### Using the V$ACTIVE_SESSION_HISTORY View

The V$ACTIVE_SESSION_HISTORY view provides a window on the ASH data held in memory by the Oracle instance before it’s flushed as part of the hourly AWR snapshots. You can use it to get information on things such as the SQL that’s consuming the most resources in the database, the particular objects causing the most waits, and the identities of the users who are waiting the most.

In the following sections I show how to use the ASH information to gain valuable insights into the nature of the waits in your instance, including answering such questions as the objects with the highest waits, the important wait events in your instance, and the users waiting the most.

---

<table>
<thead>
<tr>
<th>SEQ#</th>
<th>EVENT</th>
<th>WAIT_TIME</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>db file sequential read</td>
<td>0</td>
<td>52</td>
<td>21944</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>db file sequential read</td>
<td>0</td>
<td>50</td>
<td>19262</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>latch: shared pool</td>
<td>0.13835E+19</td>
<td>198</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>db file sequential read</td>
<td>0</td>
<td>205</td>
<td>21605</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>db file sequential read</td>
<td>4</td>
<td>52</td>
<td>13924</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>db file sequential read</td>
<td>1</td>
<td>49</td>
<td>29222</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>db file sequential read</td>
<td>2</td>
<td>52</td>
<td>14591</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>db file sequential read</td>
<td>2</td>
<td>52</td>
<td>12723</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>db file sequential read</td>
<td>0</td>
<td>205</td>
<td>11883</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>db file sequential read</td>
<td>0</td>
<td>205</td>
<td>21604</td>
<td></td>
</tr>
</tbody>
</table>

10 rows selected.

SQL>
**Objects with the Highest Waits**

The following query identifies the objects causing the most waits and the type of events the objects waited for during the last 15 minutes:

```sql
SQL> SELECT o.object_name, o.object_type, a.event,
  2  SUM(a.wait_time +
  3   a.time_waited) total_wait_time
  4  FROM v$active_session_history a,
  5  dba_objects o
  6  WHERE a.sample_time between sysdate - 30/2880 and sysdate
  7  AND a.current_obj# = o.object_id
  8  GROUP BY o.object_name, o.object_type, a.event
  9* ORDER BY total_wait_time;
```

<table>
<thead>
<tr>
<th>OBJECT_NAME</th>
<th>OBJECT_TYPE</th>
<th>EVENT</th>
<th>TOTAL_WAIT_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC_ADDRESS</td>
<td>TABLE</td>
<td>SQL*Net message to client</td>
<td>2</td>
</tr>
<tr>
<td>PERS_PHONES</td>
<td>TABLE</td>
<td>db file sequential read</td>
<td>8836</td>
</tr>
<tr>
<td>PAY_FK_I</td>
<td>INDEX</td>
<td>db file sequential read</td>
<td>9587</td>
</tr>
<tr>
<td>UC_STAGING</td>
<td>TABLE</td>
<td>log file sync</td>
<td>23633</td>
</tr>
<tr>
<td>PERSONNEL</td>
<td>TABLE</td>
<td>db file sequential read</td>
<td>43612</td>
</tr>
</tbody>
</table>

**Most Important Wait Events**

The following query lists the most important wait events in your database in the last 15 minutes:

```sql
SQL> SELECT a.event,
  2  SUM(a.wait_time +
  3   a.time_waited) total_wait_time
  4  FROM v$active_session_history a
  5  WHERE a.sample_time between
  6  sysdate - 30/2880 and sysdate
  7  GROUP BY a.event
  8* ORDER BY total_wait_time DESC;
```

<table>
<thead>
<tr>
<th>EVENT</th>
<th>TOTAL_WAIT_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait for SGA component shrink</td>
<td>878774247</td>
</tr>
<tr>
<td>smon timer</td>
<td>300006992</td>
</tr>
<tr>
<td>PL/SQL lock timer</td>
<td>210117722</td>
</tr>
<tr>
<td>SQL*Net message from client</td>
<td>21588571</td>
</tr>
<tr>
<td>db file scattered read</td>
<td>1062608</td>
</tr>
<tr>
<td>db file sequential read</td>
<td>105271</td>
</tr>
<tr>
<td>log file sync</td>
<td>13019</td>
</tr>
<tr>
<td>latch free</td>
<td>274</td>
</tr>
<tr>
<td>SQL*Net more data to client</td>
<td>35</td>
</tr>
<tr>
<td>null event</td>
<td>6</td>
</tr>
</tbody>
</table>

17 rows selected.

**Users with the Most Waits**

The following query lists the users with the highest wait times within the last 15 minutes:

```sql
SQL> SELECT a.event,
  2  SUM(a.wait_time +
  3   a.time_waited) total_wait_time
  4  FROM v$active_session_history a
  5  WHERE a.sample_time between
  6  sysdate - 30/2880 and sysdate
  7  GROUP BY a.event
  8* ORDER BY total_wait_time DESC;
```

**Users with the Most Waits**

The following query lists the users with the highest wait times within the last 15 minutes:
SQL> SELECT s.sid, s.username,
       2  SUM(a.wait_time +
       3  a.time_waited) total_wait_time
       4  FROM v$active_session_history a,
       5  v$session s
       6  WHERE a.sample_time between sysdate - 30/2880 and sysdate
       7  AND a.session_id=s.sid
       8  GROUP BY s.sid, s.username
       9* ORDER BY total_wait_time DESC;

SID   USERNAME        TOTAL_WAIT_TIME
----   ---------       ---------------
1696   SYSOWNER              165104515
885    SYSOWNER               21575902
1087   BLONDI                  5019123
1318   UCRSL                    569723
1334   REBLOOM                  376354
1489   FRAME                       395
15 rows selected.

Identifying SQL with the Highest Waits

Using the following query, you can identify the SQL that’s waiting the most in your instance. The sample time covers the last 15 minutes.

SQL> SELECT a.user_id,d.username,s.sql_text,
       2  SUM(a.wait_time + a.time_waited) total_wait_time
       3  FROM v$active_session_history a,
       4  v$sqlarea s,
       5  dba_users d
       6  WHERE a.sample_time between sysdate - 30/2880 and sysdate
       7  AND a.sql_id = s.sql_id
       8  AND a.user_id = d.user_id
       9* GROUP BY a.user_id,s.sql_text, d.username;

USER_ID    USERNAME SQL_TEXT                   TOTAL_WAIT_TIME
----------  -------- ----------------------------  ----------------
0           SYS       BEGIN dbms_stats . . .; END;           9024233
...

Wait Classes and the Wait-Related Views

The V$SESSION_WAIT view shows the events and resources that active sessions are waiting for. Using the V$SESSION_WAIT view, you can also see what types of wait classes your session waits belong to. Here’s an example:

SQL> SELECT wait_class, event, sid, state, wait_time, seconds_in_wait
       FROM v$session_wait
       ORDER BY wait_class, event, sid;
The previous query indicates that the most important wait lies within the Application wait class.

The V$SYSTEM_WAIT_CLASS view gives you a breakdown of waits by wait classes, as shown here:

```
SQL> SELECT wait_class, time_waited
  2  FROM v$system_wait_class
  3 ORDER BY time_waited DESC;

  WAIT_CLASS          TIME_WAITED
  ------------------- -----------
      Idle            1.0770E+11
  User I/O                4728148400
        Other               548221433
  Concurrency             167154949
     Network                56271499
     Application           46336445
       Commit              35742104
   Configuration           11667683
     System I/O           8045920
     Administrative        760
         Scheduler           16
11 rows selected.
```

The V$SESSION_WAIT_CLASS view shows the total time spent in each type of wait class by an individual session. Here’s an example:

```
SQL> SELECT wait_class, time_waited
  2  FROM v$session_wait_class
  3  WHERE sid = 1053
  4* ORDER BY time_waited DESC;

  WAIT_CLASS    TIME_WAITED
  ------------- -----------
        Idle      21190
     User I/O     8487
         Other        70
  Concurrency     13
     Application     0
         Network     0
6 rows selected.
```

The V$SESSION_WAIT_CLASS view shows the total time spent in each type of wait class by an individual session. Here’s an example:

```
SQL> SELECT wait_class, time_waited
  2  FROM v$session_wait_class
  3  WHERE sid = 1053
  4* ORDER BY time_waited DESC;

  WAIT_CLASS    TIME_WAITED
  ------------- -----------
        Idle      21190
     User I/O     8487
         Other        70
  Concurrency     13
     Application     0
         Network     0
6 rows selected.
```
The V$WAITCLASSMETRIC view shows metric values of wait classes for the most recent 60-second interval. The view keeps information for up to one hour. Here's an example of using the query:

```
SQL> SELECT WAIT_CLASS#, WAIT_CLASS_ID
2   dbtime_in_wait,time_waited,wait_count
3  FROM v$waitclassmetric
4* ORDER BY time_waited DESC;
```

<table>
<thead>
<tr>
<th>WAIT_CLASS#</th>
<th>DBTIME_IN_WAIT</th>
<th>TIME_WAITED</th>
<th>WAIT_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2723168908</td>
<td>170497</td>
<td>51249</td>
</tr>
<tr>
<td>0</td>
<td>1893977003</td>
<td>5832</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>1740759767</td>
<td>717</td>
<td>1351</td>
</tr>
<tr>
<td>5</td>
<td>3386400367</td>
<td>11</td>
<td>68</td>
</tr>
<tr>
<td>7</td>
<td>2000153315</td>
<td>8</td>
<td>52906</td>
</tr>
<tr>
<td>9</td>
<td>4108307767</td>
<td>6</td>
<td>99</td>
</tr>
<tr>
<td>1</td>
<td>4271401580</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3290255840</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4166625743</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>3871361733</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2396326324</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3875070507</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

12 rows selected.

SQL>

As you can see, WAIT_CLASS 6 tops the list, meaning that idle class waits currently account for most of the wait time in this instance.

**Looking at Segment-Level Statistics**

Whether you use the AWR or the wait-related V$ views, you're going to find no information about where a certain wait event is occurring. For example, you can see from the V$SYSTEM_EVENT view that buffer busy waits are your problem, and you know that you reduce these waits by switching from manual segment space management to Automatic Segment Space Management (ASSM). However, neither AWR nor the V$ view indicates which tables or indexes you should be looking at to fix the high wait events. Oracle provides three V$ views to help you drill down to the segment level.

The segment-level dynamic performance views are V$SEGSTAT_NAME, V$SEGSTAT, and V$SEGMENT_STATISTICS. Using these, you can find out which of your tables and indexes are being subjected to high resource usage or high waits. Once you're aware of a performance problem due to high waits, you can use these segment-level views to find out exactly which table or index is the culprit and fix that object to reduce the waits and increase database performance. The V$SEGMENT_NAME view provides you with a list of all the segment levels that are being collected, and tells you whether the statistics are sampled or not.

Let's see how you can use these segment-level views to your advantage when you're confronted with a high number of wait events in your system. Say you look at the V$SYSTEM_EVENT view and realize that there are a large number of buffer busy waits. You should now examine the V$SEGMENT_STATISTICS view with a query such as the following to find out which object is the source of the high buffer busy waits. You can then decide on the appropriate corrective measures for this wait event, as discussed in the section “Important Oracle Wait Events” later in this chapter.
Collecting Detailed Wait Event Information

Selecting data from V$ dynamic performance views and interpreting them meaningfully isn’t always so easy to do. Because the views are dynamic, the information that they contain is constantly changing as Oracle updates the underlying tables for each wait event. Also, the wait-related dynamic performance views you just examined don’t provide crucial data such as bind variable information. For a more detailed level of wait information, you can use one of the methods described in the following sections.

Method 1: Using the Oracle Event 10046 to Trace SQL Code

You can get all kinds of bind variable information by using a special trace called the 10046 trace, which is much more advanced than the SQL Trace utility you saw in Chapter 19. The use of this trace causes an output file to be written to the trace directory. You can set the 10046 trace in many ways by specifying various levels, and each higher level provides you with more detailed information. (Level 12 is used in the following case as an example only—it may give you much more information than necessary. Level 4 gives you detailed bind value information, and Level 8 gives you wait information.)

You can use the ALTER SESSION statement as follows:

```
SQL> SELECT owner, object_name, object_type, tablespace_name
  2  FROM V$SEGMENT_STATISTICS
  3  WHERE statistic_name='buffer busy waits'
  4* ORDER BY value DESC;
```

<table>
<thead>
<tr>
<th>OWNER</th>
<th>OBJECT_NAME</th>
<th>OBJECT_TYPE</th>
<th>TABLESPACE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSOWNER</td>
<td>LAB_DATA</td>
<td>TABLE</td>
<td>LAB_DATA_D</td>
</tr>
<tr>
<td>SYSOWNER</td>
<td>LAB_ADDR_I</td>
<td>INDEX</td>
<td>LAB_DATAS_I</td>
</tr>
<tr>
<td>SYSOWNER</td>
<td>PERS_SUMMARIES</td>
<td>TABLE</td>
<td>PERS_SUMMARIES_D</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQL>

You can also incorporate the following line in your init.ora file:

```
event = 10046 trace name context forever, level 12
```

Method 2: Using the Oradebug Utility to Perform the Trace

You can use the oradebug utility as shown in the following example:

```
SQL> ORADEBUG SETMYPID
Statement processed.
SQL> ORADEBUG EVENT 10046 TRACE NAME CONTEXT FOREVER LEVEL 8;
Statement processed.
```

In this example, SETMYPID indicates that you want to trace the current session. If you want a different session to be traced, you replace this with SETOSPID <Process Id>.
Method 3: Using the DBMS_SYSTEM Package to Set the Trace

Use the SET_EV procedure of the DBMS_SYSTEM package so you can set tracing on in any session, as shown in the following example:

```sql
SQL> EXECUTE SYS.DBMS_SYSTEM.SET_EV (9,271,10046,12,'');
PL/SQL procedure successfully completed.
SQL>
```

Method 4: Using the DBMS_MONITOR Package

The DBMS_MONITOR package provides you with an easy way to collect extended session trace information. You enable tracing of a user's session using the DBMS_MONITOR.SESSION_TRACE_ENABLE package. Here's the structure of the procedure:

```sql
DBMS_MONITOR.SESSION_TRACE_ENABLE(
    session_id   IN  BINARY_INTEGER DEFAULT NULL,
    serial_num   IN  BINARY_INTEGER DEFAULT NULL,
    waits        IN  BOOLEAN DEFAULT TRUE,
    binds        IN  BOOLEAN DEFAULT FALSE)
```

If you set the WAITS parameter to TRUE, the trace will contain wait information. Similarly, setting the BINDS parameter to TRUE will provide bind information for the session being traced.

If you don't set the SESSION_ID parameter or set it to NULL, your own session will be traced. Here's how you trace your session using the DBMS_MONITOR package:

```sql
SQL> EXECUTE dbms_monitor.session_trace_enable (waits=>TRUE, binds=>TRUE);
```

In addition to all the preceding methods of gathering wait information, you have the handy OEM Database Control tool, which lets you drill down to various items from the Database Control home page.

---

**Note** Both the AWR report that you can obtain by using the awrrpt.sql script and the ADDM report that you can obtain with the addmrpt.sql script contain copious amounts of wait information.

---

**Important Oracle Wait Events**

The wait events listed in the sections that follow have a significant impact on system performance by increasing response times. Each of these events (and several other events) indicates an unproductive use of time because of an excessive demand for a resource, or contention for Oracle structures such as tables or the online redo log files.

---

**Note** The query `SELECT NAME FROM V$EVENT_NAME` gives you the complete list of all Oracle wait events.

---

**Buffer Busy Waits**

The buffer busy waits event occurs in the buffer cache area when several processes are trying to access the same buffer. One session is waiting for another session's read of a buffer into the buffer cache. This wait could also occur when the buffer is in the buffer cache, but another session is changing it.
Note Starting with the Oracle Database 10.2 release, the buffer busy wait has been divided into several events: you can have very few buffer busy waits, but a huge number of read by other session waits, which were previously reported as buffer busy waits.

You should observe the V$SESSION_WAIT view while this wait is occurring to find out exactly what type of block is causing the wait.

Two of the common causes of high buffer busy waits are contention on data blocks belonging to tables and indexes, and contention on segment header blocks. If you’re using dictionary managed tablespaces or locally managed tablespaces with manual segment space management (see Chapter 7), you should proceed as follows:

- If the waits are primarily on data blocks, try increasing the PCTFREE parameter to lower the number of rows in each data block. You may also want to increase the INITRANS parameter to reduce contention from competing transactions.
- If the waits are mainly in segment headers, increase the number of freelists or freelist groups for the segment in question, or consider increasing the extent size for the table or index.

The best way to reduce buffer busy waits due to segment header contention is to use locally managed tablespaces with ASSM. ASSM also addresses contention for data blocks in tables and indexes.

Besides the segment header and data block contention, you could also have contention for rollback segment headers and rollback segment blocks. However, if you’re using Automatic Undo Management (AUM), you don’t have to do anything other than make sure you have enough space in your undo management tablespace to address the rollback (undo) headers and blocks, leaving table and index data blocks and segment headers as the main problem areas. The following query clearly shows that in this database, the buffer busy waits are in the data blocks:

```sql
SQL> SELECT class, count FROM V$WAITSTAT
  2  WHERE COUNT > 0
  3* ORDER BY COUNT DESC;

CLASS                   COUNT
---------------    ----------
data block             519731
undo block               5829
undo header              2026
segment header             25
SQL>
```

If data-block buffer waits are a significant problem even with ASSM, this could be caused by poorly chosen indexes that lead to large index range scans. You may try using global hash-partitioned indexes, and you can also tune SQL statements as necessary to fix these waits. Oracle seems to indicate that if you use AUM instead of traditional rollback segments, then two types of buffer busy waits, undo block and undo header, will go away. However, that’s not the case in practice, as the following example from a database with AUM shows:

```sql
CLASS                   COUNT
---------------     ---------
undo header             29891
data block                 52
segment header            1
SQL>
```
Occasionally, you may have a situation where the buffer busy waits spike suddenly, seemingly for no reason. The sar utility (use the sar –d option) might indicate high request queues and service times. This often happens when the disk controllers get saturated by a high amount of I/O. Usually, you see excessive core dumps during this time, and if core dumps are choking your I/O subsystem, do the following:

- Move your core dump directory to a less busy file system, where it resides by itself.
- Use the following init.ora or SPFILE parameters to control core dumps in your system. Setting these parameters’ values could reduce the size of a core dump to a few megabytes from a gigabyte or more:
  
  \[
  \begin{align*}
  \text{SHADOW\_CORE\_DUMP} & = \text{PARTIAL} /* \text{or} \text{NONE} */ \\
  \text{BACKGROUND\_CORE\_DUMP} & = \text{PARTIAL} /* \text{or} \text{NONE} */
  \end{align*}
  \]
- Investigate the core dumps and see whether you can fix them by applying necessary Oracle and operating-system patch sets.

**Checkpoint Completed**

The **CHECKPOINT COMPLETED** wait event means that a session is waiting for a checkpoint to complete. This could happen when you’re shutting the database down or during normal checkpoints.

**Db File Scattered Read**

The **db file scattered read** wait event indicates that full table scans (or index fast full scans) are occurring in the database. The initialization parameter `DB_FILE_MULTIBLOCK_READ_COUNT` sets the number of blocks read at one time by Oracle. The database will automatically tune this parameter if you don’t set any value for it in your parameter file. Although Oracle reads data in multiblock chunks, it scatters the data into noncontiguous cache buffers. If you don’t have many full table scans and if they mainly consist of smaller tables, don’t worry about it.

However, if this event is showing up as an important wait event, you need to look at it as an I/O-related problem—the database isn’t able to cope with an excessive request for physical I/Os. There are two possible solutions. You can either reduce the demand for physical I/Os or increase the capacity of the system to handle more I/Os. You can reduce the demand for physical I/O by drilling down further to see whether one of the following solutions will work. Raising the buffer cache component of the SGA would normally contribute to lowering physical I/Os. However, I’m assuming that you’re using Automatic Shared Memory Management by setting the `SGA\_TARGET` initialization parameter, in which case your buffer cache is already optimally set by the database:

- Add missing indexes on key tables (unlikely in a production system).
- Optimize SQL statements if they aren’t following an efficient execution plan.

If you don’t see any potential for reducing the demand for physical I/O, you’re left with no choice but to increase the number of disks on your system. You also need to make sure you’re reducing the hot spots in your system by carefully distributing the heavily hit tables and indexes across the available disks. You can identify the datafiles where the full table or index fast full scans are occurring with help of a query using the V$FILESTAT view. In this view, two columns are of great use:

- **phyrds**: The number of physical reads done
- **phyblkrd**: The number of physical blocks read

Obviously, the number of phyrds is equal to or close to the number of phyblkhrsds because almost all reads are single block reads. If the column phyrds shows a much smaller value than the phyblkhrsds
column, Oracle is reading multiple blocks in one read—a full table scan or an index fast full scan, for example. Here’s a sample query on the V$FILESTAT view:

```
SQL> SELECT file#, phyrd, phyblkrd
2  FROM V$FILESTAT
3* WHERE phyrd != phyblkrd;
```

```
FILE#     PHYRDS   PHYBLKRD
----------     ------   --------
1       4458      36533
7      67923     494433
15      28794     378676
16      53849     408981
SQL>
```

**Db File Sequential Read**

The db file sequential read wait event signifies that a single block is being read into the buffer cache. This event occurs when you’re doing an indexed read and you’re waiting for a physical I/O call to return. This is nothing to be alarmed about, because the database has to wait for file I/O. However, you should investigate disk I/O if this statistic seems extraordinarily high. If disk sorts are high, you can make them lower by increasing the value of the PGA_AGGREGATE_TARGET initialization parameter. Because the very occurrence of this event proves that your application is making heavy use of an index, you can’t do much to reduce the demand for physical I/Os in this case, unlike in the case of the db file scattered read event. Increasing the number of disks and striping indexes across them may be your best bet to reduce db file sequential read waits. If the objects aren’t too large, you can use the DEFAULT and KEEP buffer pools to retain them in memory. However, if the objects are large, you may not have this option. Indexed reads are going to show up in most systems as a wait, and it’s not necessarily a bad thing, because indexes are required in most cases for faster data retrieval.

**Direct Path Read and Direct Path Write**

The direct path read and direct path write events are waits that occur while performing a direct read or write into the PGA, bypassing the SGA buffer cache. Direct path reads indicate that sorts are being done on disk instead of in memory. They could also result from a busy I/O system. If you use automatic PGA tuning, you shouldn’t encounter this problem too often.

Automatic tuning of the PGA by Oracle should reduce your disk sorts due to a low PGA memory allocation. Another solution may be to increase the number of disks, as this problem also results in an I/O system that can’t keep up with the increased requests for reading blocks into the PGA. Of course, tuning the SQL statements themselves to reduce sorting wouldn’t hurt in this case.

**Free Buffer Waits**

Free buffer waits usually show up when the database writer process is slow. The database writer process is simply unable to keep up with the requests to service the buffer cache. The number of dirty buffers in cache waiting to be written to disk is larger than the number of buffers the database writer process can write per batch. Meanwhile, sessions have to wait because they can’t get free buffers to write to. First, you need to rule out whether the buffer cache is too small, and check the I/O numbers on the server, especially the write time, using an operating system tool. A check of the database buffer cache and a quick peek at the Database Control’s Memory Advisor will show you the pattern of usage of the various memory components and if you’re below the optimal buffer cache level, in which case you can increase the size of the buffer cache. Of course, if you’re using Automatic Shared Memory Management, the database will size the SGA allocations for you.
The other reason for a high number of free buffer waits in your system is that the number of database writer processes is inadequate to perform the amount of work your instance needs to get done. As you know, you can add additional database writer processes to the default number of processes, which is one database writer process for every eight processors on your host machine. If your database performs heavy data modifications and you determine that the database writer is responsible for wait events, you can reduce these waits in most cases by increasing the number of database writer processes. You can choose a value between 2 and 20 for the `DB_WRITER_PROCESSES` initialization parameter. Oracle recommends that you use one database writer process for every four CPUs on your system. You can’t change this variable on the fly, so you’ll need to perform a system restart to change the number of database writer processes.

**Enqueue Waits**

Enqueues are similar to locks in that they are internal mechanisms that control access to resources. High enqueue waits indicate that a large number of sessions are waiting for locks held by other sessions. You can query the dynamic performance view `V$ENQUEUE_STAT` to find out which of the enqueues have the most wait times reported. You can do this by using the `cum_wait_time` (shows the cumulative time spent waiting for the enqueue) column of the view.

Note that the use of locally managed tablespaces eliminates several types of enqueues such as space transactions (ST) enqueues. In a system with a massive concurrent user base, most common enqueues are due to infrequent commits (or rollbacks) by transactions that force other transactions to wait for the locks held by the early transactions. In addition, there may be a problem with too few interested transactions list (ITL) slots, which also show up as transaction (TX) enqueues. Locally managed tablespaces let you avoid the most common types of space-related enqueues.

**Latch Free**

*Latches* are internal serialization mechanisms used to protect shared data structures in Oracle’s SGA. You can consider a latch as a type of lock that’s held for an extremely short time period. Oracle has several types of latches, with each type guarding access to a specific set of data. The latch free wait event is incremented when a process can’t get a latch on the first attempt. If a required Oracle latch isn’t available, the process requesting it keeps spinning and retrying to gain the access. This spinning increases both the wait time and the CPU usage in the system. Oracle uses about 500 latches, but two of the important latches that show up in wait statistics are the shared pool latch (and the library cache latches) and the cache buffers LRU chain. It’s normal to see a high number of latch free events in an instance. You should worry about this wait event only if the total time consumed by this event is high.

High latch waits will show up in your AWR reports, or you can use the query shown in Listing 20-17 to find out your latch hit ratio.

**Listing 20-17. Determining the Latch Hit Ratio**

```
SQL> SELECT a.name "Latch Name",
      a.gets "Gets (Wait)",
      a.misses "Misses (Wait)",
      (1 - (misses / gets)) * 100 "Latch Hit Ratio %"
FROM   V$LATCH a
WHERE a.gets   != 0
UNION
SELECT a.name "Latch Name",
      a.gets "Gets (Wait)",
      a.misses "Misses (Wait)",
      100 "Latch Hit Ratio"
```
If the ratio isn’t close to 1, it’s time to think about tuning the latch contention in your instance. There’s only one shared pool latch for the database, and it protects the allocation of memory in the library cache. The library cache latch regulates access to the objects present in the library cache. Any SQL statement, PL/SQL code, procedure, function, or package needs to acquire this latch before execution. If the shared pool and library cache latches are high, more often than not that’s because the parse rates in the database are high. The high parse rates are due to the following factors:

- An undersized shared pool
- Failure to use bind variables
- Using dissimilar SQL statements and failing to reuse statements
- Users frequently logging off and logging back into the application
- Failure to keep cursors open after each execution
- Using a shared pool size that’s too large

The cache buffers LRU chain latch free wait is caused by high buffer cache throughput, either due to full table scans or the use of unselective indexes, which lead to large index range scans. Unselective indexes can also lead to yet another type of latch free wait: the cache buffer chain latch free wait. These wait events are often due to the presence of hot blocks, so you need to investigate why that might be happening. If you see a high value for row cache object latch waits, it indicates contention for the dictionary cache, and you need to increase the shared pool memory allocation.

In most instances, latch waits tend to show up as a wait event, and DBAs sometimes are alarmed by their very presence in the wait event list. As with the other Oracle wait events, ask yourself this question: “Are these latch waits a significant proportion of my total wait time?” If the answer is no, don’t worry about it—your goal isn’t to try and eliminate all waits in the instance, because you can’t do it.

**Log Buffer Space**

The log buffer space wait event indicates that a process waited for space in the log buffer. Either the log buffer is too small or the redo is being written faster than the log writer process can write it to the redo log buffer. If the redo log buffer is already large, then investigate the I/O to the disk that houses the redo log files. There’s probably some contention for the disk, and you need to work on reducing the I/O contention. This type of wait usually shows up when the log buffer is too small, in which case you increase the log buffer size. A large log buffer tends to reduce the redo log I/O in general. Note that Oracle’s default value for this parameter is several megabytes in size. If you have a large number of huge transactions, you might want to bump up the value of the LOG_BUFFER initialization parameter from its default value, although too high a value means that too much data may have to be written to the redo log files at one time.

**Log File Switch**

The log file switch wait event can occur when a session is forced to wait for a log file switch because the log file hasn’t yet been archived. It can also occur because the log file switch is awaiting the completion of a checkpoint.

If the problem isn’t due to the archive destination getting full, it means that the archive process isn’t able to keep up with the rate at which the redo logs are being archived. In this case, you need to increase the number of archiver (ARCn) processes to keep up with the archiving work. The default
for the ARCn process is 2. This is a static parameter, so you can't use this fix to resolve a slowdown right away.

You also need to investigate whether too-small redo log files are contributing to the wait for the log file switch. If the log file switch is held up pending the completion of a checkpoint, obviously the log files are too small and hence are filling up too fast. You need to increase the size of the redo log files in this case. Redo log files are added and dropped online, so you can consider this a dynamic change.

If you see high values for redo log space requests in V$SYSSTAT, that means that user processes are waiting for space in the redo log buffer. This is because the log writer process can't find a free redo log file to empty the contents of the log buffer. Resize your redo logs, with the goal of having a log switch every 15 to 30 minutes.

**Log File Sync**

You'll see a high number of waits under the log file sync category if the server processes are frequently waiting for the log writer process to finish writing committed transactions (redo) to the redo log files from the log buffer. This is usually the result of too-frequent commits, and you can reduce it by adopting batch commits instead of a commit after every single transaction. This wait event may also be the result of an I/O bottleneck.

**Idle Events**

You can group some wait events under the category idle events. Some of these may be harmless in the sense that they simply indicate that an Oracle process was waiting for something to do. These events don't indicate database bottlenecks or contention for Oracle's resources. For example, the system may be waiting for a client process to provide SQL statements for execution. The following list presents some common idle events:

- **Rdbms ipc message**: This is used by the background process, such as the log writer process and PMON, to indicate they are idle.
- **SMON timer**: The SMON process waits on this event.
- **PMON timer**: This indicates the PMON process idle event.
- **SQL*Net message from client**: This is the user process idle event.

You should ignore many idle events during your instance performance tuning. However, some events, such as the SQL*Net message from client event, may indicate that your application isn't using an efficient database connection strategy. In this case, you need to see how you can reduce these waits, maybe by avoiding frequent logging on and off by applications.

**Examining System Performance**

You can use the various operating system tools, such as vmstat, to examine system performance. You can also use the new V$OSSTAT dynamic view to figure out the performance characteristics of your system. The V$OSSTAT view provides operating system statistics in the form of busy ticks.

Here are some of the key system usage statistics:

- **NUM_CPUS**: Number of processors
- **IDLE_TICKS**: Number of hundredths of a second that all processors have been idle
- **BUSY_TICKS**: Number of hundredths of a second that all processors have been busy executing code
- **USER_TICKS**: Number of hundredths of a second that all processors have been busy executing user code
• SYS_TICKS: Number of hundredths of a second that all processors have been busy executing kernel code

• IOWAIT_TICKS: Number of hundredths of a second that all processors have been waiting for I/O to complete

The AVG_IDLE_WAITS, AVG_BUSY_TICKS, AVG_USER_TICKS, AVG_SYS_TICKS, and AVG_IOWAIT_TICKS columns provide the corresponding information average over all the processors. Here’s a simple example that shows how to view the system usage statistics captured in the V$OSSTAT view:

SQL> SELECT * FROM V$OSSTAT;

<table>
<thead>
<tr>
<th>STAT_NAME</th>
<th>VALUE</th>
<th>OSSTAT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_CPUS</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>IDLE_TICKS</td>
<td>17812</td>
<td>1</td>
</tr>
<tr>
<td>BUSY_TICKS</td>
<td>2686882247</td>
<td>2</td>
</tr>
<tr>
<td>USER_TICKS</td>
<td>1936724603</td>
<td>3</td>
</tr>
<tr>
<td>SYS_TICKS</td>
<td>750157644</td>
<td>4</td>
</tr>
<tr>
<td>IOWAIT_TICKS</td>
<td>1933617293</td>
<td>5</td>
</tr>
<tr>
<td>AVG_IDLE_TICKS</td>
<td>545952047</td>
<td>7</td>
</tr>
<tr>
<td>AVG_BUSY_TICKS</td>
<td>167700614</td>
<td>8</td>
</tr>
<tr>
<td>AVG_USER_TICKS</td>
<td>120815895</td>
<td>9</td>
</tr>
<tr>
<td>AVG_SYS_TICKS</td>
<td>46655696</td>
<td>10</td>
</tr>
<tr>
<td>AVG_IOWAIT_TICKS</td>
<td>120621649</td>
<td>11</td>
</tr>
<tr>
<td>OS_CPU_WAIT_TIME</td>
<td>5.3432E+13</td>
<td>13</td>
</tr>
<tr>
<td>RSRC_MGR_CPU_WAIT_TIME</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>INBYTES</td>
<td>6.2794E+10</td>
<td>1000</td>
</tr>
<tr>
<td>OUTBYTES</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>AVG_INBYTES</td>
<td>1.7294E+19</td>
<td>1004</td>
</tr>
<tr>
<td>AVG_OUTBYTES</td>
<td>0</td>
<td>1005</td>
</tr>
</tbody>
</table>

17 rows selected.

SQL>

Know Your Application

Experts rely on hit ratios or wait statistics, or sometimes both, but there are situations in which both the hit ratios and the wait statistics can completely fail you. Imagine a situation where all the hit ratios are in the 99 percent range. Also, imagine that the wait statistics don’t show any significant waiting for resources or any contention for latches. Does this mean that your system is running optimally? Well, your system is doing what you asked it to do extremely well, but there’s no guarantee that your SQL code is processing things efficiently. If a query is performing an inordinate number of logical reads, the hit ratios are going to look wonderful. The wait events also won’t show you a whole lot, because they don’t capture the time spent while you were actually using the CPU. However, you’ll be burning a lot of CPU time, because the query is making too many logical reads.

This example shows why it’s important not to rely only on the hit ratios or the wait statistics, but also to look at the major consumers of resources on your instance with an intense focus. Check the Top Sessions list (sorted according to different criteria) on your instance and see if there’s justification for the major consumers to be in that list.
Above all, try not to confuse the symptoms of poor performance with the causes of poor performance. If your latch rate is high, you might want to adjust some initialization parameters right away—after all, isn’t Oracle a highly configurable database? You may succeed sometimes by relying solely on adjusting the initialization parameters, but it may be time to pause and question why exactly the latch rate is so high. More than likely, the high latch rate is due to application coding issues rather than a specific parameter setting. Similarly, you may notice that your system is CPU bound, but the reason may not be slow or inadequate CPU resources. Your application may again be the real culprit because it’s doing too many unnecessary I/Os, even if they’re mostly from the database buffer cache and not disk.

When you’re examining wait ratios, understand that your goal isn’t to make all the wait events go away, because that will never happen. Learn to ignore the unimportant, routine, and unavoidable wait events. As you saw in the previous section, wait events such as the SQL*Net message from client event reflect waits outside the database, so don’t attribute these waits to a poorly performing database. Focus on the total wait time rather than the number of wait events that show up in your performance tables and AWR reports. Also, if the wait events make up only a small portion of response time, there’s no point in fretting about them. As Einstein might say, the significance of wait events is relative—relative to the total response time and relative to the total CPU execution time.

Recently, there has been a surge in publications expounding the virtues of the wait event analysis-based performance approach (also called the wait interface approach). You can always use the buffer hit ratios and the other ratios for a general idea about how the system is using Oracle’s memory and other resources, but an analysis of wait events is still a better bet in terms of improving performance. If you take care of the wait issues, you’ll have taken care of the traditional hit ratios as well. For example, if you want to fix a problem that’s the result of a high number of free buffer waits, you may need to increase the buffer cache. Similarly, if latch free wait events are troublesome, one of the solutions is to check whether you need to add more memory to the shared pool. You may fix a problem due to a high level of waits caused by the direct path reads by increasing the value of the PGA_AGREGATE_TARGET parameter.

**Examining SQL Response Time with the Database Control**

You can use the OEM Database Control to examine quickly the current SQL response time compared to a normal “baseline” SQL response time. The Database Control computes the SQL response time percentage by dividing the baseline SQL response time by the current SQL response time, both expressed in microseconds. If the SQL response time percentage exceeds 100 percent, then the instance is processing SQL statements slower than the baseline times. If the percentage is approximately equal to 100 percent, then the current response time and the baseline response time are equal, and your instance is performing normally. The SQL Response Time section is right on the Database Control home page.

**Using the ADDM to Analyze Performance Problems**

There’s no question that the new ADDM tool should be the cornerstone of your performance-tuning efforts. In Chapter 17, I showed how you can manually get an ADDM report or use the OEM Database Control to view the ADDM analysis. Use the findings and recommendations of the ADDM advisor to fine-tune database performance. Here’s the partial output from an ADDM analysis (invoked by running the addmrpt.sql script located in the $ORACLE_HOME/rdbms/admin directory). Listing 20-18 shows part of an ADDM report.

DETAILED ADDM REPORT FOR TASK 'TASK_1493' WITH ID 1493

Analysis Period: 22-JUL-2008 from 07:01:02 to 17:00:36
Database ID/Instance: 877170026/1
Database/Instance Names: NINA/nina
Host Name: finance1
Database Version: 10.2.0.0
Snapshot Range: from 930 to 940
Database Time: 801313 seconds
Average Database Load: 22.3 active sessions

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

FINDING 1: 24% impact (193288 seconds)
--------------------------------------
The buffer cache was undersized causing significant additional read I/O.

RECOMMENDATION 1: DB Configuration, 24% benefit (193288 seconds)
ACTION: Increase SGA target size by increasing the value of parameter
"sga_target" by 1232 M.
SYMPTOMS THAT LED TO THE FINDING:
Wait class "User I/O" was consuming significant database time. (54% impact [436541 seconds])

FINDING 2: 19% impact (150807 seconds)
--------------------------------------
SQL statements consuming significant database time were found.

RECOMMENDATION 1: SQL Tuning, 4.4% benefit (34936 seconds)
ACTION: Run SQL Tuning Advisor on the SQL statement with SQL_ID "b3bkjk3ybcp5p".
RELEVANT OBJECT: SQL statement with SQL_ID b3bkjk3ybcp5p and
PLAN_HASH 954860671

ADDM may sometimes recommend that you run the Segment Advisor for a certain segments,
or the Automatic SQL Advisor for a specific SQL statement. See Chapter 17 for a detailed analysis of
an ADDM performance report.

Using AWR Reports for Individual SQL Statements

In Chapter 17, you learned how to use AWR reports to analyze the performance of the database
during a time period encompassed by a pair of snapshots. As explained in that chapter, AWR reports
are an excellent source of information for wait-related as well as other instance performance indica-
tors. You can also use the AWR to produce reports displaying performance statistics for a single SQL
statement, over a range of snapshot IDs. Listing 20-19 shows how you can get an AWR report for a
particular SQL statement.

Note The awrsq rpt.sql script seems to run slower than the instance-wide report-generating AWR script,
awr rpt.sql, that you encountered in Chapter 17 during the introduction to AWR.

SQL> @ORACLE_HOME/rdbms/admin/awrsqrpt.sql

Current Instance

<table>
<thead>
<tr>
<th>DB Id</th>
<th>DB Name</th>
<th>Inst Num</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>877170026</td>
<td>PASPROD</td>
<td>1</td>
<td>pasprod</td>
</tr>
</tbody>
</table>

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: text

Instances in this Workload Repository schema

<table>
<thead>
<tr>
<th>DB Id</th>
<th>Inst Num</th>
<th>DB Name</th>
<th>Instance</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 877170026</td>
<td>1</td>
<td>PASPROD</td>
<td>pasprod</td>
<td>prod1</td>
</tr>
</tbody>
</table>

Using 877170026 for database Id
Using 1 for instance number

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: 3

Listing the last 3 days of Completed Snapshots

<table>
<thead>
<tr>
<th>Instance</th>
<th>DB Name</th>
<th>Snap Id</th>
<th>Snap Started</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>pasprod</td>
<td>PASPROD</td>
<td>3829</td>
<td>23 Apr 2008 00:01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3830</td>
<td>23 Apr 2008 02:00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3832</td>
<td>23 Apr 2008 03:00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3833</td>
<td>23 Apr 2008 04:00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3834</td>
<td>23 Apr 2008 05:00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3835</td>
<td>23 Apr 2008 06:00</td>
<td>1</td>
</tr>
</tbody>
</table>

Specify the Begin and End Snapshot Ids

Enter value for begin_snap: 3830
Begin Snapshot Id specified: 3830
Enter value for end_snap: 3835
End  Snapshot Id specified: 3835

Specify the Report Name

The default report file name is 1_3830_3835. To use this name, press <return> to continue, otherwise enter an alternative.
Enter value for report_name
Using the report name 1_3830_3835

Specify the SQL Id

Enter value for sql_id: 9a64dvpzryzza:
Operating System Memory Management

You can use the `vmstat` utility, as explained in Chapter 3, to find out whether enough free memory is on the system. If the system is paging and swapping, database performance will deteriorate and you need to investigate the causes. If the heavy consumption of memory is due to a non-Oracle process, you may want to move that process off the peak time for your system. You may also want to consider increasing the size of the total memory available to the operating system. You can use the `vmstat` command to monitor virtual memory on a UNIX system. The UNIX tool `top` shows you CPU and memory use on your system.

Analyzing Recent Session Activity with an ASH Report

The `V$ACTIVE_SESSION_HISTORY` view records active session activity by sampling all active sessions on a per-second basis. The `V$ACTIVE_SESSION_HISTORY` view’s column data is similar to that of the `V$SESSION` history view, but contains only sample data from active sessions. An active session could be on the CPU, or could be waiting for a wait event that’s not part of the `idle` wait class. When the AWR performs its snapshot, the data in the `V$ACTIVE_SESSION_HISTORY` view is flushed to disk as part of the AWR snapshot data. However, the data in the `V$ACTIVE_SESSION_HISTORY` VIEW isn’t permanently lost when the AWR flushes the view’s contents during its snapshots. Another view, the `DBA_HIST_ACTIVE_SESS_HISTORY`, stores snapshots of the `V$ACTIVE_SESSION_HISTORY` view.

You don’t have to use either of the two `ACTIVE_SESSION_HISTORY`–related views to analyze session history. You can simply produce an ASH report, which contains both the current active session data from the `V$ACTIVE_SESSION_HISTORY` view as well as the historical active session data stored in the `DBA_HIST_ACTIVE_SESS_HISTORY` view. The ASH report shows you the SQL identifier of SQL statements, object information, session information, and relevant wait event information.

You can produce an ASH report by simply going to the OEM Database Control, or by running an Oracle-provided script. In fact, Oracle provides you with two ASH-related scripts, as follows:

- The `ashrpt.sql` script produces an ASH report for a specified duration for the default database.
- The `ashrpti.sql` script produces the same report as the `ashrpt.sql` script, but lets you specify a database instance.

Actually, the `ashrpt.sql` script defaults the DBID and instance number to those of the current instance, and simply runs the `ashrpti.sql` script. Both of the preceding described scripts are available in the `$ORACLE_HOME/rdbms/admin` directory. Here’s how you get an ASH report for your instance:

```sql
SQL> @ORACLE_HOME/rdbms/admin/ashrpt.sql
```

You can then look at the ASH report, which is placed in the directory from which you ran the `ashrpt.sql` script. Chapter 18 explains a typical ASH report, in the section titled “Producing an ASH Report.”

When a Database Hangs

So far in this chapter, you’ve looked at ways to improve performance—how to make the database go faster. Sometimes, however, your problem is something much more serious: the database seems to have stopped all of a sudden! The following sections describe the most important reasons for a hanging or an extremely slow-performing database, and how you can fix the problem ASAP.

One of the first things I do when the database seems to freeze is check and make sure that the archiver process is doing its job. The following sections describe the archiver process.
Handling a Stuck Archiver Process

If your archive log destination is full and there isn’t room for more redo logs to be archived, the archiver process is said to be stuck. The database doesn’t merely slow down—it freezes in its tracks. As you are aware, in an archive log mode the database simply won’t overwrite redo log files until they’re archived successfully. Thus, the database starts hanging when the archive log directory is full. It stays in that mode until you move some of the archive logs off that directory manually.

The Archiver Process

The archiver process is in charge of archiving the filled redo logs. It reads the control files to find out if there are any unarchived redo logs that are full, and then it checks the redo log headers and blocks to make sure they’re valid before archiving them. You may have archiving-related problems if you’re in the archive log mode but the archiver process isn’t running for some reason. In this case, you need to start the archiver process by using the following command:

```
SQL> ALTER SYSTEM ARCHIVE LOG START;
```

If the archiver process is running but the redo logs aren’t being archived, then you may have a problem with the archive log destination, which may be full. This causes the archiver process to become stuck, as you’ll learn in the next section.

Archiver Process Stuck?

When the archiver process is stuck, all database transactions that involve any changes to the tables can’t proceed any further. You can still perform SELECT operations, because they don’t involve the redo logs.

If you look in the alert log, you can see the Oracle error messages indicating that the archiver process is stuck due to lack of disk space. You can also query the V$ARCHIVE view, which holds information about all the redo logs that need archiving. If the number of these logs is high and increasing quickly, you know your archiver process is stuck and that you need to clear it manually. Listing 20-20 shows the error messages you’ll see when the archiver process is stuck.

Listing 20-20. Database Hang Due to Archive Errors

```
$ sqlplus system/system_passwd
ERROR:
ORA-00257: archiver error. Connect internal only, until freed.
$
$ oerr ora 257
00257, 00000, "archiver error. Connect internal only, until freed."
/*Cause: The archiver process received an error while trying to
archive a redo log. If the problem is not resolved soon, the
database will stop executing transactions. The most likely cause
of this message is the destination device is out of space to
store the redo log file.
/*Action: Check archiver trace file for a detailed description
of the problem. Also verify that the device specified in the
initialization parameter ARCHIVE_LOG_DEST is set up properly for
archiving.
$```
You can do either of the following in such a circumstance:

- Redirect archiving to a different directory.
- Clear the archive log destination by removing some archive logs. Just make sure you back up the archive logs to tape before removing them.

Once you create more space in the archive log directory, the database resumes normal operations, and you don’t have to do anything further. If the archiver process isn’t the cause of the hanging or frozen database problem, then you need to look in other places to resolve the problem.

If you see too many “checkpoint not complete” messages in your alert log, then the archiver process isn’t causing the problem. The redo logs are causing the database slowdown, because they’re unable to keep up with the high level of updates. You can increase the size of the redo logs online to alleviate the problem.

**Note** The database logs all connections as SYS in the default audit trail, which is usually the $ORACLE_HOME/rdbms/audit directory. If you don’t have adequate space in that directory, it may fill up eventually, and you’ll get an error when you try logging in as the SYS user. Delete the old audit trail files or choose an alternative location for them.

**System Usage Problems**

You need to check several things to make sure there are no major problems with the I/O subsystem or with the CPU usage. Here are some of the important things you need to examine:

- Make sure your system isn’t suffering from a severe paging and swapping problem, which could result in a slower-performing database.
- Use top, sar, vmstat, or similar operating-system–level tools to check resource usage. Large queries, sorting, and space management operations could all lead to an increase in CPU usage.
- Runaway processes and excessive snapshot processes (SNPs) could gobble excessive CPU resources. Monitor any replication (snapshot) processes or DBMS_JOB processes, because they both use resource-hungry SNP processes. If CPU usage spikes, make sure no unexpected jobs are running in the database. Even if no jobs are executing currently, the SNP processes consume a great deal of CPU because they have to query the job queue constantly.
- High run queues indicate that the system is CPU bound, with processes waiting for an available processor.
- If your disk I/O is close to or at 100 percent and you’ve already killed several top user sessions, you may have a disk controller problem. For example, the 100 percent busy disk pack might be using a controller configured to 16-bit, instead of 32-bit like the rest of the controllers, causing a severe slowdown in I/O performance.

**Excessive Contention for Resources**

Usually when people talk about a database hang, they’re mistaking a severe performance problem for a database hang. This is normally the case when there’s severe contention for internal kernel-level resources such as latches and pins. You can use the following query to find out what the contention might be:

```sql
SQL> SELECT event, count(*)
   2  FROM v$session_wait
   3  GROUP BY event;
```
EVENT                                      COUNT(*)
---------------------------------------------------
PL/SQL lock timer                            2
Queue Monitor Wait                           1
SQL*Net message from client                  61
SQL*Net message to client                    1
jobq slave wait                              1
pmon timer                                   1
rdbms ipc message                            11
smon timer                                   1
wakeup time manager                          1

9 rows selected.

SQL>

The previous query doesn’t reveal any significant contention for resources—all the waits are for idle events.

If your database is performing an extremely high number of updates, contention for resources such as undo segments and latches could potentially be a major source of database-wide slowdowns, making it seem sometimes like the database is hanging. In the early part of this chapter, you learned how to analyze database contention and wait issues using the V$SESSION_WAIT view and the AWR output. On Windows servers, you can use the Performance Monitor and Event Monitor to locate possible high resource usage.

Check for excessive library cache contention if you’re confronted by a database-wide slowdown.

Locking Issues

If a major table or tables are locked unbeknownst to you, the database could slow down dramatically in short order. Try running a command such as SELECT * FROM persons, for example, where persons is your largest table and is part of just about every SQL statement. If you aren’t sure which tables (if any) might be locked, you can run the following statement to identify the table or index that’s being locked, leading to a slow database:

```
SQL> SELECT l.object_id, l.session_id,
          2 l.oracle_username, l.locked_mode,
          3 o.object_name
       FROM V$LOCKED_OBJECT l,
           DBA_OBJECTS o
       WHERE o.object_id=l.object_id;
```

<table>
<thead>
<tr>
<th>OBJECT_ID</th>
<th>SESSION_ID</th>
<th>ORACLE_USERNAME</th>
<th>LOCKED_MODE</th>
<th>OBJECT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>6699</td>
<td>22</td>
<td>NICHOLAS</td>
<td>6</td>
<td>EMPLOYEES</td>
</tr>
</tbody>
</table>

SQL>

As the preceding query and its output show, user Nicholas has locked up the Employees table. If this is preventing other users from accessing the table, you have to remove the lock quickly by killing the locking user’s session. You can get the locking user’s SID from the session_id column in the preceding output, and the V$SESSION view gives you the SERIAL# that goes with it. Using the ALTER SYSTEM KILL . . . command, you can then kill the offending session. The same analysis applies to a locked index, which prevents users from using the base table. For example, an attempt to create an index or rebuild it when users are accessing the table can end up inadvertently locking up the table.

If there’s a table or index corruption, that could cause a problem with accessing that object(s). You can quickly check for corruption by running the following statement:
Abnormal Increase in Process Size

On occasion, there might be a problem because of an alarming increase in the size of one or more Oracle processes. You have to be cautious in measuring Oracle process size, because traditional UNIX-based tools can give you a misleading idea about process size. The following sections explain how to measure Oracle process memory usage accurately.

What Is Inside an Oracle Process?

An Oracle process in memory has several components:

- **Shared memory**: This is the SGA that you’re so familiar with.
- **The executable**: Also known as TEXT, this component consists of the machine instructions. The TEXT pages in memory are marked read-only.
- **Private data**: Also called DATA or heap, this component includes the PGA and the User Global Area (UGA). The DATA pages are writable and aren’t shared among processes.
- **Shared libraries**: These can be private or public.

When a new process starts, it requires only the DATA (heap) memory allocation. Oracle uses the UNIX implementation of shared memory. The SGA and TEXT components are visible to and shared by all Oracle processes, and they aren’t part of the cost of creating new Oracle processes. If 1,000 users are using Oracle Forms, only one set of TEXT pages is needed for the Forms executable.

Unfortunately, most operating system tools such as ps and top give you a misleading idea as to the process size, because they include the common shared TEXT sizes in individual processes. Sometimes they may even include the SGA size. Solaris’s pmap and HP’s glance are better tools from this standpoint, as they provide you with a more accurate picture of memory usage at the process level.

**Note** Even after processes free up memory, the operating system may not take the memory back, indicating larger process sizes as a result.

Measuring Process Memory Usage

As a result of the problems you saw in the previous section, it’s better to rely on Oracle itself for a true indication of its process memory usage. If you want to find out the total DATA or heap memory size (the biggest nonsharable process memory component), you can do so by using the following query:

```
SQL> SELECT value, n.name|| '('||s.statistic#||')', sid 
FROM v$sesstat s, v$statname n 
WHERE s.statistic# = n.statistic# 
AND n.name like '%ga memory%' 
ORDER BY value;
```

If you want to find out the total memory allocated to the PGA and UGA memory together, you can issue the command in the next example. The query reveals that a total of more than 367MB of memory is allocated to the processes. Note that this memory is in addition to the SGA memory allocation, so you need to make allowances for both types of memory to avoid paging and swapping issues.
If the query shows that the total session memory usage is growing abnormally over time, you might have a problem such as a memory leak. A telltale sign of a memory leak is when Oracle's memory usage is way outside the bounds of the memory you've allocated to it through the initialization parameters. The Oracle processes are failing to return the memory to the operating system in this case. If the processes continue to grow in size, eventually they may hit some system memory barriers and fail with the ora 4030 error:

```
$ oerr ora 4030
04030, 00000, "out of process memory when trying to allocate %s bytes (%s,%s)"
// *Cause: Operating system process private memory has been exhausted
$
```

Note that Oracle tech support may request that you collect a heap dump of the affected Oracle processes (using the oradebug tool) to fix the memory leak problem.

If your system runs out of swap space, the operating system can’t continue to allocate any more virtual memory. Processes fail when this happens, and the best way to get out of this mess is to see whether you can quickly kill some of the processes that are using a heavy amount of virtual memory.

### Delays Due to Shared Pool Problems

Sometimes, database performance deteriorates dramatically because of inadequate shared pool memory. Low shared pool memory relative to the number of stored procedures and packages in your database could lead to objects constantly aging out of the shared pool and having to be executed repeatedly.

### Problems Due to Bad Statistics

As you know by now, the Oracle Cost-Based Optimizer (CBO) needs up-to-date statistics so it can pick the most efficient method of processing queries. If you’re using the Automatic Optimizer Statistics Collection feature, Oracle will naturally keep optimizer statistics up to date for you without any effort on your part. However, if you have deactivated the automatic statistics collection process, you could run the risk of not providing representative statistics to the CBO.

If you don’t collect statistics regularly while lots of new data is being inserted into tables, your old statistics will soon be out of date, and the performance of critical SQL queries could head south. DBAs are under time constraints to collect statistics overnight or over a weekend. Sometimes, they may be tempted to use a small sample size while using the DBMS_STATS package to collect statistics. This could lead to unreliable statistics, resulting in the slowing down of query processing.

### Collecting Information During a Database Hang

It can sometimes be downright chaotic when things come to a standstill in the database. You might be swamped with phone calls and anxious visitors to your office who are wondering why things are slow. Oftentimes, especially when serious unknown locking issues are holding up database activity,
it’s tempting just to bounce the database because usually that clears up the problem. Unfortunately, you don’t know what caused the problem, so when it happens again, you’re still just as ignorant as you were the first time. Bouncing the database also means that you’re disconnecting all active users, which may not always be a smart strategy.

It’s important that you collect some information quickly for two reasons. First, you might be able to prevent the problem next time or have someone in Oracle tech support (or a private firm) diagnose the problem using their specialized tools and expertise in these matters. Second, most likely a quick shutdown and restart of the database will fix the problem for sure (as in the case of some locking situations, for example). But a database bounce is too mighty a weapon to bring to bear on every similar situation. If you diagnose the problem correctly, simple measures may prevent the problem or help you fix it when it does occur. The following sections describe what you need to do to collect information on a slow or hanging database.

**Using the Database Control’s Hang Analysis Page**

You can use OEM’s Database Control during an instance slowdown to see a color-coded view of all sessions in the database. The Hang Analysis page provides the following information:

- Instantaneously blocked sessions
- Sessions in a prolonged wait state
- Sessions that are hung

Figure 20-1 shows the Database Control Hang Analysis page, which you can access from the Performance page. Click the Hang Analysis link under the Additional Monitoring Links section.

![The Database Control Hang Analysis page](image-url)
Gathering Error Messages

The first thing you do when you find out the database suddenly slowed down or is hanging is to look in some of the log files where Oracle might have sent a message. Quickly look in the alert log file to see whether there are any Oracle error messages or any other information that could pinpoint any problems. You can check the directory for background dumps for any other trace files with error messages. I summarize these areas in the following discussion.

Getting a Systemstate Dump

A systemstate dump is simply a trace file that is output to the user dump directory. Oracle (or a qualified expert) can analyze these dumps and tell you what was going on in the database when the hanging situation occurred. For example, if logons are slow, you can do a systemstate dump during this time, and it may reveal that most of the waits are for a particular type of library cache latch. To get a systemstate dump (for level 10), run the following command:

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name systemstate level 10';
Session altered.
```

**Caution** Oracle Corp. strongly warns against customers setting events on their own. You may sometimes end up causing more severe problems when you set events. Please contact Oracle technical support before you set any event. For example, the event 10235 has been known to cause heavy latch contention.

You can send the resulting output to Oracle so it can analyze the output for you. Note that at this stage, you need to open a technical assistance request (TAR) with Oracle technical support through MetaLink (http://metalink.oracle.com). (The hanging database problem gets you a priority level 1 response, so you should hear from an analyst within minutes.) Oracle technical support may ask you for more information, such as a core dump, and ask you to run a debugger or another diagnostic tool and FTP the output to them.

Using the Hanganalyze Utility

The systemstate dumps, although useful, have several drawbacks, including the fact that they dump out too much irrelevant information and take too much time to complete, leading to inconsistencies in the dump information. The newer hanganalyze utility is more sophisticated than a systemstate dump. Hanganalyze provides you with information on resources each session is waiting for, and what is blocking access to those resources. The utility also provides you with a dependency graph among the active sessions in the database. This utility isn’t meant to supplant the systemstate dumps; rather, you should use it to help make systemstate dumps more meaningful. Again, use this utility in consultation with Oracle technical support experts. Here’s a typical HANGANALYZE command:

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name HANGANALYZE level 3';
```

THE PROMISE AND THE PERFORMANCE

A few years ago, the Immigration and Naturalization Service (INS) of the United States created a new $36 million Student and Exchange Visitor Information System (SEVIS) to replace the old paper-based methods the INS had used for years to track foreign students in U.S. educational institutions. More than 5,400 high schools, colleges, and universities have to use SEVIS to enter the necessary information about enrolled students from other countries.
The INS had imposed a deadline by which all educational institutions had to switch over fully to the SEVIS system. However, it extended the deadline by at least two weeks amid several complaints about the system working slowly, if at all. Here are a few of those complaints from users across the country:

- Some employees in Virginia could enter data only in the mornings, before the West Coast institutions logged onto the system. In the afternoons, the system slowed to a crawl.

- From the University of Minnesota came complaints that that the officials were “completely unable” to use the system at all. The users mentioned that the system “was really jammed with users trying to get on.” They also complained that the system was “unbelievably slow.” An INS spokesperson admitted that the system had been “somewhat sluggish” and that schools were having trouble using the SEVIS system.

- The University of North Carolina complained that the situation, if it continued any further, was going to be “a real nightmare” and that it was already “starting to cause some problems.”

- One worker at a college in Michigan was quoted as saying this in frustration: “Please tell me what I’m doing wrong, or I am going to quit.”

The INS realized the colleges and universities weren’t going to meet the deadline, and they announced a grace period after saying that “upgrades to the system” had greatly improved performance.

Behind the SEVIS system is an Oracle database that was performing awfully slowly. The system apparently couldn’t scale well enough. When a large number of users got on, it ground to a halt. Obviously, the system wasn’t configured to handle a high number of simultaneous operations. Was the shared server approach considered, for example? How were the wait statistics? I don’t know the details. I do know that the Oracle database is fully capable of meeting the requirements of an application such as this. I picked this example to show that even in high-profile cases, DBAs sometimes have to eat humble pie when the database isn’t tuned properly and consequently performance doesn’t meet expectations.

A Simple Approach to Instance Tuning

Most of the instance tuning that DBAs perform is in response to a poorly performing database. The following sections present a brief summary of how you can start analyzing the instance to find out where the problem lies.

First, examine all the major resources such as the memory, CPUs, and storage subsystem to make sure your database isn’t being slowed down by bottlenecks in these critical areas.

**Note** Collecting baseline data about your database statistics, including wait events, is critically important for troubleshooting performance issues. If you have baseline data, you can immediately check whether the current resource-usage patterns are consistent with the load on the system.

What’s Happening in the Database?

It isn’t rare for a single user’s SQL query to cause an instance-wide deterioration in performance if the query is bad enough. SQL statements are at the root of all database activity, so you should look at what’s going on in the database right now. The following are some of the key questions to which you need to find answers:
Who are the top users in your Top Sessions display?
What are the exact SQL statements being executed by these users?
Is the number of users unusually high compared to your baseline numbers for the same time period?
Is the load on the database higher than what your baseline figures show for the time of the day or the time of the week or month?
What top waits can you see in the V$SESSION or the V$SESSION_WAIT view? These real-time views show the wait events that are happening right now or that have just happened in the instance. You have already seen how you can find out the actual users responsible for the waits by using other V$ views.

A critical question here is whether the performance problem *du jour* is something that’s sudden without any forewarnings or if it’s caused by factors that have been gradually creeping up on you. Under the latter category are things such as a growing database, a larger number of users, and a larger number of DML operation updates than what you had originally designed the system for. These types of problems may mean that you need to redesign at least some of your tables and indexes with different storage parameters, and other parameters such as freelists. If, on the other hand, the database has slowed down suddenly, you need to focus your attention on a separate set of items.

Your best bet for analyzing what’s happening in the database currently is to probe the ASH. You can easily find out the users, the objects, and the SQL causing the waits in your instance by using the queries based on V$ACTIVE_SESSION_HISTORY, which I explained in the section “Using the V$ACTIVE_SESSION_HISTORY View” earlier in this chapter. You can also run a quick ASH report encompassing the past few minutes to see where the bottlenecks may lie, and who is causing them.

---

**Tip** The OEM Database Control provides the Gather Statistics Wizard, which you can use if there are performance issues due to out-of-date statistics for fixed and dictionary objects.

---

### Using the OEM Database Control to Examine Database Performance

I reviewed the OEM Database Control and Grid Control in Chapter 19. It’s nice to learn about all the different V$ views regarding waits and performance, but nothing beats the Database Control when it comes to finding out quickly what’s happening in your database at any given time. I present a simple approach to using the Database Control’s various performance-related pages in the following sections.

#### The Database Control Home Page

Start your performance analysis by looking at the following three instance performance charts on the Database Control’s home page. Figure 20-2 shows the Database Control home page.
CHAPTER 20  ■  PERFORMANCE TUNING: TUNING THE INSTANCE

Figure 20-2 the OEM Database Control home page

Host CPU
The CPU consumption on the host server is shown in the form of a bar chart. The chart shows two categories: the instance and another category called other, which represents all the processes that don’t belong to the database instance.

Active Sessions
The Active Sessions chart is a key chart, because it shows the extent of performance bottlenecks in your database instance. The chart consists of three components:

- CPU
- User I/O
- Wait

The Active Sessions chart shows the time consumed by the three items: CPU, User I/O, and Wait. You can drill down to each of these categories by clicking on the respective links. Note that the Wait category includes all waits in the instance except User I/O, which is shown in a separate category by itself.

SQL Response Time
The SQL Response Time chart provides a quick idea about how efficiently the instance is executing SQL statements. If the current SQL response ratio exceeds the baseline response ratio of 100 percent, then the SQL statements are executing slower than “normal.” If the SQL Response Time shows a small response percentage, then you have inefficient SQL statement processing in the instance.
Note: If you have a pre-Oracle Database 10g database, you may have to configure certain things for the SQL activity metrics to show up in the SQL Response Time chart. You do this by using the Database Configuration wizard, which you activate by clicking the Configure button in the SQL Activity Monitoring file under the Diagnostic Summary.

Using the ADDM Analysis in the Performance Analysis Section

The Performance Analysis section of the Database Control home page summarizes the most recent ADDM analysis. Figure 20-3 shows the Performance Analysis section. From here, you can click any of the findings to analyze any performance issues further. ADDM reports, which use the AWR statistics, provide you with a quick top-down analysis of instance activity.

Figure 20-3. Summary of ADDM findings

Using the Database Performance Page

The Database Performance page is your jump-off point for evaluating instance performance. This page helps you do the following:

- Check for problems both within the database and the system.
- Run the ASH report to get a quick session-sampling data-based performance diagnostic report.
- Quickly see what bottlenecks exist within the system.
- Run ADDM reports.
- For slow or hung systems, access the Memory Access Mode.
Using the Memory Access Mode

You can view the Performance page in the default mode, which is called the SQL Access Mode, or the new Memory Access Mode. The SQL Access Mode works through SQL statements that mostly query the V$ dynamic performance view to obtain instance performance data. However, when the database is running painfully slowly, or is completely hung, running in the SQL Access Mode puts further stress due to the additional parsing and execution load of the SQL statements run by the OEM interface to diagnose instance performance. If your instance is already facing heavy library cache contention, your attempt to diagnose the problem will exacerbate the situation.

Oracle recommends that you switch to the Memory Access Mode while diagnosing slow or hung systems. Under this mode, the database gets its diagnostic information straight from the SGA, using more lightweight system calls than the resource-intensive SQL statements that are employed during the default SQL Access Mode. Because the data is sampled more frequently under the Memory Access Mode, you’re less likely to miss events that span short intervals of time as well. Figure 20-4 shows how to use the drop-down window to switch between the Memory Access Mode and the SQL Access Mode.

![Using the Performance page in the Memory Access Mode](image)

The following sections describe the main charts you’ll see on the Database Performance page.
Host

The Host chart indicates whether there is a CPU bottleneck. If the number of users is low while the Host section shows a high run-queue length, it means that the database users may not be the main contributing factor for high CPU consumption. Look at what else may be running on your system and consuming the CPU resources.

Average Active Sessions

The Average Active Sessions chart shows performance problems within your instance, by focusing on the wait events in your instance. This is the key chart in the Performance page and should be the starting point of a performance analysis using the OEM. Figure 20-5 shows the Average Active Sessions chart. The chart shows you which of the active sessions are waiting on CPU and which are waiting on an event.

![The Average Active Sessions page of the Database Control](image)

Figure 20-5. The Average Active Sessions page of the Database Control

The Average Active Sessions chart is color coded for your benefit. Green represents users on the CPU and the other colors show users waiting on various events such as disk I/O, locks, or network communications. Here’s how you can tell whether you have too many waits in your instance: if the level of waits is twice the Max CPU line, you have too many waits, and should look at tuning the instance.
To the right of the Average Active Sessions screen, you can see the breakdown of the components that contribute to session time. For example, if you see user I/O as the main culprit for high waits, you can click this component to find out details about the wait. Figure 20-5 also shows the buttons you can click to run the ADDM or get an ASH report.

You can also click the link for Top Activity to find out details about the sessions that are most responsible for waits in your instance right now. Figure 20-6 shows the Top Activity page of the Database Control. Database activity is ranked into Top SQL and Top Sessions. You can run the SQL Tuning Advisor from here to get tuning recommendations about the top SQL statements.

If you suspect that an individual session is wait bound or you get complaints from particular users that their sessions are running slowly, you can examine the Top Sessions page. You can go the Top Sessions page by clicking the Top Sessions link under the Additional Monitoring Links group on the Performance page. Once you get to the Top Sessions page, click the username and SID you’re interested in. That takes you to the Session Details page for that session. By clicking the Wait Event History tab in the Session Details page, you can see the nature of the recent waits for that session. Figure 20-7 shows the Wait Event History for a session.
The Performance Data Report Page

You can get to the Performance Data Report page by clicking the Create ASH Report button in the Average Active Sessions screen on the Database Control’s Performance home page. The AWR reports are good for analyzing instance performance, but they’re usually collected at 30-minute or 1-hour intervals. What if you have a three-to-four-minute performance spike that’s not shown in the aggregated AWR report? ASH reports focus on session-sampling data over a recent period of time.

When you click the Create ASH Report button, you’re given a choice as to the time period over which you want to create your ASH report. You can choose a time period that lies within the last seven days, because that’s how long the AWR saves its statistics. Remember that ASH statistics are saved in the AWR repository. Figure 20-8 shows the ASH report, which relies on the V$ACTIVE_SESSION_HISTORY view. This is the same ASH report that you can produce by running the ashrpt.sql script. It contains information about the following items:

- Top Events
- Load Profile
- Top SQL
- Top Sessions, including Top Blocking Sessions
- Other entities causing contention in the instance, including Top Database Objects, Top Database Files, and Top Latches
- Activity Over Time
Are There Any Long-Running Transactions?

You can use the V$SQL view, as shown in the following example, to find out which of the SQL statements in the instance are taking the most time to finish and are the most resource intensive. The query ranks the transactions by the total number of elapsed seconds. You can also rank the statements according to CPU seconds used.

```
SQL> SELECT hash_value, executions,
       2    ROUND (elapsed_time/1000000, 2) total_time,
       3    ROUND (cpu_time/1000000, 2) cpu_seconds
       4  FROM (SELECT * FROM V$SQL
       5     ORDER BY elapsed_time desc);
```

<table>
<thead>
<tr>
<th>HASH_VALUE</th>
<th>EXECUTIONS</th>
<th>TOTAL_TIME</th>
<th>CPU_SECONDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>238087931</td>
<td>168</td>
<td>9.51</td>
<td>9.27</td>
</tr>
<tr>
<td>1178035321</td>
<td>108</td>
<td>4.98</td>
<td>5.01</td>
</tr>
</tbody>
</table>

Once you have the value for the HASH_VALUE column from the query you just ran, it’s a simple matter to find out the execution plan for this statement, which is in your library cache. The following query uses the V$SQL_PLAN view to get you the execution plan for your longest-running SQL statements:

```
SQL> SELECT * FROM V$SQL_PLAN WHERE hash_value = 238087931;
```
Is Oracle the Problem?

Just because your database users are complaining, you shouldn’t be in a hurry to conclude that the problem lies within the database. After all, the database doesn’t work in a vacuum—it runs on the server and is subject to the resource constraints and bottlenecks of that server. If the non-Oracle users on the server are using up critical resources such as CPU processing and disk I/O, your database may be the victim of circumstances, and you need to look for answers outside the database. That’s why it’s critical that DBAs understand how to measure general system performance, including memory, the disk storage subsystem, the network, and the processors. In the following sections you’ll take a look at the system resources you should focus on.

Is the Network Okay?

One of the first things you need to do when you’re investigating slowdowns is to rule out network-related problems. Quite often, users complain of being unable to connect to the system, or being abruptly disconnected from the system. Check your round-trip ping times and the number of collisions. Your network administrator should check the Internet connections and routers.

On the Oracle end, you can check the following dynamic views to find out if there’s a slowdown due to a network problem. The V$SESSION_EVENT view shows the average amount of time Oracle waits between messages in the average wait column. The V$SESSION_WAIT view, as you’ve seen, shows what a session is waiting for, and you can see whether waits for network message transport are higher than normal.

If the time for SQL round-trips is extremely long, it could reflect itself as a high amount of network-related wait time in the V$ views. Check to see whether your ping time for network round-trips has gone up appreciably. You should discuss with your network administrator what you can do to decrease the waits for network traffic.

You may explore the possibility of setting the parameter TCP,NODELAY=TRUE in your sqlnet.ora file. This results in TCP sending packets without waiting, thus increasing response time for real-time applications.

If the network seems like one of your constant bottlenecks, you may want to investigate the possibility of using the shared server approach instead of the dedicated server approach for connecting users to your database. By using a shared server and its connection pooling feature, you can reduce the number of physical network connections and thus help your application scale more efficiently to large user bases.

Is the System CPU Bound?

Check the CPU performance to make sure a runaway process or a valid Oracle process isn’t hogging one or more processes and contributing to the system slowdown. Often, killing the runaway processes or the resource-hogging sessions will bring matters to a more even keel. Using the OEM Database Control, you can get a quick idea about the breakdown of CPU usage among parse, recursive, and other usage components.

Normally, you should expect to see no more than 20 to 25 percent of total CPU usage by the system itself, and about 60 to 65 percent usage by the Oracle application. If the system usage is close to 50 percent, it’s an indication that there are too many system calls, for example, which leads to excessive use of the processors.

As you learned earlier in this chapter, the V$SESSTAT view shows CPU usage by session. Using the following query, you can find out the top CPU-using Oracle sessions. You may want to look into the actual SQL that these sessions are executing.
Is the System I/O Bound?

Before you go any further analyzing other wait events, it’s a good idea to rule out whether you’re limited by your storage subsystem by checking your I/O situation. Are the read and write times on the host system within the normal range? Is the I/O evenly distributed, or are there hot spots with one or two disks being hit hard? If your normal, healthy I/O rates are 40–50/ms and you’re seeing an I/O rate of 80/ms, obviously something is amiss. The AWR and ASH reports include I/O times (disk read and disk write) by datafile. This will usually tip you off about what might be causing the spike. For example, if the temporary tablespace datafiles are showing up in the high I/O list often, that’s usually an indication that disk sorting is going on, and you need to investigate that further.

You can use the V$SYSTEM_EVENT view to verify whether the top wait events include events such as db file scattered read, db file sequential read, db file single write, and Logfile parallel write, which are database file, log file, and redo log file-related wait events. You can run an AWR report and identify the tablespaces and datafiles causing the I/O contention. Use the V$SQLAREA view, as shown in this chapter, to identify SQL statements that lead to high disk reads and have them tuned.

Too often, a batch program that runs into the daytime could cause spikes in the I/O rates. Your goal is to see whether you can rule out the I/O system as the bottleneck. Several of the wait events that occur in the Oracle database, such as the db file sequential read and db file scattered read waits, can be the result of extremely heavy I/O in the system. If the average wait time for any of these I/O-related events is significant, you should focus on improving the I/O situation. You can do two things to increase the I/O bandwidth: reduce the I/O workload or increase the I/O bandwidth. In Chapter 21, you learned how you can reduce physical I/Os by proper indexing strategies and the use of efficient SQL statements.

Improving SQL statements is something that can’t happen right away, so you need to do other things to help matters in this case. This means you need to increase the I/O bandwidth by doing either or both of the following:

- Make sure that the key database objects that are used heavily are spread evenly on the disks.
- Increase the number of disks.

Storage disks are getting larger and larger, but the I/O rates aren’t quite keeping up with the increased disk sizes. Thus, servers are frequently I/O bound in environments with large databases. Innovative techniques such as file caching might be one solution to a serious I/O bottleneck. On average, about 50 percent of I/O activity involves less than 5 percent of the total datafiles in your database, so caching this limited number of hot files should be a win. Caching gives you the benefit of read/write operations at memory speeds, which could be 200 times faster than disk speed. You can include your temp, redo log, and undo tablespace files, as well as the most frequently used table and index datafiles on file cache accelerators.

It’s possible for large segments to waste a lot of disk space due to fragmentation caused by update and delete operations over time. This space fragmentation could cause severe performance degradation. You can use the Segment Advisor to find out which objects are candidates for a space reclamation exercise due to excessive fragmentation within the segment.
Is the Database Load Too High?

If you have baseline numbers for the database load, you can see whether the current load on the database is relatively too high. Pay attention to the following data, which you can obtain from the V$SYSSTAT view: physical reads and writes, redo size, hard and soft parse counts, and user calls. You can also check the Load Profile section of the AWR report for load data that’s normalized over transactions and over time.

Checking Memory-Related Issues

As you saw earlier in this chapter, high buffer cache and shared pool hit ratios aren’t guarantees of efficient instance performance. Sometimes, an excessive preoccupation with hit ratios can lead you to allocate too much memory to Oracle, which opens the door to serious problems such as paging and swapping at the operating-system level. Make sure that the paging and swapping indicators don’t show anything abnormal. High amounts of paging and swapping slow down everything, including the databases on the server.

Due to the virtual memory system used by most operating systems, a certain amount of paging is normal and to be expected. If physical memory isn’t enough to process the demand for memory, the operating system will go to the disk to use its virtual memory, and this results in a page fault. Processes that result in high page faults are going to run slowly.

When it comes to Oracle memory allocation, don’t forget to pay proper attention to PGA memory allocation, especially if you’re dealing with a DSS-type environment. Databases that perform a large number of heavy sorting and hashing activities need a high amount of PGA memory allocation. The database self-tunes the PGA, but you still have to ensure that the pga_aggregate_target value is high enough for Oracle to perform its magic.

Tip  
Unlike the SGA, the PGA memory allocation isn’t immediately and permanently allocated to the Oracle database. Oracle is allowed to use PGA memory up to the limit specified by the PGA_TARGET parameter. Once a user’s job finishes executing, the PGA memory used by the job is released back to the operating system. Therefore, you shouldn’t hesitate to use a high value for the PGA_TARGET initialization parameter. There’s absolutely no downside to using a high number, and it guarantees that your instance won’t suffer unnecessary disk sorting and hashing.

See whether you can terminate a few of the Top Sessions that seem to be consuming inordinate amounts of memory. It’s quite possible that some of these processes are orphan or runaway processes.

Are the Redo Logs Sized Correctly?

If the redo logs are too few or if they are too small relative to the DML activity in the database, the archiver process will have to work extra hard to archive the filled redo log files. This may cause a slowdown in the instance. It’s easy to resize the redo logs or add more redo log groups. When you use the FAST_START_MTTR_TARGET parameter to impose a ceiling on instance recovery time, Oracle will checkpoint as frequently as necessary to ensure the instance can recover from a crash within the MTTR setting. You must ensure that the redo logs are sized large enough to avoid additional checkpointing. You can get the optimal redo log size from the OPTIMAL_LOGFILE_SIZE column from the V$INSTANCE_RECOVERY view. You can also use the Database Control’s Redo Log Groups page to get advice on sized redo logs. As a rule of thumb, Oracle recommends that you size the log files so they switch every 20 minutes.
Is the System Wait Bound?

If none of the previous steps indicated any problems, chances are that your system is suffering from a serious contention for some resource such as library cache latches. Check to see whether there’s contention for critical database resources such as locks and latches. For example, parsing similar SQL statements leads to an excessive use of CPU resources and affects instance performance by increasing the contention for the library cache or the shared pool. Contention for resources manifests itself in the form of wait events. The wait event analysis earlier in this chapter gave you a detailed explanation of various critical wait events. You can use AWR and ASH reports to examine the top wait events in your database.

The V$SESS_TIME_MODEL (and the V$SYS_TIME_MODEL) view is useful in finding out accumulated time for various database operations at the individual session level. This view helps you understand precisely where most of the CPU time is being spent. As explained in Chapter 17, the V$SESS_TIME_MODEL view shows the following things, among others:

- **DB time**, which is the elapsed time spent in performing database user-level calls.
- **DB CPU** is the amount of CPU time spent on database user-level calls.
- **Background CPU time** is the amount of CPU time used by the background processes.
- **Hard parse elapsed time** is the time spent hard parsing SQL statements.
- **PL/SQL execution elapsed time** is the amount of time spent running the PL/SQL interpreter.
- **Connection management call elapsed time** is the amount of time spent making session connect and disconnect calls.

You can use segment data in the V$SEGMENT_STATISTICS view to find out the hot table and index segments causing a particular type of wait, and focus on eliminating (or reducing, anyway) that wait event.

The Compare Periods Report

Let’s say you encounter a situation where one of your key nightly batch jobs is running past its time window and continuing on into the daytime, where it’s going to hurt the online OLTP performance. You know the batch job used to finish within the stipulated time, but now it’s tending to take a much longer time. As of Oracle Database 10g Release 2, you can use the Database Control’s Compare Periods Report to compare the changes in key database metrics between two time intervals. As you know, an AWR snapshot captures information between two points in time. However, you can use the Time Periods Comparison feature to examine the difference in database metrics between two different time intervals or periods, by analyzing performance statistics captured by two sets of AWR snapshots. If your nightly batch job ran just fine on Tuesday but was slow on Wednesday, you can find out why, using the Compare Periods Report.

To use the Compare Periods Report, use the following steps:

1. In the Database Control home page, click the Performance tab.
2. Under the Additional Monitoring Links group, click the Snapshots link.
3. In the drop-down list for Actions, select Compare Periods and click Go.
4. The Compare Periods: First Period End page appears. You must select the start time for the comparison analysis by selecting an ending snapshot ID for the first period. You may also choose a time period, if you wish, instead of the ending snapshot ID. Click Next.
5. The Compare Periods: Second Period Start page is next. You must select a snapshot ID to mark the beginning of the second period. Click Next.
6. The Compare Periods: Second Period End page is next. You select the ending snapshot for the second period on this page and click Next.

7. The Compare Periods: Review page is next, as shown in Figure 20-9. It shows the first period and second period beginning and ending snapshot IDs. After confirming that the first and second period ranges are correct, click Finish.

8. You’ll now get the Compare Period: Results page, which summarizes the differences in key database metrics between the two periods.

   Going through the differences in key database metrics between the two periods helps you identify the root causes of the performance slowdown in the latter period when compared to the earlier “good” period. You can also view the database configuration differences between the two periods as well.

   To compare the two periods in detail and to drill down into various items such as SQL statements executed, SGA usage, and so on, click the Report link in the Compare Periods: Results page. You can see a nicely formatted report comparing the two periods on the basis of configuration, top five timed events, and the load profile. By viewing the various statistics for the two periods, you can determine whether there was excessive load or some such thing during the second period.

   At the bottom of the report, you’ll find the Report Details section, with links for various items like wait events, I/O statistics, segment statistics, and SGA statistics. You can click any of these links to drill down into what exactly went on inside the database during the two periods. For example, by clicking the SQL Statistics link, you can get to the top ten SQL statements compared by execution time, CPU time, buffer gets, physical reads, and so on. For example, Figure 20-10 shows the top ten SQL statements compared on the basis of physical reads during each period.
Instead of running myriad SQL scripts and manually examining various statistics, as is the tradition, you can use the Compare Periods feature to zoom in quickly on the reasons for deterioration in recent database performance compared to a past period of time.

Eliminating the Contention

Once you identify wait events due to contention in the system, you need to remove the bottleneck. Of course, this is easier said than done in the short run. You may be able to fix some contention problems right away, whereas you may need more time with others. Problems such as high db_file_scattered_read events, which are due to full table scans, may indicate that the I/O workload of the system needs to be reduced. However, if the reduction in I/O requires creating new indexes and rewriting SQL statements, obviously you can’t fix the problem right away. You can’t add disks and rearrange objects to reduce hot spots right away either. Similarly, most latch contention requires changes at the application level. Just make sure you don’t perform a whole bunch of changes at once—you’ll never be able to find out what fixed the problem (or in some cases, what made it worse).

The trick, as usual, is to go after the problems you can fix in the short run. Problems that you can fix by changing the memory allocation to the shared pool or the buffer cache you can easily handle almost immediately by dynamically adjusting the cache values. You can also take care of any changes that concern the redo logs right away. If you notice one or two users causing a CPU bottleneck, it may be a smart idea to kill those sessions so the database as a whole will perform better. As you know, prevention is much better than a cure, so consider using the Oracle Database Resource Manager tool (Chapter 12 shows you in detail how to use the Database Resource Manager) to create resource groups and prevent a single user or group from monopolizing the CPU usage.
If intense latch contention is slowing your database down, you probably should be setting the CURSOR_SHARING initialization parameter’s value to FORCE or SIMILAR to ameliorate the situation.

Most other changes, though, may require more time-consuming solutions. Some changes may even require major changes in the code or the addition or modification of important indexes. However, even if the problem isn’t fixed immediately, you have learned your craft, and you’re on the right path to improving instance performance.

Although I’ve discussed various methods and techniques that use SQL scripts to analyze instance performance, try to make the OEM Database Control (or Grid Control) the center of your database performance monitoring, and use Oracle’s powerful tools, such as the ADDM, to save time. The AWR and ASH reports are also highly useful when searching for root causes of performance problems.

Real Application Testing

One of the biggest problems facing a DBA is how to figure out the potential performance impact of a major system change, such as an upgrade to a new release of the database server, for example. Several third-party tools can help you test the changes, but Oracle provides you the new Real Application Testing option, also known as Total Recall, which is an easy-to-use solution that enables you to test the impact of system changes in a test environment before introducing those changes in the production system. You can thus safely introduce changes into your system without any adverse impact. Real Application Testing consists of two distinct features, Database Replay and the SQL Performance Analyzer, that together provide a complete solution to assess the impact of major system changes. I explain these features in the following sections.

Database Replay

When you’re planning a major system change, you spend a significant amount of time testing your system before cutting over to production. No matter how much prior testing you perform, there’s no guarantee that the production cutover to the new system will be trouble free, as you’ve never had the chance to “test” in the production environment itself. Oracle offers two brand-new tools called Database Replay and the SQL Performance Analyzer as part of the new Real Application Testing, or Total Recall, feature, to help you test your application performance before an actual system change, thus providing you great change management support. I discuss the Database Replay feature first in this section and explain the SQL Performance Analyzer feature toward the end of the chapter.

Database Replay offers you a way to test your system changes on a test system where you can simulate the actual production workload. You first capture the actual production workload over a representative period such as a peak period and replay it on a test system, thus re-creating your production system on a test system. The replay adheres to the original production concurrency and timing characteristics. The replay executes the RDBMS code in a way similar to how it was executed on the production system. The way it does this is by replaying all external client requests made to the RDBMS. The testing process will reveal whether there are any significant performance differences or errors between the before and after system change performance. Database Replay will also recommend fixes to the problems it encounters during the production workload replay. Database Replay offers a powerful, easy-to-implement system that lets you test system changes with confidence. If you’re moving from a single instance to an Oracle RAC environment, for example, you can test the database performance on a test system first before making the production cut over.

You can use Database Replay to test operating system and database upgrades, configuration changes such as a switch to a RAC system from a single-instance system and changes in the storage system. Database Replay captures all external requests such as SQL queries, PL/SQL code blocks, logins and logoffs, and DML/DDL statements. It ignores background jobs and requests made by
internal clients such as the Enterprise Manager. Database Replay ignores the following types of client requests:

- SQL*Loader direct path load of data
- Oracle Streams
- Data Pump Import and Export
- Advanced replication streams
- Non–PL/SQL-based Advanced Queuing (AQ)
- Flashback Database and Flashback Queries
- Distributed transactions and remote describe/commit operations
- Shared server

You can use either Enterprise Manager or APL/SQL APIs to run Database Replay. I show the manual steps in the following sections.

**Capturing the Production Workload**

Use the DBMS_WORKLOAD_CAPTURE package to capture the database workload. The database uses binary files called *capture files* to stop all captured external client requests to the database. The capture files hold client request–related information such as SQL statements and bind values. Here are the steps you must follow to capture the database workload:

1. Restart the database; although this isn't mandatory, it will minimize the errors and data divergence due to uncommitted or partial transactions at the time of the start of the data capture. You want to restart the database in restricted mode. Once you start the workload capture, the database automatically switches to an unrestricted mode of operation.

2. Define workload filters. You can use exclusion or inclusion filters to capture only a part of the actual workload and ignore the rest, as shown in the following example:

   ```sql
   SQL> begin
   dbms_workload_capture.add_filter (  
       fname      => 'user_salapati',  
       fattribute => 'USER',  
       fvalue     => 'salapati'
   end;
   /
   
   In this example, I restrict the workload capture to external calls made by the user SALAPATI.
   
   3. Set up a capture directory before starting the workload capture, making sure this directory is large enough to hold the workload. You can use a new or a preexisting directory.
4. Capture the production workload using a representative period. Execute the START_CAPTURE procedure to capture the workload:

```sql
begin
    dbms_workload_capture.start_capture (name => '2008Jan',
                                         dir => 'jan08',
                                         duration => 1200);
end;
```

Only the \texttt{DIR} parameter, which specifies the capture directory, is mandatory. If you omit the \texttt{DURATION} parameter, the workload capture will continue until you manually stop it as shown here:

```sql
begin
    dbms_workload_capture.finish_capture ();
end;
```

You can use the \texttt{DBA_WORKLOAD_CAPTURES} view to get information about the workload capture.

\textbf{Preprocessing the Workload}

You must preprocess the captured workload before you can replay it. Preprocessing is the step that converts the captured workload into replay files. As long as the database version is identical, you can preprocess the workload on the production system or a test system. Here’s how to preprocess the workload:

```sql
begin
    dbms_workload_replay.process_capture (capture_dir => '2008Jan');
end;
```

Preprocessing the workload produces the metadata for the captured workload and transforms the captured workload datafiles into replay streams called \textit{replay files}.

\textbf{Making the System Change}

At this point, make the system change you want to test on the test system. Once you make the change such as upgrading the database release, you can replay the captured production workload in the changed system to test the impact of the system change on performance, errors, and other related areas.

\textbf{Replaying the Captured Workload}

Create a test system that’s identical in every respect to the production system to run the captured production workload. You can duplicate the database on a test server to do this. Replaying the workload on the test system involves the following steps:

\textbf{Setting Up the Test System} Restore the production system on the test server, making sure it has the same application state as the production system. To avoid date-related errors, make sure the system time on the test system is set to the same time as prevailed at the start of the workload capture. Start the test system in the restricted mode to avoid errors.
Resolving External References

External references include objects such as database links, directory objects, and URLs. Before starting the replay, resolve all external references from the database. For example, all database links must be fully functional on the test system and point to the test system instead of the production database.

Setting Up the Replay Clients

Database Replay relies on a special application called the replay driver to send workload replay requests to the database. The replay driver is made up of replay clients that connect to the test system and simulate the external requests. So, the replay clients replace all external client interactions by sending requests that seem as if they came from the external clients themselves. You can have more than one replay client to share the replay workload, in which case it’s best to install the multiple replay clients on separate servers.

After ensuring that you’ve moved the workload files to the appropriate replay directory, start the replay client as shown here:

```
$ wrc [user/password[@server]] mode=[value] [keyword=[value]]
```

You can execute the wrc executable in different modes such as `REPLAY`, `CALIBRATE`, or `LIST_HOSTS`, by setting the `MODE` parameter. The parameter `KEYWORD` lets you specify execution options. You can find out the options available to you by typing in `wrc` at the command line:

```
$ wrc
```

```
Workload Replay Client: Release 11.1.0.6.0 - Production on Wed April 30 12:45:01 2007
Copyright (c) 1982, 2007, Oracle. All rights reserved.
FORMAT:
=======
wrc [user/password[@server]] [MODE=mode-value] KEYWORD=value
Example:
=======
wrc REPLAYDIR=.
wrc scott/tiger@myserver REPLAYDIR=.
wrc MODE=calibrate REPLAYDIR=./capture
The default privileged user is: SYSTEM

Mode:
=====
wrc can work in different modes to provide additional functionalities.
The default MODE is REPLAY.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLAY</td>
<td>Default mode that replays the workload in REPLAYDIR</td>
</tr>
<tr>
<td>CALIBRATE</td>
<td>Estimate the number of replay clients and CPUs needed to replay the workload in REPLAYDIR.</td>
</tr>
<tr>
<td>LIST_HOSTS</td>
<td>List all the hosts that participated in the capture or replay.</td>
</tr>
</tbody>
</table>
Options (listed by mode):

MODE=REPLAY (default)

---

**Keyword** | **Description**
---|---
USERID | username (Default: SYSTEM)
PASSWORD | password (Default: default password of SYSTEM)
SERVER | server connection identifier (Default: empty string)
REPLAYDIR | replay directory (Default: .)
WORKDIR | work directory (Default: .)
DEBUG | FILES, STDOUT, NONE (Default: NONE)
FILES | (write debug data to files at WORKDIR)
STDOUT | (print debug data to stdout)
BOTH | (print to both files and stdout)
NONE | (no debug data)
CONNECTION_OVERRIDE | TRUE, FALSE (Default: FALSE)
TRUE | All replay threads connect using SERVER, settings in DBA_WORKLOAD_CONNECTION_MAP will be ignored!
FALSE | Use settings from DBA_WORKLOAD_CONNECTION_MAP
SERIALIZE_CONNECTS | TRUE, FALSE (Default: FALSE)
TRUE | All the replay threads will connect to the database in a serial fashion one after another. This setting is recommended when the replay clients use the bequeath protocol to communicate to the database server.
FALSE | Replay threads will connect to the database in a concurrent fashion mimicking the original capture behavior.

MODE=CALIBRATE

MODE=LIST_HOSTS

If you have a large number of user sessions, you'll need multiple wrc clients on different hosts. Each replay thread from a replay client represents a single stream from the captured workload.

Although the default mode is **REPLAY**, it may be a good idea to first execute the wrc in **CALIBRATE** mode to estimate the number of replay clients and hosts you'll need to replay the workload. After you run the wrc in **CALIBRATE** mode, you can execute wrc in **REPLAY** mode, as shown here:

```bash
$ wrc system/<system_password> mode=replay replay_dir=./test_dir
```

### Initializing the Replay Data

Your next step is to initialize the workload data by executing the `INITIALIZE_REPLAY` procedure:

```sql
SQL> exec dbms_workload_replay.initialize_replay(replay_name => 'test_replay',replay_dir => 'test_dir');
```

Initializing the data loads the metadata into tables that are used by Database Replay.
Remapping External Connections

Before you start the workload replay, you must remap all external connections by executing the REMAP_CONNECTION procedure as shown here.

```sql
SQL> exec dbms_workload_replay.remap_connection (connection_id =>999,
   replay_connection => 'prod1:1521/testdb');
```

Remapping connections ensures users can connect to all the external databases. If you leave the `REPLAY_CONNECTION` parameter out, all replay sessions will automatically try connecting to the default host.

Setting Workload Options

The next step is the setting of various workload replay options. You can select from one of the following four options:

- **SYNCHRONIZATION**: The default for this parameter is `true`. This option preserves the commit order of the workload during the replay. With this parameter set to `true`, you can eliminate data divergence caused by not following the commit order among dependent transactions.

- **CONNECTION_TIME_SCALE**: This parameter lets you adjust the time between the beginning of the workload capture and the time when a session connects with the specified value. By adjusting this parameter, you can control the number of concurrent users during the replay.

- **THINK_TIME_SCALE**: This parameter enables you to calibrate the elapsed time between user calls in the same session. The smaller the value, the faster the client requests are sent to the database.

**Note** During a workload capture, elapsed time consists only of user time and user think time, whereas during a workload replay, elapsed time also includes the synchronization time component.

- **THINK_TIME_AUTO_CORRECT**: If you set this parameter to `true`, the database automatically corrects the think time specified by the `THINK_TIME_SCALE` parameter. For example, if the replay is moving slowly, the database reduces the value of the `THINK_TIME_SCALE` parameter. By default, this parameter is set to `false`.

Preparing the Workload for Replay

Before replaying the captured workload, prepare the workload by executing the PREPARE_REPLAY procedure:

```sql
SQL> dbms_workload_replay.prepare_replay (replay_name =>
   'replay1',replay_dir => 'test_dir',
   synchronization= FALSE);
```

If the workload consists mostly of independent transactions, it’s better to ignore the commit order by setting the `SYNCHRONIZATION` parameter to `false`, as shown in the example.

Starting the Workload Replay

Execute the START_REPLAY procedure to begin the workload replay, as shown here:

```sql
SQL> exec dbms_workload_replay.start_replay();
```
You can cancel the workload replay in midstream by doing this:

```
SQL> exec dbms_workload_replay.cancel_replay();
```

### Analyzing Workload Capture and Replay

Once the database completes the workload replay, you can analyze the replay report to find out about any errors and performance differences, as well as possible data anomalies between the original workload and the replayed workload. Here’s how you’d get a replay report by executing the `GET_REPLAY_INFO` function:

```sql
declare
cap_id     number;
rep_id     number;
rep_rpt    clob;
begin
cap_id  :=  dbms_workload_replay.get_replay_info (dir => 'mytestdir');
select max(id) into rep_id
from dba_workload_replays
where capture_id = cap_id;
rep_rpt  :=  dbms_workload_replay.report(
    replay_id  => rep_id,
    format     => dbms_workload_replay.type_text);
end;
/
```

The `REPLAY_REPORT` function produces the following text report:

**Error Data**

- **(% of total captured actions)**
  - New errors: 12.3%
  - Not reproduced old errors: 1.0%
  - Mutated errors: 2.0%

**Data Divergence**

- Percentage of row count diffs: 7.0%
- Average magnitude of difference (% of captured): 4.0%
- Percentage of diffs because of error (% of diffs): 20.0%
- Result checksums were generated for 10% of all actions (% of checksums)
- Percentage of failed checksums: 0.0%
- Percentage of failed checksums on same row count: 0.0%

**Replay Specific Performance Metrics**

- Total time deficit (-)/speed up (+): -32 min
Total time of synchronization:
44 min
Average elapsed time difference of calls:
0.1 sec
Total synchronization events:
367519064

You can also get a report in the HTML or XML format. You can query the DBA_WORKLOAD_REPLAYS view for the history of the replays performed by the database.

You must pay attention to any significant divergence between the captured workload and the replay of that workload. Any data divergence such as a smaller or larger result set in one of the two executions of the analyzer is a serious issue and merits further investigation. You must also check the performance deviation between the replay and the original workload. If the replay is taking longer to complete, you must consider this a serious issue. Any errors during the workload replay are good things for you to focus on as well. You can also use the ADDM to analyze the performance differences between the capture and the replay systems. Note that the presence of any of the following in the workload will exacerbate data or error divergence:

- Implicit session dependencies due to things such as the use of the DBMS_PIPE package
- Multiple commits within PL/SQl
- User locks
- Using nonrepeatable functions
- Any external interaction with URLs or database links

Use the following views to manage Database Replay:

- DBA_WORKLOAD_CAPTURES shows all workload captures you performed in a database.
- DBA_WORKLOAD_FILTERS shows all workload filters you defined in a database.
- DBA_WORKLOAD_REPLAYS shows all workload replays you performed in a database.
- DBA_WORKLOAD_REPLAY_DIVERGENCE helps monitor workload divergence.
- DBA_WORKLOAD_THREAD helps monitor the status of external replay clients.
- DBA_WORKLOAD_CONNECTION_MAP shows all connection strings used by workload replays.

Database Replay tests almost all of the database workload, unlike third-party tools, which can only simulate part of the real workload in an Oracle database. Compared to the third-party tools, the Database Replay tool is faster, and therefore you can compile the replay in a much shorter time period.

### SQL Performance Analyzer

The SQL Performance Analyzer, which together with the Database Replay feature forms the Total Replay feature offered by Oracle, enables you to test the impact of major system changes such as a database upgrade on SQL workload response time. The SQL Performance Analyzer analyzes and compares SQL performance before and after the system change and provides suggestions to improve any deterioration in performance. You can use the SQL Performance Analyzer to analyze potential changes in SQL performance following system changes such as database, application, operating system, or hardware upgrades; changes in initialization parameter settings; SQL tuning actions; and statistics gathering and schema changes.

The SQL Performance Analyzer lets you know, ahead of an actual database upgrade, which of your SQL statements is possibly going to regress in performance, so you can take care of them. You
can take steps to preserve SQL performance by using SQL Plan Management (SPM), which I discussed in Chapter 19. Or, you can employ the SQL Tuning Advisor to tune the potentially negatively impacted SQL statements.

You can use either the production system or a test system to run the SQL Performance Analyzer. Of course, if you run the analysis on a test system, you can avoid overhead on your production system. You can capture the SQL workload on a production system and run the analyzer on the test system. You can use either the Enterprise Manager or components of the DBMS_SQLPA package to use the SQL Performance Analyzer. You capture the SQL workload in the production system by using a SQL Tuning Set (STS). Once you capture SQL information in the STS, you can export the STS to the test system to provide the data for the SQL Performance Analyzer analysis. You can use any of the following as sources of the statements you load into the STS:

- AWR snapshots
- AWR baselines
- A cursor cache
- Another STS

The SQL Performance Analyzer executes SQL serially on the test server, ignoring the concurrency characteristics. It analyzes performance differences in the before- and after-change SQL workloads. The analyzer is integrated with the SQL Tuning Advisor for easy tuning of regressed statements.

I explain the workflow of a SQL Performance Analyzer analysis by showing how to predict changes in SQL performance following an upgrade to Oracle Database 11g from Oracle Database Release 10.2.

**Capturing the Production SQL Workload**

Select a representative period to capture the SQL workload from the production database. The workload that you collect consists of the SQL text and information pertaining to bind variable values and execution frequency. Following are the steps in capturing the production SQL workload:

**Creating the SQL Tuning Set**

Create the STS by executing the CREATE_SQLSET procedure as shown here:

```sql
SQL> exec dbms_sqltune.create_sqlset(sqlset_name => 'test_set',
                                    description  => '11g upgrade workload);
```

The next step is to load the empty STS you created in this step.

**Loading the SQL Tuning Set**

Execute the DBMS_SQLTUNE SELECT_CURSOR_CACHE procedure to load the empty STS.

```sql
declare
  mycur dbms_sqltune.sqlset_cursor;
begin
  open  mycur for
    select value (P)
    from table (dbms_sqltune.select_cursor_cache(
      'parsing_schema_name <> ''SYS'' AND elapsed_time >
      2500000',null,null,null,null,1,null,
      'ALL')) P;
```
The database incrementally loads the STS from the cursor cache over a period of time.

**Transporting the SQL Tuning Set**

Create a staging table first, in order to transport the STS to the test system.

```sql
SQL> exec dbms_sqltune.create_stgtb_sqlset (table_name => 'stagetab');
```

Export the STS to the staging table using the `PACK_STGTAB_SQLSET` procedure.

```sql
SQL> exec dbms_sqltune.pack_stgtab_sqlset(sqlset_name => 'test_sts',
staging_table_name => 'stagetab');
```

In the next step, you'll import the staging table you created into the test system. Use Data Pump to import the staging table stagetab into the test system. Execute the `UNPACK_STGTAB_SQLSET` procedure to import the STS into the test database.

```sql
SQL> exec dbms_sqltune.unpack_stgtab_sqlset (sqlset_name = '%',
replace => true, staging_table_name => ('stagetab');
```

Next, you'll create the SQL Performance Analyzer task.

**Creating the SQL Performance Analyzer Task**

Use the `CREATE_ANALYSIS_TASK` procedure to create a new SQL Performance Analyzer task, as shown here:

```sql
SQL> exec dbms_sqlpa.create_analysis_task(sqlset_name => 'sts1',
task_name => 'spa_task1');
```

The procedure you execute here enables you to create an analyzer job to analyze one or more SQL statements.

**Analyzing the Prechange SQL Workload**

The before-change SQL workload analysis analyzes the performance of SQL statements in an Oracle 10.2 environment. Make sure that the `OPTIMIZER_FEATURES_ENABLE` initialization parameter is correctly set.

```sql
optimizer_features_enable=10.2.0
```

Execute the `EXECUTE_ANALYSIS_TASK` procedure to analyze the preupgrade performance of the SQL workload, as shown here:

```sql
SQL> exec dbms_sqlpa.execute_analysis_task (task_name =>
'spa_task1',
execution_type => 'test_execute',
execution_name= 'before_change');
```
Note that the value of the EXECUTION_TYPE parameter is set to TEST_EXECUTE. This value ensures that the database executes all SQL statements in the workload and generates both the execution plans and execution statistics such as disk reads. You can assign two other values for the EXECUTION_TYPE parameter. The COMPARE_PERFORMANCE parameter will compare the performance based on a comparison of two different analyses. The value EXPLAIN_PLAN will generate SQL plans without executing them.

Get a report of the preupgrade SQL performance by executing the REPORT_ANALYSIS_TASK function:

```
SQL> select dbms_sqlpa.report_analysis_task (task_name =>
'spa_task1',
type => 'text',
section=> 'summary') from dual;
```

You now have a performance baseline with which to compare the after-upgrade SQL performance.

### Analyzing the After-Upgrade SQL Workload

Set the value of the OPTIMIZER_FEATURES_ENABLE parameter to match the Oracle Database 11g release:

```
optimizer_features_enable=11.1
```

Execute the SPA task again, this time to analyze the after-upgrade performance of the SQL workload.

```
SQL> exec dbms_sqlpa.execute_analysis_task (task_name  => 'spa_task2',
execution_type => 'test_execute',
execution_name => 'after_change')
```

Get a report of the after-upgrade performance as shown here:

```
SQL> select dbms_sqlpa.report_analysis_task (task_name =>   'spa_task2',
type => 'text', section=> 'summary') from dual;
```

### Comparing the SQL Performance

Execute the EXECUTE_ANALYSIS_TASK procedure once again, but with the value COMPAREPERFORMANCE for the EXECUTION_TYPE parameter, in order to analyze and compare the SQL performance data before and after the database upgrade.

```
SQL> exec dbms_sqltune.execute_analysis_task (task_name =>
'spa_task3',
execution_type => 'compare performance',
execution_params =>
dbms_advisor.arglist('execution_name1','before_change',
execution_name2','after_change''comparision_metric',
'disk_reads',)
```

In addition to DISK READS, you can specify metrics such as ELAPSED_TIME, PARSE_TIME, or BUFFER GETS when comparing the performance.

### Generating the Analysis Report

Execute the REPORT_ANALYSIS_TASK function to get a report of the performance comparison:

```
var report clob;
exec :report := dbms_sqlpa.report_analysis_task('spa_task1',
'text',
'typical','summary');
```
set long 100000 longchunksize 100000 linesize 120

print :report

During the compare and analysis phase, you can do the following:

- Calculate the impact of the change on specific SQL statements.
- Calculate the impact of the change on the SQL workload as a whole.
- Assign weights to important SQL statements in the workload.
- Detect performance regression and improvements.
- Detect changes in the execution plans of the SQL statements.
- Recommend the running of the SQL Tuning Advisor to tune regressed SQL statements.

You can use the following views when working with the SQL Performance Analyzer:

- `DBA_ADVISOR_TASKS` shows details about the analysis task.
- `DBA_ADVISOR_FINDINGS` shows analysis findings, which are classified as performance regression, symptoms, informative messages, and errors.
- `DBA_ADVISOR_EXECUTIONS` shows metadata information for task executions.
- `DBA_ADVISOR_SQLPLANS` shows a list of SQL execution plans.
- `DBA_ADVISOR_SQLSTATS` shows a list of SQL compilation and execution statistics.

### Analyzing the Performance Report

The SQL Performance Analyzer contains both a result summary and a result details section. The former shows quickly whether the database upgrade in our example will result in any performance deterioration or improvement. The advisor also provides recommendations to avoid any potential performance deterioration.

Since the SQL Performance Analyzer is an integral part of the Oracle database, it can take advantage of tools such as the SQL Tuning Advisor as well as features such as SQL Plan Management to fine-tune database performance.
I'm sure most of you are already familiar with SQL to some extent. However, I present in this appendix a quick introduction to Oracle Database 11g SQL and its programmatic cousin, PL/SQL, as a starting point for those new to programming Oracle databases. My goal here is simply to present a short summary of the classic DML and DDL commands and to discuss the newer SQL and PL/SQL concepts in greater detail.

Your need to know SQL or PL/SQL depends somewhat on the type of DBA you are—a production support DBA won’t need to know as much about Oracle programming as a DBA assisting in development efforts. It’s becoming increasingly important, however, for DBAs to learn a number of advanced SQL and PL/SQL concepts, including the new Java and XML-based technologies. The reason is simple: even when you aren’t developing applications yourself, you’re going to be assisting people who are doing so, and it helps to know what they’re doing.

This appendix aims to summarize some of the most important Oracle Database 11g SQL and PL/SQL features so you and the developers you work with can take advantage of them. Oracle SQL and PL/SQL represent an enormously broad topic, so this appendix lightly covers several important topics without attempting any detailed explanation due to space considerations. Please refer to the Oracle manuals Application Developer’s Guide—Fundamentals and PL/SQL User’s Guide and Reference for a comprehensive introduction to SQL and PL/SQL.

The Oracle Database 11g Sample Schemas

The examples in this appendix use the demo schemas provided by Oracle as part of the Oracle Database 11g server software. The demo data is for a fictitious company and contains the following five schemas:

- **HR** is the human resources division, which contains information on employees. It is the most commonly used schema, with its familiar employees and dept tables. The schema uses scalar data types and simple tables with basic constraints.
- **OE** is the order entry department, which contains inventory and sales data. This schema covers a simple order-entry system and includes regular relational objects as well as object-relational objects. Because the OE schema contains synonyms for HR tables, you can query HR’s objects from the OE schema.
- **PM** is the product media department, which covers content management. You can use this schema if you’re exploring Oracle’s Multimedia option. The tables in the PM schema contain audio and video tracks, images, and documents.
• *IX* is the information exchange department in charge of shipping using various B2B applications.

• *SH* is the sales history department in charge of sales data. It is the largest sample schema, and you can use it for testing examples with large amounts of data. The schema contains partitioned tables, an external table, and Online Analytical Processing (OLAP) features. The SALES and COSTS tables contain 750,000 rows and 250,000 rows, respectively, as compared to 107 rows in the employees table from the HR schema.

In order to install the SH schema, you must have the partitioning option installed in your Oracle database; this option lets you use table and index partitioning in your database. Ideally, you should install the Oracle demo schemas in a test database where you can safely practice the parts of SQL you aren’t familiar with. The Oracle Sample Schemas documentation manual provides detailed information about the sample schemas.

If you’ve created a starter database using the Database Configuration Assistant (DBCA) as part of your Oracle software installation (the Basic Installation option), it will have automatically created the sample schemas in the new starter database.

If you’ve chosen not to create the starter database (by selecting a Software Only installation option), you can run the DBCA to install the sample schemas. Choose the Sample Schemas option when you use the DBCA to create the sample schemas in an existing database. By default, all the sample schema accounts are locked, and you must use the `ALTER USER . . . ACCOUNT UNLOCK` statement to unlock them.

If you want to create the sample schemas in a database without using the DBCA, you can run Oracle-provided scripts to install the sample schemas.

### Oracle Data Types

Data in an Oracle database is organized in rows and columns inside tables. The individual columns are defined with properties that limit the values and format of the column contents. Let’s review the most important Oracle built-in data types before we look at Oracle SQL statements.

### Character Data Types

The `CHAR` data type is used for fixed-length character literals:

```sql
SEX    CHAR(1)
```

The `VARCHAR2` data type is used to represent variable-length character literals:

```sql
CITY   VARCHAR2 (20)
```

The `CLOB` data type is used to hold large character strings and the `BLOB` and `BFILE` data types are used to store large amounts of binary data.

### Numeric Data Types

There are two important SQL data types used to store numeric data:

• The `NUMBER` data type is used to store real numbers, either in a fixed-point or floating-point format.

• The `BINARY_FLOAT` and `BINARY_DOUBLE` data types store data in a floating-point format.
Date and Time Data Types

There are a couple of special data types that let you handle date and time values:

- The DATE data type stores the date and time (such as year, month, day, hours, minutes, and seconds).
- The TIMESTAMP data type stores time values that are precise to fractional seconds.

Conversion Functions

Oracle offers several conversion functions that let you convert data from one format to another. The most common of these functions are the TO_CHAR, TO_NUMBER, TO_DATE, and TO_TIMESTAMP functions. The TO_CHAR function converts a floating number to a string, and the TO_NUMBER function converts a floating number or a string to a number. The TO_DATE function converts character data to a DATE data type. Here are some examples:

```sql
SQL> SELECT TO_CHAR(TO_DATE('20-JUL-08', 'DD-MON-RR') ,'YYYY') "Year" FROM DUAL;

Year
----------------------
2008

SQL>

SQL> SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY')
       FROM DUAL;

TO_CHAR(SYSDATE
-----------
20-JUL-2008

SQL>
```

SQL

In Chapter 7 you saw how Oracle SQL statements include DDL, DML, and other types of statements. Let’s begin with a review of the basic SQL statements.

The SELECT Statement

The SELECT statement is the most common SQL statement (it is also called a projection). A SELECT statement retrieves all or some of the data in a table, based on the criteria that you specify.

The most basic SELECT statement is one that retrieves all the data in the table:

```sql
SQL> SELECT * FROM employees;
```

To retrieve only certain columns from a table, you specify the column names after the SELECT keyword, as shown in the following example:

```sql
SQL> SELECT first_name, last_name, hiredate FROM employees;
```

If you want only the first ten rows of a table, you can use the following statement:

```sql
SQL> SELECT * FROM employees WHERE rownum <11;
```
If you want just a count of all the rows in the table, you can use the following statement:

```
SQL> SELECT COUNT(*) FROM employees;
```

If a table has duplicate data, you can use the DISTINCT clause to eliminate the duplicate values, as shown here:

```
SQL> SELECT DISTINCT username FROM V$SESSION;
```

The optional WHERE clause in a SELECT statement uses conditions to help you specify that only certain rows be returned. Table A-1 lists some of the common conditions you can use in a WHERE clause.

**Table A-1. Common Conditions Used in WHERE Clauses**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;+</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>+=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt; or !</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Here are some examples of using the WHERE clause:

```
SQL> SELECT employee_id WHERE salary = 50000;
SQL> SELECT employee_id WHERE salary < 50000;
SQL> SELECT employee_id WHERE salary > 50000;
SQL> SELECT employee_id WHERE salary <= 50000;
SQL> SELECT employee_id WHERE salary >= 50000;
SQL> SELECT employee_id WHERE salary ! 50000;
```

**The LIKE Condition**

The LIKE condition uses pattern matching to restrict rows in a SELECT statement. Here’s an example:

```
SQL> SELECT employee_id, last_name FROM employees
    2* WHERE last_name LIKE 'Fa%';
EMPLOYEE_ID | LAST_NAME
----------- | -------
    109      | Faviet
     202      | Fay
```

The pattern that you want the WHERE clause to match should be enclosed in single quotes (’ ’). In the preceding example, the percent sign (%) indicates that the letters Fa can be followed by any character string. Thus, the percent sign acts as a wildcard for one or more characters, performing the same job as the asterisk (*) in many operating systems. Note that a single underscore character (_) acts as a wildcard for one and only one character.
The INSERT Statement

The INSERT statement enables you to add new data to a table, including duplicate data if there are no unique requirements enforced by a primary key or an index. The general form of the INSERT statement is as follows:

```sql
INSERT INTO <table> [(<column i, . . . , column j>)]
VALUES (<value i, . . . ,value j>);
```

Here is an example of the `insert` command:

```sql
SQL> INSERT INTO employees(
2   employee_id,last_name,email,hire_date,job_id)
3   VALUES
4   (56789,'alapati','salapati@netbsa.org', sysdate,98765);
1 row created.
SQL>
```

In the preceding list, the column names were specified because only some columns were being populated in the row being inserted. The rest of them are left blank, which is okay, provided the column isn’t defined as a “not null” column.

If you’re inserting values for all the columns of a table, you can use the simpler `INSERT` statement shown here:

```sql
SQL> INSERT INTO department
VALUES
(34567, 'payroll', 'headquarters', 'dallas');
1 row created.
SQL>
```

If you want to insert all the columns of a table into another table, you can use the following `INSERT` statement:

```sql
SQL> INSERT INTO b SELECT * FROM a
WHERE city='DALLAS';
```

If table b doesn’t exist, you can use the `CREATE TABLE table_name AS SELECT * FROM (CTAS)` statement, as shown here:

```sql
SQL> CREATE table b as SELECT * FROM a;
```

The DELETE Statement

You use the DELETE statement to remove rows from a table. The DELETE statement has the following structure:

```sql
DELETE FROM <table> [WHERE ,condition];
```

For example, if you want to delete employee Fay’s row from the employees table, you would use the following DELETE statement:

```sql
SQL> DELETE FROM employees
WHERE last_name='Fay';
1 row deleted.
```

If you don’t have a limiting `WHERE` condition, the DELETE statement will result in the removal of all the rows in the table, as shown here:

```sql
SQL> DELETE FROM X;
```
You can also remove all rows in a table using the TRUNCATE command, but you can’t undo or roll back the TRUNCATE command’s effects. You can undo a delete by using the ROLLBACK statement:

SQL> ROLLBACK;

The UPDATE Statement

The UPDATE statement changes the value (or values) of one or more columns of a row (or rows) in a table. The expression to which a column is being set or modified can be a constant, arithmetic, or string operation, or the product of a SELECT statement.

The general structure of the UPDATE statement is as follows (note that the elements in square brackets are optional):

UPDATE <table>
SET <column i> = <expression i>, . . . , <column j> = <expression j>
[WHERE <condition> ];

If you want to change or modify a column’s values for all the rows in the table, you use an UPDATE statement without a WHERE condition:

SQL> UPDATE persons SET salary=salary*0.10;

If you want to modify only some rows, you need to use the WHERE clause in your UPDATE statement:

SQL> UPDATE persons SET salary = salary * 0.10
WHERE review_grade > 5;

Filtering Data

The WHERE clause in a SELECT, INSERT, DELETE, or UPDATE statement lets you filter data. That is, you can restrict the number of rows on which you want to perform a SQL operation. Here’s a simple example:

SQL> INSERT INTO a
SELECT * FROM b
WHERE city='DALLAS';

Sorting the Results of a Query

Frequently, you’ll have to sort the results of a query in some order. The ORDER BY clause enables you to sort the data based on the value of one or more columns. You can choose the sorting order (ascending or descending) and you can choose to sort by column aliases. You can also sort by multiple columns. Here’s an example:

SQL> SELECT employee_id, salary FROM employees
ORDER BY salary;

Changing the Sorting Order

By default, an ORDER BY clause sorts in ascending order. If you want to sort in descending order, you need to specify the DESC keyword:

SQL> SELECT employee_id, salary FROM employees
ORDER BY salary desc;
Sorting by Multiple Columns

You can sort results based on the values of more than one column. The following query sorts on the basis of two columns, salary and dept:

```sql
SQL> SELECT employee_id, salary FROM employees
               ORDER BY salary, dept;
```

Operators

SQL provides you with a number of operators to perform various tasks, such as comparing column values and performing logical operations. The following sections outline the important SQL operators: comparison operators, logical operators, and set operators.

Comparison Operators

Comparison operators compare a certain column value with several other column values. These are the main comparison operators:

- **BETWEEN**: Tests whether a value is between a pair of values
- **IN**: Tests whether a value is in a list of values
- **LIKE**: Tests whether a value follows a certain pattern, as shown here:

```sql
SQL> SELECT employee_id from employees
               WHERE dept LIKE 'FIN%';
```

Logical Operators

The logical operators, also called Boolean operators, logically compare two or more values. The main logical operators are AND, OR, NOT, GE (greater than or equal to), and LE (less than or equal to). Here's an example that illustrates the use of some of the logical operators:

```sql
SQL> SELECT last_name, city
               WHERE salary GT 100000 and LE 200000;
```

When there are multiple operators within a single statement, you need rules of precedence. Oracle always evaluates arithmetical operations such as multiplication, division, addition, and subtraction before it evaluates conditions. The following is the order of precedence of operators in Oracle, with the most important first:

```
=, !=, <, >, <=, >=
IS NULL, LIKE, BETWEEN, IN, EXISTS
NOT
AND
OR
```

The Set Operators

Sometimes your query may need to combine results from more than one SQL statement. In other words, you need to write a compound query. Set operators facilitate compound SQL queries. Here are the important Oracle set operators:
• **UNION**: The **UNION** operator combines the results of more than one **SELECT** statement after removing any duplicate rows. Oracle will sort the resulting set of data. Here’s an example:

```sql
SQL> SELECT emp_id FROM old_employees
UNION
SELECT emp_id FROM new_employees;
```

• **UNION ALL**: The **UNION ALL** operator is similar to **UNION**, but it doesn’t remove the duplicate rows. Oracle doesn’t sort the result set in this case, unlike the **UNION** operation.

• **INTERSECTION**: The **INTERSECTION** operator gets you the common values in two or more result sets derived from separate **SELECT** statements. The result set is distinct and sorted.

• **MINUS**: The **MINUS** operator returns the rows returned by the first query that aren’t in the second query’s results. The result set is distinct and sorted.

### SQL Functions

Oracle functions manipulate data items and return a result, and built-in Oracle functions help you perform many transformations quickly, without your having to do any coding. In addition, you can build your own functions. Functions can be divided into several groups: single-row functions, aggregate functions, number and date functions, general and conditional functions, and analytical functions.

#### Single-Row Functions

**Single-row functions** are typically used to perform tasks such as converting a lowercase word to uppercase or vice versa, or replacing a portion of text in a row. Here are the main single-row functions used in Oracle:

- **CONCAT**: The **CONCAT** function concatenates or puts together two or more character strings into one string.
- **LENGTH**: The **LENGTH** function gives you the length of a character string.
- **LOWER**: The **LOWER** function transforms uppercase letters into lowercase, as shown in the following example:

```sql
SQL> SELECT LOWER('SHANNON ALAPATI') from dual;

LOWER('SHANNONALAPATI')
-----------------------
shannon alapati
SQL>
```

- **SUBSTR**: The **SUBSTR** function returns part of a string.
- **INSTR**: The **INSTR** function returns a number indicating where in a string a certain string value starts.
- **LPAD**: The **LPAD** function returns a string after padding it for a specified length on the left.
- **RPAD**: The **RPAD** function pads a string on the right side.
- **TRIM**: The **TRIM** function trims a character string.
- **REPLACE**: The **REPLACE** function replaces every occurrence of a specified string with a specified replacement string.
Aggregate Functions

You can use *aggregate functions* to compute things such as averages and totals of a selected column in a query. Here are the important aggregate functions:

- **MIN**: The MIN function returns the smallest value. Here's an example:
  ```sql
  SELECT MIN(join_date) FROM employees;
  ```
- **MAX**: The MAX function returns the largest value.
- **AVG**: The AVG function computes the average value of a column.
- **SUM**: The SUM function computes the sum of a column:
  ```sql
  SQL> SELECT SUM(bytes) FROM dba_free_space;
  ```
- **COUNT**: The COUNT function returns the total number of columns.
- **COUNT(*)**: The COUNT(*) function returns the number of rows in a table.

Number and Date Functions

Oracle includes several *number functions*, which accept numeric input and return numeric values. The *date functions* help you format dates and times in different ways. Here are some of the important number and date functions:

- **ROUND**: This function returns a number rounded to the specified number of places to the right of the decimal point.
- **TRUNC**: This function returns the result of a date truncated in the specified format.
- **SYSDATE**: This commonly used function returns the current date and time:
  ```sql
  SQL> SELECT sysdate FROM dual;
  ```

  ```
  SYSDATE
  -------------------
  07/AUG/2008
  SQL>
  ```
- **TO_TIMESTAMP**: This function converts a CHAR or VARCHAR(2) data type to a timestamp data type.
- **TO_DATE**: You can use this function to change the current date format. The standard date format in Oracle is DD-MMM-YYYY, as shown in the following example:

  ```
  07-AUG-2008
  ```

- The TO_DATE function accepts a character string that contains valid data and converts it into the default Oracle date format. It can also change the date format, as shown here:
  ```sql
  SQL> SELECT TO_DATE('August 20,2008', 'MonthDD,YYYY') FROM dual;
  ```

  ```
  TO_DATE('AUGUST20,2008'
  -----------------------
  08/20/2008
  SQL>
  ```
• **TO_CHAR**: This function converts a date into a character string, as shown in the following example:

```
SQL> SELECT SYSDATE FROM dual;

SYSDATE
--------
04-AUG-2008
SQL>
```

```
SQL> SELECT TO_CHAR(SYSDATE, 'DAY, DDTH MONTH YYYY') FROM DUAL;

TO_CHAR(SYSDATE, 'DAY, DDTH MONTH YYYY')
----------------------------------------
THURSDAY, 04TH AUGUST 2008
```

• **TO_NUMBER**: This function converts a character string to a number format:

```
SQL> UPDATE employees SET salary = salary +
  TO_NUMBER('100.00', '9G999D99')
  WHERE last_name = 'Alapati';
```

### General and Conditional Functions

Oracle provides some very powerful general and conditional functions that enable you to extend the power of simple SQL statements into something similar to a traditional programming language construct. The conditional functions help you decide among several choices. Here are the important general and conditional Oracle functions:

• **NVL**: The NVL function replaces the value in a table column with the value after the comma if the column is null. Thus, the NVL function takes care of column values if the column values are null and converts them to non-null values:

```
SQL> SELECT last_name, title,
  salary * NVL (commission_pct,0)/100 COMM
FROM employees;
```

• **COALESCE**: This function is similar to NVL, but it returns the first non-null value in the list:

```
SQL> COALESCE(region1, region2, region3, region4)
```

• **DECODE**: This function is used to incorporate basic if-then functionality into SQL code. The following example assigns a party name to all the voters in the table based on the value in the affiliation column. If there is no value under the affiliation column, the voter is listed as an independent:

```
SQL> SELECT DECODE(affiliation, 'D', 'Democrat',
  'R', 'Republican', 'Independent') FROM voters;
```

• **CASE**: This function provides the same functionality as the DECODE function, but in a much more intuitive and elegant way. Here’s a simple example of using the CASE statement, which helps you incorporate if-then logic into your code:

```
SQL> SELECT ename,
  (CASE deptno
  WHEN 10 THEN 'Accounting'
  WHEN 20 THEN 'Research'
  WHEN 30 THEN 'Sales'
```
WHEN 40 THEN 'Operations'
ELSE 'Unknown'
END)
department
FROM employees;

Analytical Functions

Oracle’s SQL analytical functions are powerful tools for business intelligence applications. Oracle claims a potential improvement of 200 to 500 percent in query performance with the use of the SQL analytical functions. The purpose behind using analytical functions is to perform complex summary computations without using a lot of code. Here are the main SQL analytical functions of the Oracle database:

- **Ranking functions**: These enable you to rank items in a data set according to some criteria. Oracle has several types of ranking functions, including RANK, DENSE_RANK, CUME_DIST, PERCENT_RANK, and NTILE. Listing A-1 shows a simple example of how a ranking function can help you rank some sales data.

Listing A-1. An Example of a Ranking Function

```sql
SQL> SELECT sales_type,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES,
       RANK() OVER (ORDER BY SUM(amount_sold) ) AS original_rank,
       RANK() OVER (ORDER BY SUM(amount_sold)
       DESC NULLS LAST) AS derived_rank
FROM sales, products, customers, time_frame, sales_types
WHERE sales.prod_id=products.prod_id AND
  sales.cust_id=customers.cust_id AND
  sales.time_id=time_frame.time_id AND
  sales.sales_type_id=sales_types.sales_type_id AND
  timeframe.calendar_month_desc IN ('2008-07', '2008-08')
  AND country_id='INDIA'
GROUP BY sales_type;
```

<table>
<thead>
<tr>
<th>SALES_TYPE</th>
<th>SALES</th>
<th>ORIGINAL_RANK</th>
<th>DERIVED_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Sales</td>
<td>5,744,263</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Internet</td>
<td>3,625,993</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Catalog</td>
<td>1,858,386</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Partners</td>
<td>1,500,213</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Tele Sales</td>
<td>604,656</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Moving-window aggregates**: These functions provide cumulative sums and moving averages.
- **Period-over-period comparisons**: These functions let you compare two periods (for example, “How does the first quarter of 2008 compare with the first quarter of 2006 in terms of percentage growth?”).
- **Ratio-to-report comparisons**: These make it possible to compare ratios (for example, “What is August’s enrollment as a percentage of the entire year’s enrollment?”).
- **Statistical functions**: These functions calculate correlations and regression functions so you can see cause-and-effect relationships among data.
- **Inverse percentiles**: These help you find the data corresponding to a percentile value (for example, “Get me the names of the salespeople who correspond to the median sales value.”).
- **Hypothetical ranks and distributions**: These help you figure out how a new value for a column fits into existing data in terms of its rank and distribution.

- **Histograms**: These functions return the number of the histogram data appropriate for each row in a table.

- **First/last aggregates**: These functions are appropriate when you are using the `GROUP BY` clause to sort data into groups. Aggregate functions let you specify the sort order for the groups.

### Hierarchical Retrieval of Data

If a table contains hierarchical data (data that can be grouped into levels, with the parent data at higher levels and child data at lower levels), you can use Oracle's hierarchical queries. Hierarchical queries typically use the following structure:

- The `START WITH` clause denotes the root row or rows for the hierarchical relationship.
- The `CONNECT BY` clause specifies the relationship between parent and child rows, with the prior operator always pointing out the parent row.

Listing A-2 shows a hierarchical relationship between the employees and manager columns. The `CONNECT BY` clause describes the relationship. The `START WITH` clause specifies where the statement should start tracing the hierarchy.

**Listing A-2. A Hierarchical Relationship Between Data**

```sql
SQL> SELECT employee_id, last_name, manager_id
    FROM employees
    START WITH manager_id = 100
    CONNECT BY PRIOR employee_id = manager_id;

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Reddy</td>
<td>100</td>
</tr>
<tr>
<td>108</td>
<td>Greenberg</td>
<td>101</td>
</tr>
<tr>
<td>109</td>
<td>Faviet</td>
<td>108</td>
</tr>
<tr>
<td>110</td>
<td>Colon</td>
<td>108</td>
</tr>
<tr>
<td>111</td>
<td>Chowdhary</td>
<td>108</td>
</tr>
<tr>
<td>112</td>
<td>Urman</td>
<td>108</td>
</tr>
<tr>
<td>113</td>
<td>Singh</td>
<td>108</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>101</td>
</tr>
</tbody>
</table>
```

SQL>

### Selecting Data from Multiple Tables

So far, we've mostly looked at how to perform various DML operations on single tables, including using SQL functions and expressions. However, in real life, you'll mostly deal with query output retrieved from several tables or views. When you need to retrieve data from several tables, you need to **join** the tables. A join is a query that lets you combine data from tables, views, and materialized views. Note that a table can be joined to other tables or to itself.

The Cartesian product or Cartesian join is simply a join of two tables without a selective `WHERE` clause. Therefore, the query output will consist of all rows from both tables. Here's an example of a Cartesian join:

```sql
SQL> SELECT * FROM employees, dept;
```
Cartesian products of two large tables are almost always the result of a mistaken SQL query that omits the join condition. By using a join condition when you’re combining data from two or more tables, you can limit the number of rows returned. A join condition can be used in the WHERE clause or the FROM clause, and it limits the data returned by selecting only data that satisfies the condition stipulated by the join condition.

Here’s an example of a join statement that uses a join condition:

```sql
SQL> SELECT * FROM employees, dept
    WHERE dept='HR';
```

### Types of Oracle Joins

Oracle offers various types of joins based on the way you combine rows from two or more tables or views. The next sections discuss the most commonly used types of Oracle joins.

#### Equi-Join

With an equi-join, two or more tables are joined based on an equality condition between two columns. In other words, the same column has the same value in all the tables that are being joined. Here’s an example:

```sql
SQL> SELECT e.last_name, d.dept
    FROM emp e, dept d WHERE e.emp_id = d.emp_id;
```

You can also use the following new syntax for the preceding join statement:

```sql
SQL> SELECT e.last_name, d.dept
    FROM emp e JOIN dept d
    USING (emp_id);
```

If you want to join multiple columns, you can do so by using a comma-delimited list of column names, as in USING (dept_id, emp_name).

#### Natural Join

A natural join is an equi-join where you don’t specify any columns to be matched for the join. Oracle will automatically determine the columns to be joined, based on the matching columns in the two tables. Here’s an example:

```sql
SQL> SELECT e.last_name, d.dept
    FROM emp e NATURAL JOIN dept d;
```

In the preceding example, the join is based on identical values for the last_name column in both the emp and dept tables.

#### Self Join

A self join is a join of a table to itself through the use of table aliases. In the following example, the employees table is joined to itself using an alias. The query deletes duplicate rows in the employees table.

```sql
SQL> DELETE FROM employees X WHERE ROWID >
    2 (select MIN(rowid) FROM employees Y
    3 where X.key_values = Y.key_values);
```
Inner Join

An inner join, also known as a simple join, returns all rows that satisfy the join condition. The traditional Oracle inner join syntax used the WHERE clause to specify how the tables were to be joined. Here’s an example:

```
SQL> SELECT e.flast_name, d.dept
           FROM emp e, dept d WHERE e.emp_id = d.emp_id;
```

The newer Oracle inner joins (or simply joins) specify join criteria with the new ON or USING clause. Here’s a simple example:

```
SQL> SELECT DISTINCT NVL(dname, 'No Dept'),
                   COUNT(empno) nbr_emps
           FROM emp JOIN DEPT
           ON emp.deptno = dept.deptno
           WHERE emp.job IN ('MANAGER', 'SALESMAN', 'ANALYST')
           GROUP BY dname;
```

Outer Join

An outer join returns all rows that satisfy the join condition, plus some or all of the rows from the table that doesn’t have matching rows that meet the join condition. There are three types of outer joins: left outer join, right outer join, and full outer join. Usually, the word “outer” is omitted from the full outer join statement.

Oracle provides the outer join operator, wherein you use a plus sign (+) to indicate missing values in one table, but it recommends the use of the newer ISO/ANSI join syntax. Here’s a typical query using the full outer join:

```
SQL> SELECT DISTINCT NVL(dept_name, 'No Dept') deptname,
                   COUNT(empno) nbr_emps
           FROM emp FULL JOIN dept
           ON dept.deptno = emp.deptno
           GROUP BY dname;
```

Grouping Operations

Oracle provides the GROUP BY clause so you can group the results of a query according to various criteria. The GROUP BY clause enables you to consider a column value for all the rows in the table fulfilling the SELECT condition.

A GROUP BY clause commonly uses aggregate functions to summarize each group defined by the GROUP BY clause. The data is sorted on the GROUP BY columns, and the aggregates are calculated. Here’s an example:

```
SQL> SELECT department_id, MAX(salary)
            FROM employees
       2 GROUP BY department_id;
```

```
DEPARTMENT_ID   MAX(SALARY)
----------------  -----------
    10        4400
    20       13000
    30       11000
    40        6500
    50        8200
5 rows selected.
```

SQL>
Oracle also allows you to nest group functions. The following query gets you the minimum average budget for all departments (the AVG function is nested inside the MIN function here):

```
SQL> SELECT MIN(AVG(budget))
        FROM dept_budgets
        GROUP BY dept_no;
```

**The GROUP BY Clause with a ROLLUP Operator**

You’ve seen how you can derive subtotals with the help of the `GROUP BY` clause. The `GROUP BY` clause with a `ROLLUP` operator gives you subtotals and total values. You can thus build subtotal aggregates at any level. In other words, the `ROLLUP` operator gets you the aggregates at each group by level. The subtotal rows and the grand total row are called the superaggregate rows.

Listing A-3 shows an example of using the `ROLLUP` operator.

**Listing A-3. A GROUP BY Clause with a ROLLUP Operator**

```
SQL> SELECT Year,Country,SUM(Sales) AS Sales
        FROM Company_Sales
        GROUP BY ROLLUP (Year,Country);
```

<table>
<thead>
<tr>
<th>YEAR</th>
<th>COUNTRY</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>France</td>
<td>3990</td>
</tr>
<tr>
<td>1997</td>
<td>USA</td>
<td>13090</td>
</tr>
<tr>
<td>1997</td>
<td></td>
<td>17080</td>
</tr>
<tr>
<td>1998</td>
<td>France</td>
<td>4310</td>
</tr>
<tr>
<td>1998</td>
<td>USA</td>
<td>13900</td>
</tr>
<tr>
<td>1998</td>
<td></td>
<td>18210</td>
</tr>
<tr>
<td>1999</td>
<td>France</td>
<td>4570</td>
</tr>
<tr>
<td>1999</td>
<td>USA</td>
<td>14670</td>
</tr>
<tr>
<td>1999</td>
<td></td>
<td>19240</td>
</tr>
<tr>
<td></td>
<td></td>
<td>54530 /*This is the grand total */</td>
</tr>
</tbody>
</table>

SQL>

**The GROUP BY Clause with a CUBE Operator**

You can consider the `CUBE` operator to be an extension of the `ROLLUP` operator, as it helps extend the standard Oracle `GROUP BY` clause. The `CUBE` operator computes all possible combinations of subtotals in a `GROUP BY` operation. In the previous example, the `ROLLUP` operator gave you yearly subtotals. Using the `CUBE` operator, you can get countrywide totals in addition to the yearly totals. Here’s a simple example:

```
SQL> SELECT department_id, job_id, SUM(salary)
        FROM employees
        GROUP BY CUBE (department_id, job_id);
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>44000</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>130000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>60000</td>
</tr>
<tr>
<td>30</td>
<td>PU_MAN</td>
<td>110000</td>
</tr>
<tr>
<td>30</td>
<td>PU_CLERK</td>
<td>139000</td>
</tr>
</tbody>
</table>

SQL>
The GROUP BY Clause with a GROUPING Operator

As you’ve seen, the ROLLUP operator gets you the superaggregate subtotals and grand totals. The GROUPING operator in a GROUP BY clause helps you distinguish between superaggregated subtotals and the grand total column from the other row data.

The GROUP BY Clause with a GROUPING SETS Operator

The GROUPING SETS operator lets you group multiple sets of columns when you’re calculating aggregates such as sums. Here’s an example that shows how you can use this operator to calculate aggregates over three groupings: (year, region, item), (year, item), and (region, item). The GROUPING SETS operator eliminates the need for inefficient UNION ALL operators.

```sql
SQL> SELECT year, region, item, sum(sales)
    2   FROM regional_salesitem
    3   GROUP BY
    4   GROUPING SETS (( year, region, item),
    5           (year, item), (region, item));
```

The GROUP BY Clause with a HAVING Operator

The HAVING operator lets you restrict or exclude the results of a GROUP BY operation, in essence putting a WHERE condition on the GROUP BY clause’s result set. In the following example, the HAVING operator restricts the query results to only those departments that have a maximum salary greater than 20,000:

```sql
SQL> SELECT department_id, max(salary)
2  FROM employees
3  GROUP BY department_id
4  HAVING MAX(salary)>20000;
```

Writing Subqueries

Subqueries resolve queries that have to be processed in multiple steps—where the final result depends on the results of a child query or subquery to the main query. If the subquery occurs in the WHERE clause of the statement, it’s called a nested subquery.

Top-N Analysis

The following query gives you the top ten employees in a firm ranked by salary. You can just as easily retrieve the bottom ten employees by using the ORDER BY clause instead of the ORDER BY DESC clause.

```sql
SQL> SELECT emp_id, emp_name, job, manager, salary
    2   FROM (SELECT emp_id, emp_name, job, manager, salary,
    3       RANK() OVER (ORDER BY SALARY DESC NULLS LAST) AS Employee_Rank
    4   FROM employees
    5   ORDER BY SALARY DESC NULLS LAST)
    6   WHERE employee_Rank < 5;
```

Subqueries can be single-row or multiple-row SQL statements. Let’s take a quick look at both types of subqueries.
Single-Row Subqueries

Subqueries are useful when you need to answer queries on the basis of as-yet unknown values, such as “Which employees have a salary higher than the employee with the employee ID 9999?” To answer such a question, a subquery or inner query is executed first (and only once). The result of this subquery is then used by the main or outer query. Here’s the query:

```
SQL> SELECT first_name||last_name, dept
    2  FROM employee
    3  WHERE sal >
    4  (SELECT sal
    5  FROM emp
    6  WHERE empno= 9999);
```

Multiple-Row Subqueries

A multiple-row subquery returns multiple rows in the output, so you need to use multiple-row comparison operators, such as IN, ANY, and ALL. Using a single-row operator with a multiple-row subquery returns this common Oracle error:

```
ERROR:
ORA-01427: single-row subquery returns more than one row
```

Multiple-Column Subqueries

Multiple-column subqueries are queries where the inner query retrieves the values of more than one column. The rows in the subquery are then evaluated in the main query in pair-wise comparison, column by column and row by row.

Advanced Subqueries

Correlated subqueries are more complex than regular subqueries and answer questions such as “What are the names of all employees whose salary is below the average salary of their department?” The inner query computes the average salary, and the outer or main query gets the employee information. However, for each employee in the main (outer) query, the inner query has to be computed, because department averages depend on the department number of the employee in the outer query.

The Exists and Not Exists Operators

The EXISTS operator tests for the existence of rows in the inner query or subquery when you’re using subqueries. The NOT EXISTS operator tests for the nonexistence of rows in the inner query. In the following statement, the EXISTS operator will be TRUE if the subquery returns at least one row:

```
SQL> SELECT department_id
    2  FROM departments d
    3  WHERE EXISTS
    4  (SELECT * FROM employees e
    5  WHERE d.department_id = e.department_id);
```

Using Regular Expressions

Oracle Database 11g provides support for regular expressions that you can use as part of SQL statements. Regular expressions let you use special operators to manipulate strings or carry out a search.
Traditionally, developers used operators such as LIKE, REPLACE, and SUBSTRING in their search expressions. However, these expressions forced you to write lengthy SQL and PL/SQL code when performing complex searches. Oracle Database 11g lets you perform complex searches and string manipulations easily with regular expressions.

**Note** Oracle’s regular expression features follow the popular POSIX standards.

A regular expression searches for patterns in character strings. The character string has to be one of CHAR, VARCHAR2, NCHAR, or NVARCHAR2, and the regular expression function can be one of the following:

- REGEXP_LIKE
- REGEXP_REPLACE
- REGEXP_INSTRING
- REGEXP_SUBSTRING

The REGEXP_LIKE function evaluates strings using a specified set of characters. The regular expression function searches for a pattern in a string, which is specified with the SOURCE_STRING parameter in the function. The PATTERN variable represents the actual regular expression, which is the pattern to search for. A regular expression is usually a text literal; it can be one of CHAR, VARCHAR2, NCHAR, or NVARCHAR2, and it can be a maximum of 512 bytes long. You can also specify an optional match parameter to modify the matching behavior. For example, a value of i specifies case-insensitive matching, while c specifies case-sensitive matching.

Here is the syntax of the REGEXP_LIKE function:

```
REGEXP_LIKE(source_string, pattern [,match_parameter])
```

If you want to carry out string-manipulation tasks, you can use the REGEXP_INSTR, REGEXP_REPLACE, or REGEXP_SUBSTR built-in functions. These are really extensions of the normal SQL INSTR, REPLACE, and SUBSTR functions.

Regular expression features use characters like the period (.), asterisk (*), caret (^), and dollar sign ($), which are common in UNIX and Perl programming. The caret character (^), for example, tells Oracle that the characters following it should be at the beginning of the line. Similarly, the $ character indicates that a character or a set of characters must be at the very end of the line. Here’s an example using the REGEXP_LIKE function that picks up all names with consecutive vowels:

```sql
SQL> SELECT last_name
    FROM employees
    WHERE REGEXP_LIKE (last_name, '([aeiou])\1', 'i');

LAST_NAME
---------
Freedman
Greenberg
Khoo
Gee
Lee
...;
```

SQL>
Here’s another example that quickly searches for employees who were hired between the years 2000 and 2008.

```sql
SQL> SELECT emp_name, salary,
  2  TO_CHAR(hire_date,'yyyy') year_of_hire
  3  FROM emp
  4* WHERE REGEXP_LIKE (TO_CHAR (hire_date, 'yyyy'), '^200[0-8]$');
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>SALARY</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Austin</td>
<td>David</td>
<td>4800</td>
<td>2007</td>
</tr>
<tr>
<td>Chen</td>
<td>John</td>
<td>8200</td>
<td>2007</td>
</tr>
<tr>
<td>Alapati</td>
<td>Shannon</td>
<td>7700</td>
<td>2007</td>
</tr>
<tr>
<td>Baida</td>
<td>Shelli</td>
<td>2900</td>
<td>2007</td>
</tr>
<tr>
<td>Tobias</td>
<td>Sigal</td>
<td>2800</td>
<td>2007</td>
</tr>
<tr>
<td>Weiss</td>
<td>Matthew</td>
<td>8000</td>
<td>2007</td>
</tr>
</tbody>
</table>

Abstract Data Types

This section briefly reviews the important Oracle features that facilitate object-oriented programming. **Abstract types**, also called **object types**, are at the heart of Oracle’s object-oriented programming. Unlike a normal data type, an abstract data type contains a data structure along with the functions and procedures needed to manipulate the data; thus, data and behavior are coupled.

Object types are like other schema objects, and they consist of a name, attributes, and methods. Object types are similar to the concept of classes in C++ and Java. Oracle support for object-oriented features, such as types, makes it feasible to implement object-oriented features, such as encapsulation and abstraction, while modeling complex real-life objects and processes. Oracle also supports single inheritance of user-defined SQL types.

The CREATE TYPE Command

Object types are created by users and stored in the database like Oracle data types such as VARCHAR2, for example. The **CREATE TYPE** command lets you create an abstract template that corresponds to a real-world object. Here’s an example:

```sql
SQL> CREATE TYPE person AS object
  2  (name varchar2(30),
  3  phone varchar2(20));
```

```
Type created.
SQL>
```

Object Tables

Object tables contain objects such as the person type that was created in the previous section. Here’s an example:

```sql
SQL> CREATE TABLE person_table OF person;
```

```
Table created.
SQL>
```

Here’s the interesting part. The person_table table doesn’t contain single-value columns like a regular Oracle table—its columns are types, which can hold multiple values. You can use object
tables to view the data as a single-column table or a multicolumn table that consists of the components of the object type. Here’s how you would insert data into an object table:

```
SQL> INSERT INTO person_table
  2  VALUES
  3  ('john smith', '1-800-555-9999');
```

1 row created.

```
SQL>
```

Collections

Collections are ideal for representing one-to-many relationships among data. Oracle offers you two main types of collections: varrays and nested tables. We’ll look at these two types of collections in more detail in the following sections.

Varrays

A varray is simply an ordered collection of data elements. Each element in the array is identified by an index, which is used to access that particular element. Here’s how you declare a VARRAY type:

```
SQL> CREATE TYPE prices AS VARRAY (10) OF NUMBER (12,2);
```

Nested Tables

A nested table consists of an ordered set of data elements. The ordered set can be of an object type or an Oracle built-in type. Here’s a simple example:

```
SQL> CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

To access the elements of a collection with SQL, you can use the TABLE operator, as shown in the following example. Here, history is a nested table and courses is the column you want to insert data into:

```
SQL> INSERT INTO
  TABLE(SELECT courses FROM department WHERE name = 'History')
  VALUES('Modern India');
```

Type Inheritance

You can create not just types, but also type hierarchies, which consist of parent supertypes and child subtypes connected to the parent types by inheritance. Here’s an example of how you can create a subtype from a supertype. First, create the supertype:

```
SQL> CREATE TYPE  person_t AS OBJECT (
    name varchar2(80),
    social_sec_no number,
    hire_date date,
    member function age() RETURN number,
    member function print() RETURN varchar2) NOT FINAL;
```

Next, create the subtype, which will inherit all the attributes and methods from its supertype:
The Cast Operator

The CAST operator enables you to do two things. It lets you convert built-in data types and also convert a collection-type value into another collection-type value.

Here’s an example of using CAST with built-in data types:

    SQL> SELECT product_id,
                cast(description AS VARCHAR2(30))
            FROM product_desc;

PL/SQL

Although SQL is easy to learn and has a lot of powerful features, it doesn’t allow the procedural constructs of third-generation languages such as C. PL/SQL is Oracle’s proprietary extension to SQL, and it provides you the functionality of a serious programming language. One of the big advantages of using PL/SQL is that you can use program units called procedures or packages in the database, thus increasing code reuse and performance.

The Basic PL/SQL Block

A PL/SQL block is an executable program. A PL/SQL code block, whether encapsulated in a program unit such as a procedure or specified as a free-form anonymous block, consists of the following structures, with a total of four key statements, only two of which are mandatory:

- DECLARE: In this optional section, you declare the program variables and cursors.
- BEGIN: This mandatory statement indicates that SQL and PL/SQL statements will follow it.
- EXCEPTION: This optional statement specifies error handling.
- END: This mandatory statement indicates the end of the PL/SQL code block.

Here’s an example of a simple PL/SQL code block:

    SQL> declare isbn number(9)
        begin
            isbn := 123456789;
            insert into book values (isbn, 'databases', 59.99);
            commit;
        end;
    SQL>

Declaring Variables

You can declare both variables and constants in the DECLARE section. Before you can use any variable, you must first declare it. A PL/SQL variable can be a built-in type such as DATE, NUMBER, VARCHAR2, or CHAR, or it can be a composite type such as VARRAY. In addition, PL/SQL uses the BINARY_INTEGER and BOOLEAN data types.
Here are some common PL/SQL variable declarations:

```plsql
hired_date DATE;
emp_name VARCHAR2(30);
```

In addition to declaring variables, you can also declare constants, as shown in the following example:

```plsql
tax_rate constant number := 0.08;
```

You can also use the `%TYPE` attribute to declare a variable that is of the same type as a specified table’s column, as shown here:

```plsql
emp_num employee.emp_id%TYPE;
```

The `%ROWTYPE` attribute specifies that the record (row) is of the same data type as a database table. In the following example, the DeptRecord record has all the columns contained in the department table, with identical data types and length:

```plsql
declare
v_DeptRecord department%ROWTYPE;
```

### Writing Executable Statements

After the `BEGIN` statement, you can enter all your SQL statements. These look just like your regular SQL statements, but notice the difference in how you handle a `SELECT` statement and an `INSERT` statement in the following sections.

#### A SELECT Statement in PL/SQL

When you use a `SELECT` statement in PL/SQL, you need to store the retrieved values in variables, as shown here:

```plsql
DECLARE
  name  VARCHAR2(30);
BEGIN
  SELECT employee_name INTO name FROM employees WHERE emp_id=99999;
END;
/
```

#### DML Statements in PL/SQL

Any `INSERT`, `DELETE`, or `UPDATE` statements in PL/SQL work just as they do in regular SQL. You can use the `COMMIT` statement after any such operation, as shown here:

```plsql
BEGIN
  DELETE FROM employee WHERE emp_id = 99999;
  COMMIT;
END;
/
```

### Handling Errors

In PL/SQL, an error or a warning is called an *exception*. PL/SQL has some internally defined errors, and you can also define your own error conditions. When any error occurs, an exception is raised.
and program control is handed to the exception-handling section of the PL/SQL program. If you
define your own error conditions, you have to raise exceptions by using a special RAISE statement.
The following example shows an exception handler using the RAISE statement:

```sql
DECLARE
    acct_type INTEGER := 7;
BEGIN
    IF acct_type NOT IN (1, 2, 3) THEN
        RAISE INVALID_NUMBER;  -- raise predefined exception
    END IF;
EXCEPTION
    WHEN INVALID_NUMBER THEN
        ROLLBACK;
END;
/
```

**PL/SQL Control Structures**

PL/SQL offers you several types of control structures, which enable you to perform iterations of code
or conditional execution of certain statements. The various types of control structures in PL/SQL are
covered in the following sections.

**Conditional Control**

The main type of conditional control structure in PL/SQL is the IF statement, which enables condi-
tional execution of statements. You can use the IF statement in three forms: IF-THEN, IF-THEN-ELSE,
and IF-THEN-ELSEIF. Here’s an example of a simple IF-THEN-ELSEIF statement:

```sql
BEGIN
    . . .
    IF total_sales > 100000 THEN
        bonus := 5000;
    ELSEIF total_sales > 35000 THEN
        bonus := 500;
    ELSE
        bonus := 0;
    END IF;
    INSERT INTO new_payroll VALUES (emp_id, bonus . . .);
END;
/
```

**PL/SQL Looping Constructs**

PL/SQL loops provide a way to perform iterations of code for a specified number of times or until a
certain condition is true or false. The following sections cover the basic types of looping constructs.

**The Simple Loop**

The simple loop construct encloses a set of SQL statements between the keywords LOOP and END
LOOP. The EXIT statement ends the loop. You use the simple loop construct when you don’t know
how many times the loop should execute. The logic inside the LOOP and END LOOP statements decides
when the loop is terminated.
In the following example, the loop will be executed until a quality grade of 6 is reached:

```sql
LOOP
  . . .
  if quality_grade > 5  then
  . . .
  EXIT;
  end if;
END LOOP;
```

Another simple loop type is the LOOP . . . EXIT . . . WHEN construct, which controls the duration of the loop with a WHEN statement. A condition is specified for the WHEN statement, and when this condition becomes true, the loop will terminate. Here's a simple example:

```sql
DECLARE
  count_num  NUMBER(6);
BEGIN
  count_num := 1;
  LOOP
    dbms_output.put_line(' This is the current count '|| count_num);
    count_num := count_num + 1;
    Exit when count_num > 100;
  END LOOP;
END;
```

The **WHILE** Loop

The **WHILE** loop specifies that a certain statement be executed while a certain condition is true. Note that the condition is evaluated outside the loop. Each time the statements within the LOOP and END LOOP statements are executed, the condition is evaluated. When the condition no longer holds true, the loop is exited. Here's an example of the **WHILE** loop:

```sql
WHILE total <= 25000
  LOOP
    . . .
    SELECT sal INTO salary FROM emp WHERE . . .
    total := total + salary;
  END LOOP;
```

The **FOR** Loop

The **FOR** loop is used when you want a statement to be executed a certain number of times. The **FOR** loop emulates the classic do loop that exists in most programming languages. Here's an example of the **FOR** loop:

```sql
BEGIN
  FOR count_num IN 1..100
  LOOP
    dbms_output.put_line('The current count is : '|| count_num);
  END LOOP;
END;
```

**PL/SQL Records**

*Records* in PL/SQL let you treat related data as a single unit. Records contain fields, with each field standing for a different item. You can use the `%ROWTYPE` attribute to declare a table’s columns as a
record, which uses the table as a cursor template, or you can create your own records. Here’s a simple example of a record:

DECLARE
    TYPE MeetingTyp IS RECORD (
        date_held DATE,
        location VARCHAR2(20),
        purpose VARCHAR2(50));

To reference an individual field in a record, you use dot notation, as shown here:

MeetingTyp.location

Using Cursors

An Oracle cursor is a handle to an area in memory that holds the result set of a SQL query, enabling you to individually process the rows in the result set. Oracle uses implicit cursors for all DML statements. Explicit cursors are created and used by application coders.

Implicit Cursors

Implicit cursors are automatically used by Oracle every time you use a SELECT statement in PL/SQL. You can use implicit cursors in statements that return just one row. If your SQL statement returns more than one row, an error will result.

In the following PL/SQL code block, the SELECT statement makes use of an implicit cursor:

DECLARE
    emp_name varchar2(40);
    salary float;
BEGIN
    SELECT emp_name, salary FROM employees
    WHERE employee_id=9999;
    dbms_output.put_line('employee_name : '||emp_name||'
salary :'||salary);
END;
/

Explicit Cursors

Explicit cursors are created by the application developer, and they facilitate operations with a set of rows, which can be processed one by one. You always use explicit cursors when you know your SQL statement will return more than one row. Notice that you have to declare an explicit cursor in the DECLARE section at the beginning of the PL/SQL block, unlike an implicit cursor, which you never refer to in the code.

Once you declare your cursor, the explicit cursor will go through these steps:

1. The OPEN clause will identify the rows that are in the cursor and make them available for the PL/SQL program.
2. The FETCH command will retrieve data from the cursor into a specified variable.
3. The cursor should always be explicitly closed after your processing is completed.

Listing A-4 shows how a cursor is first created and then used within a loop.
Listing A-4. Using an Explicit Cursor

DECLARE
/* The cursor select_emp is explicitly declared */
    CURSOR select_emp IS
    select emp_id, city
    from employees
    where city = 'DALLAS';
    v_empno employees.emp_id%TYPE;
    v_empcity employees.city%TYPE;
BEGIN
/* The cursor select_emp is opened */
    Open select_emp;
    LOOP
/* The select_emp cursor data is fetched into v_empno variable */
    FETCH select_emp into v_empno;
    EXIT WHEN select_emp%NOTFOUND;
    dbms_output.put_line(v_empno|| ','||v_empcity);
    END LOOP;
/* The cursor select_emp is closed */
    Close select_emp;
END;
/

Cursor Attributes

In the example shown in Listing A-4, a special cursor attribute, %NOTFOUND, is used to indicate when
the loop should terminate. Cursor attributes are very useful when you’re dealing with explicit cursors.
Here are the main cursor attributes:

- %ISOPEN is a Boolean attribute that evaluates to false after the SQL statement completes
  execution. It returns true as long as the cursor is open.
- %FOUND is a Boolean attribute that tests whether the SQL statement matches any row—that is,
  whether the cursor has any more rows to fetch.
- %NOTFOUND is a Boolean attribute that tells you that the SQL statement doesn’t match any row,
  meaning there are no more rows left to fetch.
- %ROWCOUNT gives you the number of rows the cursor has fetched so far.

Cursor FOR Loops

Normally when you use explicit cursors, cursors have to be opened, the data has to be fetched, and
finally the cursor needs to be closed. A cursor FOR loop automatically performs the open, fetch, and
close procedures, which simplifies your job. Listing A-5 shows an example that uses a cursor FOR
loop construct.

Listing A-5. Using the Cursor FOR Loop

DECLARE
    CURSOR emp_cursor IS
    SELECT emp_id, emp_name, salary
    FROM employees;
    v_emp_info employees%RowType;
Begin
    FOR emp_info IN emp_cursor
    LOOP
        dbms_output.put_line ('Employee id : '||emp_id||'Employee name : '|| emp_name||'Employee salary :'||salary);
    END LOOP;
    END;
/

Cursor Variables

Cursor variables point to the current row in a multirow result set. Unlike a regular cursor, though, a cursor variable is dynamic—that is, you can assign new values to a cursor variable and pass it to other procedures and functions. Let’s look at how you can create cursor variables in PL/SQL.

First, define a REF CURSOR type, as shown here:

DECLARE
    TYPE EmpCurTyp IS REF CURSOR RETURN dept%ROWTYPE;

Next, declare cursor variables of the type DeptCurTyp in an anonymous PL/SQL code block or in a procedure (or function), as shown in the following code snippet:

DECLARE
    TYPE EmpRecTyp IS RECORD (
        Emp_id NUMBER(9),
        emp_name VARCHAR2(30),
        sal NUMBER(7,2));
    TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
    emp_cv EmpCurTyp;  -- declare cursor variable

Procedures, Functions, and Packages

A PL/SQL procedure can be used to perform various DML operations. The following is a simple Oracle procedure:

create or replace procedure new_employee (emp_id number,
last_name varchar(2), first_name varchar(2))
is
    begin
        insert into employees values ( emp_id, last_name, first_name);
    end new_employee;
/

Unlike a PL/SQL procedure, a function returns a value, as shown in the following example:

CREATE OR REPLACE FUNCTION sal_ok (salary REAL, title VARCHAR2) RETURN BOOLEAN IS
    min_sal REAL;
    max_sal REAL;
BEGIN
    SELECT losal, hisal INTO min_sal, max_sal FROM sals
    WHERE job = title;
    RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;

Oracle packages are objects that usually consist of several related procedures and functions, and the package is usually designed to perform an application function by invoking all the related procedures

and functions within the package. Packages are extremely powerful, because they can contain large amounts of functional code and be repeatedly executed by several users.

A package usually has two parts: a package specification and a package body. The package specification declares the variables, cursors, and subprograms (procedures and functions) that are part of the package. The package body contains the actual cursors and subprogram code.

Listing A-6 shows a simple Oracle package.

**Listing A-6. A PL/SQL Package**

```plsql
/* First, the Package Specification */
create or replace package emp_pkg as
  type list is varray (100) of number (5);
  procedure new_employee (emp_id number, last_name varchar2, first_name varchar2);
  procedure salary_raise ( emp_id number, raise number);
end emp_pkg;
/

/* The Package Body follows */
create or replace package body emp_pkg as
  procedure new_employee (emp_id number, last_name varchar2, first_name varchar2) is
    begin
      insert into employees values ( emp_id, last_name, first_name);
    end new_employee;
    procedure salary_raise ( emp_num number, raise_pct real) is
      begin
        update employees set salary = salary * raise_pct
          where emp_id = emp_num;
      end salary_raise;
end emp_pkg;
/

If you want to use emp_pkg to award a raise to an employee, all you have to do is execute the following:

SQL> EXECUTE emp_pkg.salary_raise(99999, 0.15);
```

**Oracle XML DB**

A typical organization has information stored in multiple formats, some of which may be organized in relational databases, but most of which is stored outside the database. The nondatabase information may be stored in application-specific formats, such as Excel spreadsheets. Storing the nondatabase information in XML format instead makes it easier to access and update nonstructured organizational information.

Oracle XML DB isn’t really a special type of database for XML. It simply refers to the set of built-in XML storage and retrieval technologies for the manipulation of XML data. Oracle XML DB provides the advantages of object-relational database technology and XML technology. For example, one of the major problems involved in dealing with XML data from within a relational database is that most XML data is hierarchical in nature, whereas the Oracle database is based on the relational model. Oracle manages to deal effectively with the hierarchical XML data by using special SQL operators and methods that let you easily query and update XML data in an Oracle database. Oracle XML DB builds
the XML Document Object Model (DOM) into the Oracle kernel. Thus, most XML operations are treated as part of normal database processing.

Oracle XML DB provides the ability to view both structured and nonstructured information as relational data. You can view the data as either rows in a table or nodes in an XML document.

Here is a brief list of the benefits offered by Oracle XML DB:

- You can access XML data using regular SQL queries.
- You can use Oracle’s OLTP, data warehousing, test, spatial data, and multimedia features to process XML data.
- You can generate XML from an Oracle SQL query.
- You can transform XML into HTML format easily.

Storing XML in Oracle XML DB

Oracle uses a special native data type called XMLType to store and manage XML data in a relational table. XMLType and XMLBINType, which is another built-in type for XML data, enable you to leave the XML parsing, storage, and retrieval to the Oracle database. You can use the XMLType data type just as you would the usual data types in an Oracle database. You can now store a well-formed XML document in the database as an XML test using the CLOB base data type.

Here’s an example of using the XMLType data type:

```
SQL> CREATE TABLE sales_catalog_table
2  (sales_num  number(18),
3  sales_order xmltype);
Table created.
```

```
SQL> DESC sales_catalog_table
Name                  Null? Type
--------------------- ----- --------
SALES_NUM                   NUMBER(18)
SALES_ORDER                 XMLTYPE
```

The XMLType data type comes with a set of XML-specific methods, which you use to work with XMLType objects. You can use these methods to perform common database operations, such as checking for the existence of a node and extracting a node. The methods also support several operators that enable you to access and manipulate XML data as part of a regular SQL statement. These operators follow the emerging SQL/XML standard. Using the well-known XPath notation, the SQL/XML operators traverse XML structures to find the node or nodes on which they should use the SQL operations. Here are some of the important SQL/XML operators:

- Extract() extracts a subset of the nodes contained in the XMLType.
- ExistsNode() checks whether a certain node exists in the XMLType.
- Validating() validates the XMLType contents against an XML schema.
- Transform() performs an XSL transformation.
- ExtractValue() returns a node corresponding to an XPath expression.

XML is in abstract form compared to the normal relational table entries. To optimize and execute statements that involve XML data, Oracle uses a query-rewrite mechanism to transform an XPath expression into an equivalent regular SQL statement. The optimizer then processes the transformed SQL statement like any other SQL statement.
You can store XML in Oracle XML DB in the following ways:

- You can use SQL or PL/SQL to insert the data. Using `XMLType` constructors, you must first convert the sourced data into an `XMLType` instance.
- You can use the Oracle XML DB repository to store the XML data.

Here’s a simple example using the `sales_catalog_table` table to demonstrate how to perform SQL-based DML operations with an XML-enabled table. In Listing A-7, an XML document is inserted into `sales_catalog_table`.

Listing A-7. Inserting an XML Document into an Oracle Table

```sql
SQL> INSERT INTO sales_catalog_table
2    VALUES (123456,
3    XMLTYPE(
4    '<SalesOrder>
5    <Reference>Alapati - 200302201428CDT</Reference>
6    <Actions/>
7    <Reject/>
8    <Requestor>Nina U. Alapati</Requestor>
9    <User>ALAPATI</User>
10    <SalesLocation>Dallas</SalesLocation>
11    <ShippingInstructions/>
12    <DeliveryInstructions>Bicycle Courier</DeliveryInstructions>
13    <ItemDescriptions>
14    <ItemDescription ItemNumber="1">
15    <Description>Expert Oracle DB Administration</Description>
16    <ISBN Number="1590590228" Price="59.95" Quantity="2"/>
17    </ItemDescription>
18    </ItemDescriptions>
19*  </SalesOrder>'));
1 row created.

SQL>
```

You can query the `sales_catalog_table` table’s `sales_order` column, as shown in Listing A-8, to view the XML document in its original format.

Listing A-8. Viewing XML Data Stored in an Oracle Table

```sql
SQL> SELECT sales_order FROM
2    sales_catalog_table;
3    <SalesOrder>
4    <Reference>Alapati - 200302201428CDT</Reference>
5    <Actions/>
6    <Reject/>
7    <Requestor>Sam R. Alapati</Requestor>
8    <User>ALAPATI</User>
9    <SalesLocation>Dallas</SalesLocation>
10    <ShippingInstructions/>
11    <DeliveryInstructions>Bicycle Courier</DeliveryInstructions>
12    <ItemDescriptions>
13    <ItemDescription ItemNumber="1">
14    <Description>Expert Oracle DB Administration</Description>
15    <ISBN Number="1590590228" Price="59.95" Quantity="2"/>
16    </ItemDescription>
17    </ItemDescriptions>
18    </SalesOrder>;
```
Once you create the sales_catalog_table table, it's very easy to retrieve data using one of the methods I just described. The following example shows how to query the table using the `extract()` method. Note that the query includes XPath expressions and the SQL/XML operators `extractValue` and `existsNode` to find the requestor's name where the value of the `/SalesOrder/SalesLocation/text()` node contains the value Dallas.

```sql
SQL> SELECT extractValue(s.sales_order,'/SalesOrder/Requestor')
   2   FROM sales_catalog_table s
   3   WHERE existsNode(s.SALES_ORDER,
   4* '/SalesOrder[SalesLocation="Dallas"]') = 1;
```

The Oracle XML DB Repository

The best way to process XML documents in Oracle XML DB is to first load them into a special repository called the Oracle XML DB repository. The XML repository is hierarchical, like most XML data, and it enables you to easily query XML data. The paths and URLs in the repository represent the relationships among the XML data, and a special hierarchical index is used to traverse the folders and paths within the repository. The XML repository can hold non-XML data such as JPEG images, Word documents, and more.

You can use SQL and PL/SQL to access the XML repository. XML authoring tools can directly access the documents in the XML repository using popular Internet protocols such as HTTP, FTP, and WebDAV. For example, you can use Windows Explorer, Microsoft Office, and Adobe Acrobat to work with the XML documents that are stored in the XML repository. XML is by nature document-centric, and the XML repository provides applications with a file abstraction when dealing with XML data.

Setting Up an XML Schema

Before you can start using Oracle XML DB to manage XML documents, you need to perform the following tasks:

1. Create an XML schema. For example, SalesOrder, shown in Listing A-7, is a simple XML schema that reflects a simple XML document. Within the SalesOrder schema are elements such as ItemDescription, which provides details about the attributes of the component items.

2. Register the XML schema. After the XML schema is created, you must register it with the Oracle database using a PL/SQL procedure. When you register the XML schema, Oracle will create the SQL objects and the `XMLType` tables that are necessary to store and manage the XML documents. For the example shown in Listing A-6, registering the XML schema will create a table called SalesOrder automatically, with one row in the table for each SalesOrder document loaded into the XML repository. The XML schema is registered under the URL `http://localhost:8080/home/SCOTT/xdb/salesorder.xsd`, and it contains the definition of the SalesOrder element.
Creating a Relational View from an XML Document

Even if a developer doesn’t know much XML, he or she can use the XML documents stored in the Oracle database by creating relational views based on the XML documents. The following example maps nodes in an XML document to columns in a relational view called salesorder_view:

```
SQL> CREATE OR REPLACE VIEW salesorder_view
  2  (requestor,description,sales_location)
  3  AS SELECT
  4  extractValue(s.sales_order,'/SalesOrder/Requestor'),
  5  extractValue(s.sales_order,'/SalesOrder/Sales_Location')
  6* FROM sales_Catalog_Table s;
View created.
```

You can query salesorder_view like you would any other view in an Oracle database, as shown here:

```
SQL> SELECT requestor,sales_location FROM salesorder_view;
REQUESTOR        SALES_LOCATION
Aparna Alapati    Dallas
```

Oracle and Java

You can use both PL/SQL and Java to write applications that need Oracle database access. Although PL/SQL has several object-oriented features, the Java language is well known as an object-oriented programming language. If your application needs heavy database access and must process large amounts of data, PL/SQL is probably a better bet. However, for open distributed applications, Java-based applications are more suitable.

The Oracle database contains a Java Virtual Machine (JVM) to interpret Java code from within the database. Just as PL/SQL enables you to store code on the server and use it multiple times, you can also create Java stored procedures and store them in the database. These Java stored procedures are in the form of Java classes. You make Java files available to the Oracle JVM by loading them into the Oracle database as schema objects.

You can use the Java programming language in several ways in an Oracle database. You can invoke Java methods in classes that are loaded in the database in the form of Java stored procedures. You can also use two different application programming interfaces (APIs), Java Database Connectivity (JDBC) or SQLJ, to access the Oracle database from a Java-based application program. In the sections that follow, we’ll briefly look at the various ways you can work with Java and the Oracle database.

Java Stored Procedures

Java stored procedures are, of course, written using Java, and they facilitate the implementation of data-intensive business logic using Java. These procedures are stored within the database like PL/SQL stored procedures. Java stored procedures can be seen as a link between the Java and non-Java environments.
You can execute Java stored procedures just as you would PL/SQL stored procedures. Here’s a summary of the steps involved in creating a Java stored procedure:

1. Define the Java class.
2. Using the Java compiler, compile the new class.
3. Load the class into the Oracle database. You can do this by using the loadjava command-line utility.
4. Publish the Java stored procedure.

Once you’ve completed these steps, you can invoke the Java stored procedure.

**JDBC**

JDBC is a popular method used to connect to an Oracle database from Java. Chapter 10 contains a complete example of a Java program. JDBC provides a set of interfaces for querying databases and processing SQL data in the Java programming language.

Listing A-9 shows a simple JDBC program that connects to an Oracle database and executes a simple SQL query.

**Listing A-9. A Simple JDBC Program**

```java
import java.sql.*;
public class JDBCExample {
    public static void main(String args[]) throws SQLException {
        /* Declare the type of Oracle Driver you are using */
        {DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        /* Create a database connection for the JDBC program */
        Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@nicholas:1521:aparna","hr","hr");
        Statement stmt = conn.createStatement();
        /* Pass a query to SQL and store the results in the result set rs */
        ResultSet rs = stmt.executeQuery("select emp_id, emp_name,salary from employees");
        /* Using the while loop, result set rs is accessed row by row */
        while(rs.next()){
            int number = rs.getInt(1);
            String name = rs.getString(2);
            System.out.println(number+ " "+name+ " "+salary);
        }
        /* Close the JDBC result set and close the database connection */
        rs.close();
        conn.close();
    }
}

JDBC is ideal for dynamic SQL, where the SQL statements aren’t known until run time.

**SQLJ**

SQLJ is a complementary API to JDBC, and it’s ideal for applications in which you’re using static SQL (SQL that’s known before the execution). Being static, SQLJ enables you to trap errors before they
occur during run time. Keep in mind that even with SQLJ, you still use JDBC drivers to access the database.

There are three steps involved in executing a SQLJ program:

1. Create the SQLJ source code.
2. Translate the SQLJ source code into Java source code using a Java compiler.
3. Execute the SQLJ runtime program after you connect to the database.

Listing A-10 contains a simple SQLJ example that shows how to execute a SQL statement from within Java.

Listing A-10. A Simple SQLJ Program

```java
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
/* Declare the variables here */
/* Define an Iterator type to store query results */
#sql iterator ExampleIter (int emp_id, String emp_name,float salary);
public class MyExample
/* The main method */
{ public static void main (String args[]) throws SQLException
{ /* Establish the database connection for SQLJ */
 Oracle.connect
("jdbc:oracle:thin:@shannon:1234:nicholas1", "hr", "hr");
/* Insert a row into the employees table */
#sql { insert into employees (emp_id, emp_name, salary)
values (1001, 'Nina Alapati', 50000) };
/* Create an instance of the iterator ExampleIter */
ExampleIter iter;
/* Store the results of the select query in the iterator ExampleIter */
#sql iter={ select emp_id, emp_name, salary from employees }; /* Access the data stored in the iterator, using the next() method */
while (iter.next()) {
 System.out.println
( iter.emp_id()+" "+iter.emp_name()+" "+iter.salary());
}
}
```

As you can see from the SQLJ example in Listing A-10, SQLJ is nothing more than embedded SQL in a Java program. Using SQLJ, you can easily make calls to the database from Java. For a wealth of information on Oracle and Java, please visit Oracle’s Java Center web site (http://otn.oracle.com/tech/java/content.html).

This appendix just provided a very brief introduction to the Oracle Database 11g SQL and PL/SQL capabilities. Although Oracle DBAs aren’t always expected be very proficient in SQL and PL/SQL, the more you know about them, the better off you’ll be as a professional Oracle DBA.
Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>! (bang, or exclamation point)</td>
<td>using operating system commands from SQL*Plus, 120</td>
</tr>
<tr>
<td>$ (dollar sign)</td>
<td>character, SQL, 1238</td>
</tr>
<tr>
<td>$ sign, UNIX</td>
<td>53, 55, 69</td>
</tr>
<tr>
<td>% (percent sign)</td>
<td>character, SQL, 1224</td>
</tr>
<tr>
<td>&amp; prefixing variable names, SQL*Plus</td>
<td>126</td>
</tr>
<tr>
<td>* (asterisk)</td>
<td>character, SQL, 1238</td>
</tr>
<tr>
<td>. (period)</td>
<td>character, SQL, 1238</td>
</tr>
<tr>
<td>/ (slash) command</td>
<td>126, 128</td>
</tr>
<tr>
<td>/* ... */</td>
<td>comments SQL*Plus, 128</td>
</tr>
<tr>
<td>@@commandfile notation, SQL</td>
<td>129</td>
</tr>
<tr>
<td>^ (caret)</td>
<td>character, SQL, 1238</td>
</tr>
<tr>
<td>_ (underscore)</td>
<td>character, SQL, 1224</td>
</tr>
<tr>
<td></td>
<td>(pipe) command, UNIX/Linux, 52, 57</td>
</tr>
</tbody>
</table>

Numerics

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1 (one-to-one) relationship</td>
<td>26</td>
</tr>
<tr>
<td>1:M (one-to-many) relationship</td>
<td>26</td>
</tr>
<tr>
<td>10446 trace event, SQL</td>
<td>1174</td>
</tr>
<tr>
<td>1NF (First Normal Form)</td>
<td>30–31</td>
</tr>
<tr>
<td>2NF (Second Normal Form)</td>
<td>31–33</td>
</tr>
<tr>
<td>3NF (Third Normal Form)</td>
<td>33</td>
</tr>
<tr>
<td>4NF (Fourth Normal Form)</td>
<td>34</td>
</tr>
<tr>
<td>5NF (Fifth Normal Form)</td>
<td>34</td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abnormal program failure</td>
<td>338</td>
</tr>
<tr>
<td>ACID properties, transactions</td>
<td>340</td>
</tr>
<tr>
<td>acl parameter, CREATE_ACL procedure</td>
<td>616</td>
</tr>
<tr>
<td>ADD command, ALTER TABLE</td>
<td>271</td>
</tr>
<tr>
<td>ADD LOGFILE GROUP syntax</td>
<td>983</td>
</tr>
<tr>
<td>ADD PARTITION command, ALTER TABLE</td>
<td>291</td>
</tr>
<tr>
<td>ADD_FILE command, Data Pump</td>
<td>702, 703, 704, 713</td>
</tr>
<tr>
<td>ADDFILE procedure, DBMS_LOGMNR</td>
<td>844</td>
</tr>
<tr>
<td>ADD_FILTER procedure</td>
<td>1210</td>
</tr>
<tr>
<td>ADDITIVE performance problems</td>
<td>1183–1184</td>
</tr>
<tr>
<td>automatic performance-tuning features</td>
<td>1132</td>
</tr>
<tr>
<td>Access Advisor</td>
<td>see SQL Access Advisor</td>
</tr>
<tr>
<td>access control</td>
<td>615–618</td>
</tr>
<tr>
<td>accepting addresses</td>
<td>securing network, 614</td>
</tr>
<tr>
<td>ACCEPT command, SQL*Plus</td>
<td>121</td>
</tr>
<tr>
<td>ACCEPT_SQL_PATCH procedure</td>
<td>1038</td>
</tr>
<tr>
<td>ACCEPT_SQL_PROFILE procedure</td>
<td>1115</td>
</tr>
<tr>
<td>ACCEPT_SQL_PROFILES parameter</td>
<td>1117, 1118</td>
</tr>
<tr>
<td>Active Session History</td>
<td>see ASH</td>
</tr>
<tr>
<td>Active session management</td>
<td>554</td>
</tr>
<tr>
<td>Active Session Pool</td>
<td>560</td>
</tr>
<tr>
<td>Active Sessions chart, Database Control</td>
<td>1196</td>
</tr>
<tr>
<td>active/inactive pages, memory management</td>
<td>81</td>
</tr>
<tr>
<td>ADD_LOGFILE procedure, DBMS_LOGMNR</td>
<td>844, 845</td>
</tr>
<tr>
<td>ADDM (Automatic Database Diagnostic Monitor)</td>
<td>146, 209, 877–894</td>
</tr>
<tr>
<td>analyzing performance problems</td>
<td>1183–1184</td>
</tr>
<tr>
<td>automatic performance-tuning features</td>
<td>1132</td>
</tr>
<tr>
<td>configuring, 881–882</td>
<td></td>
</tr>
<tr>
<td>data dictionary views</td>
<td>893</td>
</tr>
<tr>
<td>determining optimal I/O performance</td>
<td>884</td>
</tr>
<tr>
<td>enabling, 882</td>
<td></td>
</tr>
<tr>
<td>findings</td>
<td>880</td>
</tr>
<tr>
<td>modes</td>
<td>882, 883–884</td>
</tr>
</tbody>
</table>
INDEX

performance tuning, 877
problems diagnosed by, 878–880
pronouncing ADDM, 877
purpose of ADDM, 878
reasons for invoking ADDM manually, 885
recommending, 878, 880–881
running ADDM, 885
using Database Control, 893
time-model statistics, 879–880
tuning-related advisors, 976
when ADDM runs, 882
ADDM analysis
Performance Analysis section, Database Control, 1197
ADDM reports
abbreviated ADDM report, 886
components of report, 885
confusing with AWR reports, 966
content of, 879
displaying, 884
viewing, 885–891
using Database Control, 890–891
using DBMS_ADVISOR, 890
addmrpt.sql script, 885, 888–890
ADD_POLICY procedure, DBMS_FGA, 594, 595
ADD_POLICY procedure, DBMS_RLS, 584, 585, 586
ADD_POLICY_CONTEXT procedure, 584
ADD_PRIVILEGE procedure, 616
ADD_SQLWKLD_REF procedure, 323
ADD_TABLE_RULES procedure, 323
adhoc directory, 397
admin class, Oracle Secure Backup, 788
ADMIN privilege, 612
ADMINISTER_RESOURCE_MANAGER privilege, 555
administrative context, 537
administrative directories, 397
administrative domain, Oracle Secure Backup, 786
administrative files, 394, 397, 399
Administrative wait class, 1163
Administrators page, Database Control, 148
ADMIN_RESTRICTIONS parameter, 614
ADR (Automatic Diagnostic Repository), 178, 211, 1022, 1023–1024
alert directory, 178
core directory, 178
DIAGNOSTIC_DEST parameter, 449
flood-controlled incident system, 1026
OFA guidelines, 396
trace directory, 178
viewing results of health check, 1033
ADR Base directory, 1024
ADR Command Interpreter see ADRCI
ADR Home directory, 1024
ADRCI, 208, 211, 1022, 1024–1026
ADR homopath, 1025
creating incident package, 1027
Advanced Installation option, Universal Installer, 416
Advanced Queuing (AQ), Oracle Streams, 670
Advanced Security option, 535, 603, 618
ADVISE FAILURE command, Data Recovery Advisor, 831
Advisor Central page
Database Control, 150, 323, 979
Grid Control, 890
Advisor Mode section, SQL Access Advisor, 323
advisors, 976–977
advisory framework, 212
creating tasks, 978
data dictionary views managing, 980
Database Control managing, 979
DBMS_ADVISOR package managing, 977–979
after-image records, redo log files, 176
after SUSPEND system event, 386
aggregate functions, SQL, 1229, 1232
aggregations, materialized views, 315
alert directory, ADR, 178
Diag Alert directory, 1024
alert log files, 16, 177
creating database, 482
location of file, 178
managing flash recovery area, 740
monitoring using, 958
starting Oracle instance, 479
viewing alert log using ADRCI, 1025
alert queue, 956
alert thresholds
AWR baseline metrics, 953
tablespace, 226–227
ALERT_QUE queue, 954
alerts
alert queue managing, 956
bytes remaining alert, 226
critical alert thresholds, 952, 954, 956
Database Control managing, 954–956
DBA_ALERT_HISTORY view, 958
DBA_OUTSTANDING_ALERTS view, 957
DBMS_SERVER_ALERT package, 955–956
invoking advisors from alert messages, 977
managing database alerts, 954, 956
managing/monitoring database, 214
monitoring system with Grid Control, 160
operation suspended alerts, 386
out-of-space warning and critical alerts, 741
percent full alert, 226
problem-related alerts, 952
response action, 954
security alerts, 617
server-generated alerts, 211, 386, 952–953
setting alert thresholds, 954
setting notification rules, 955
situations causing alerts, 952
snapshot-too-old alert, 959
stateful/stateless alerts, 952
tablespace alerts, 226–227, 956–958
threshold-based alerts, 226, 227, 952
V$ALERT_TYPES view, 958
viewing error alerts, 1029
warning alert thresholds, 952, 954, 956
Alerts page, Grid Control, 159
Alerts table, Database Control, 953
algebra, relational, 21–22
algorithms, encryption, 608
alias ASM filenames, 917, 918
aliases for objects see synonyms
All Metrics page, Database Control, 950
ALL PRIVILEGES option, 568
ALL value, STATISTICS_LEVEL parameter, 461
ALL views, data dictionary views, 204
ALLOCATE CHANNEL command, RMAN, 753, 757
allocation methods see resource allocation
ALLOCATE CHANNEL command, RMAN, 753, 757
 ALERTS page, Grid Control, 159
ALERTS table, Database Control, 953
 Algebra, relational, 21–22
 Algorithms, encryption, 608
 Alias ASM filenames, 917, 918
 Aliases for objects see synonyms
 All Metrics page, Database Control, 950
 ALL PRIVILEGES option, 568
 ALL value, STATISTICS_LEVEL parameter, 461
 ALL views, data dictionary views, 204
 ALLOCATE CHANNEL command, RMAN, 753, 757
 allocation methods see resource allocation
 alloc_bytes attribute
 CREATE_INDEX_COST procedure, 299
 ALLOW n CORRUPTION recovery option, 865
 ALL_ROWS hint, 1067
 ALL_ROWS value, OPTIMIZER_MODE, 1050
 ALTER DATABASE command, 224, 232
 DATAFILE option, 364
 OPEN option, 821
 ALTER DISKGROUP command
 adding/dropping disks, 915, 916
 DISK_REPAIR_TIME attribute, 908
 DISMOUNT FORCE clause, 913
 ALTER FLASHBACK ARCHIVE statement, 871
 ALTER INDEX command
 COALESCE option, 935
 INVISIBLE clause, 304
 modifying type of buffer pool, 189
 MONITORING USAGE clause, 305
 REBUILD ONLINE option, 935
 REBUILD option, 217, 303, 305
 ALTER MATERIALIZED VIEW statement, 319
 ALTER PROFILE command, 552, 599
 ALTER RESOURCE COST statement, 549
 ALTER SESSION command
 changing dynamic parameter values, 447, 448
 ENABLE RESUMABLE clause, 384, 385
 NAME parameter, 385
 Resumable Space Allocation, 383
 SET option, 262
 SET_SCHEMA option, 327
 TIMEOUT clause, 384
 ALTER SYSTEM command, 262
 activating Database Resource Manager, 565
 activating parameter limits in user
 profiles, 553
 changing dynamic parameter values, 447, 448
 creating Oracle Wallet, 241, 242, 609
 DEFERRED option, 448
 dynamic parameter changes, SPFILE, 496
 SCOPE clause, 496
 ALTER TABLE command
 ADD, 271
 ADD PARTITION, 291
 COALESCE PARTITION, 292
 COMPRESS, 275
 COMPRESS FOR ALL OPERATIONS, 275, 276
 COMPRESS FOR DIRECT_LOAD
 OPERATIONS, 275
 DISABLE NOVALIDATE/VALIDATE, 309
 DROP, 271
 DROP PARTITION, 291
 ENABLE NOVALIDATE/VALIDATE, 309
 ENABLE ROW MOVEMENT, 929
 EXCHANGE PARTITION, 291
 MERGE PARTITION, 291
 migrating tablespaces, 217
 modifying type of buffer pool, 189
 MOVE, 273, 275
 MOVE TABLESPACE, 935
 PURGE, 244
 READ ONLY, 273
 READ WRITE, 274
 RENAME COLUMN, 272
 RENAME, 272
 RENAME PARTITION, 291
 SHRINK SPACE, 929
 SPLIT PARTITION, 291
 UNCOMPRESS, 275
 ALTER TABLESPACE command
 altering temporary tablespaces, 231
 AUTOEXTEND parameter, 224
 expanding tablespaces, 223
 FLASHBACK OFF parameter, 856
 KEEP clause, 231, 232
 managing availability of tablespaces, 228, 229
 renaming tablespaces, 228
 RETENTION GUARANTEE clause, 363
 SHRINK SPACE clause, 231–232
 temporary tablespace groups, 234
 ALTER USER command, 547
 ACCOUNT LOCK option, 548
 changing user’s password, 547
 IDENTIFIED BY clause, 547
 password expiration, 599
 QUOTA clause, 545
 ALTER_ATTRIBUTES procedure, 1013
INDEX

ALTER_PARAM procedure, 533
ALTER_SQL_PROFILE procedure, 1115
ALWAYS log group, 843
ambr_backup_intermediate file, 912
analytical functions, SQL, 1231–1232
ANALYZE command
  collection of optimizer statistics, 1054
  COMPUTE STATISTICS option, 1055
  detecting data block corruption, 796
  INDEX_STATS view, 334
  managing/monitoring database, 213
ANALYZE TABLE statement
  LIST CHAINED ROWS clause, 279
  VALIDATE STRUCTURE clause, 934
ANALYZE_DB/INST/PARTIAL procedures, 883
ancestor incarnation, 822, 823
APM (Application Performance Monitoring) tools, 138
APPEND clause, SQL*Loader, 629
APPEND command, SQL*Plus, 131
APPEND option, SQL*Plus
  SAVE command, 124
  SPOOL command, 120
  STORE command, 116
application attributes
  fine-grained data access, 579
application contexts
  creating, 580, 581, 582–583
  fine-grained data access, 579–581
Application Express, 401
Application Performance Monitoring (APM), 138
application security, 618
Application Server Control, 139
Application wait class, 1163
ARBn (ASM rebalance) process, 907
arch directory, 397
architecture
  backup and recovery architecture, 201–202
  dedicated server architecture, 512
  Optimal Flexible Architecture, 393–400
  Oracle Database 11g architecture, 165–214
  RMAN architecture, 743–744
  shared server architecture, 512
archival (long-term) backups, RMAN, 780–782
ARCHIVE LOG command, SQL*Plus, 135
  confirming archive log mode, 491
archived redo logs, 175, 184
database recovery with RMAN, 815
log%t_%s_%r.arc format, 823
not needed for recovery, 867
archivelog deletion policy parameters, RMAN, 766
archivelog destination, 491
ARCHIVELOG keyword, CREATE DATABASE, 459
archivelog mode, 184, 459, 726
see also noarchivelog mode
  backing up redo log files, 775
  benefits of, 726
Flashback Database limitations, 861
noarchivelog/archivelog modes, 176, 491, 492
partial database backups, 794
whole database backups, 790
archivelog parameters, 459–460
archivelog retention policy
  out-of-space warning and critical alerts, 741
archivelogs
  archiving redo log files, 135
  backing up with RMAN, 775
  DBCA changing archive logging mode, 491–493
defining archivelog destinations, 459
  LIST ARCHIVELOG ALL command, 761
turning on, 492
  viewing details about, 135
archiver (ARCn) process, 181, 184
database hangs, 1187–1188
flash recovery area, 735
multiple archiver processes, 184
archiving data
  backup guidelines, 730
  Flashback Data Archive, 202, 870–874
  flashback data archiver (FBDA) process, 186
  partitioned tables, 281
archiving, UNIX, 76
ARCHVIELOG parameter, RMAN, 766
ARCn see archiver (ARCn) process
arguments, UNIX, 70
arrays
  transforming array data with rules, 668–670
  user-defined data types, 264
ARRAYSIZE variable, SQL*Plus, 107
AS clause
  materialized views, 319
  privileged SQL*Plus sessions, 99
AS OF clause, SELECT statement
ASH (Active Session History), 210, 947, 971–975
analyzing database activity, 1195
analyzing waits, 1169
current active session data, 972
manageability monitor light (MMNL) process, 185
objects with highest waits, 1170
older active session history data, 972
sample data, 948
V$ACTIVE_SESSION_HISTORY view, 1169
ASH reports, 972–975, 1201
analyzing recent session activity with, 1186
wait events, 975
ashrpt.sql script, 972, 1186
ashrpti.sql script, 1186
ASM (Automatic Storage Management), 94, 209, 900–920
architecture, 901–902
benefits of, 901
Cluster Synchronization Service (CSS) and, 902–904
COMPATIBLE parameters, 910
Database Control managing, 900
disk space allocation units, 902
DISK_REPAIR_TIME attribute, 910
fast mirror resync feature, 908–909
file templates, 917, 918
installing, 902
Logical Volume Manager, 900
OSASM group, 408
preferred mirror read feature, 909
specifying database or ASM instance, 452
storage, 902, 904
SYSASM privilege, 570
TEMPLATE.TNAME attributes, 910
ASM databases
backing up ASM database, 907
creating ASM-based database, 919
migrating with Database Control, 920–921
migrating with RMAN, 919–920
ASM disk groups, 900, 901, 902
adding disks to, 915
adding performance/redundancy with, 913–914
ALTER_DISKGROUP command, 908
ASM compatibility level, 910
asmcmd managing files/directories within, 911
ASM_DISKGROUPS parameter, 905, 906, 907
changing ASM disk group attributes, 909–911
creating, 914–915
creating diskgroup directories for alias filenames, 917
creating new tablespace on, 916
disk repair time, 910
dropping disks and, 916
dropping files from, 918
failure groups, 914
managing, 913
mirroring for redundancy, 914
no diskgroups mounted error, 906
RDBMS compatibility level, 910
rebalancing disk groups, 916
redundancy levels, 909
specifying allocation unit (AU) sizes, 909
striping for performance, 914
template redundancy, 910
template striping, 910
ASM files, 901, 902, 916, 917–918
ASM instances, 900, 901, 902, 903, 904–908
Database Control managing, 907, 914
shutting down, 908, 910
ASM mirroring, 901, 909, 914
ASM rebalance (ARBn) processes, 185
asmcmd command-line tool, 911–913
md_backup command, 911, 912, 913
md_restore command, 911, 912, 913
requiring ASM instances, 911
ASM_DISKGROUPS parameter, 905, 906, 907
ASM_DISKSTRING parameter, 905
ASM_POWER_LIMIT parameter, 905, 916
ASM_PREFERRED_READ_FAILURE_GROUPS parameter, 909
ASM-related background processes, 185
AS_OF clause, Flashback Data Archive, 873–874
ASSIGN_ACE procedure, 616
asterisk (*) character, SQL, 1238
asynchronous I/O, DBWn process, 182
at (@) sign, SQL*Plus, 124, 125
at command, UNIX/Linux, 78
at scheduling utility, Windows, 126
ATO (Automatic Tuning Optimizer), 1111–1113, 1115
atomicity property, transactions, 340
ATTACH parameter
Data Pump Export utility, 688, 700, 701, 702, 703
ATTRIBUTE clause, ASM, 909–911
attributes, relational database model, 20
entity-relationship (ER) modeling, 25, 27
AUD$ table, 611
audit files, default location for, 587
AUDIT SESSION statement, 553, 589
audit trails, 596, 587
audit_column parameter, 594
audit_column_opts parameter, 594, 595
audit_condition parameter, 594
AUDIT_FILE_DEST parameter, 451, 587, 588
auditing
audit levels, 587
autonomous transactions, 382
database auditing, 612
database security, 611
database usage, 586–596
DBA_AUDIT_TRAIL view, 588
default location for audit file, 587
fine-grained auditing, 593–596
limiting data written to audit trail, 587
NOAUDIT keyword, 589
standard auditing, 587–593
SYS.AUD$ table, 588, 596
audit-related parameters, 450–451
AUDIT_SYS_OPERATIONS parameter, 451, 458, 588, 612
AUDIT_TRAIL parameter, 450, 587, 588, 611
audit_trail parameter, 594
AUM (Automatic Undo Management), 200, 356–362, 921
creating default undo tablespace, 460
reducing buffer busy wait events, 1176
snapshot-too-old error, 364
undo segments, 361
UNDO_MANAGEMENT parameter, 357
UNDO_RETENTION parameter, 359–362
UNDO_TABLESPACE parameter, 357–359
authentication
see also passwords; security
database authentication, 596–601
connecting to RMAN, 745–746
locking accounts, 598
deny remote client authentication, 615
external authentication, 601–602
password authentication, 602
proxy authentication, 602
users, 596–602
AUTHENTICATION_SERVICES parameter, 526
AUTHID clause, CREATE PROCEDURE, 573
authorization
see also database access
centralized user authorization, 602
database creation, 446
role authorization, 575–576
AUTO option/parameter
segment space management, 217, 219, 220
AUTO value, OPTIONS attribute
GATHER_DATABASE_STATS procedure, 1055
AUTOALLOCATE option
creating tablespaces, 222
extent sizing, 216, 219, 220
locally managed tablespaces, 219
managing extent sizes, 237
tablespace storage parameters, 221
temporary tablespaces, 233
AUTOBACKUP option
CONTROLFILE parameter, RMAN, 765, 770
AUTOCOMMIT variable, 107, 133, 339
AUTOEXTEND clause, 224, 227, 230, 238, 358, 362
automated maintenance tasks, 1019–1022
automated tasks feature, 211
automatic checkpoint tuning, 183, 932–933
automatic consumer group switching, 555, 560
Automatic Database Diagnostic Monitor
see ADDM
automatic database management, 208–209
automatic database startup, 499–501
Automatic Diagnostic Repository see ADR
Automatic Disk-Based Backup and Recovery, 734
Automatic Maintenance Tasks page, 488
automatic memory management, 195–196, 894–897
memory-related initialization parameters, 457
Automatic Optimizer Statistics Collection, 209, 212, 213, 897–899
providing statistics to CBO, 1053
automatic performance tuning, 1131–1132
automatic PGA memory management, 194, 894
automatic secure configuration, 611
Automatic Segment Advisor, 212
Automatic Segment Space Management, 218, 928, 1173, 1176
automatic service registration, 521–522
Automatic Shared Memory Management, 894, 1178
automatic space management, 921–933
see also space management
Automatic SQL Tuning Advisor, 209, 212, 1115–1120
SQL plan baselines, 1081
Automatic Storage Management see ASM
Automatic Undo Management see AUM
automatic undo retention tuning, 209
Automatic Workload Repository see AWR
AUTOMATIC_ORDER
transforming array data with rules, 670
autonomous transactions, Oracle, 380–382
AUTO_SAMPLE_SIZE procedure, 1055
AUTO_SPACE_ADVISOR_JOB, 931
Autotask Background Process (ABP), 1022
AUTO_TASK_CONSUMER_GROUP, 558
Autotrace utility, SQL, 1095–1099
auxiliary database
using RMAN for TSPITR, 840, 841
AUX_STATS$ table, 1060
availability
benefits of archivelog mode, 726
disk configuration strategies, 87
Availability page, Database Control, 146
Available Targets, Database Control roles, 150
Average Active Sessions chart, 1199
AVERAGE_WAIT column,
V$SYSTEM_EVENT, 1166
AVG function, SQL, 1229
AVG_XYZ_TICKS statistic, 1182
avm, vmstat utility, 82
AWR (Automatic Workload Repository), 210, 947, 959–971
ADDM and, 878
automatic performance tuning features, 1132
baseline metrics, 953
baseline templates, 965–971
configuring ADDM, 882
controlling volume of statistics collected, 882
data handling, 960
data retention period, 960
determining number of undo segments, 359
DISPLAY_AWR function, 1092
loading SQL plans manually, 1081
managing AWR statistics with data dictionary views, 971
moving window baselines, 965
performance statistics, 877, 878
performance statistics for SQL statements, 1184
performance statistics formats, 959
statistics retention period, 960
storage space requirement, 960, 964
time-model statistics, 880

types of data collected by AWR, 960

AWR page, Database Control, 962

AWR reports, 966–971, 1201

AWR snapshots, 878, 959

comparing, 961

CREATE_SNAPSHOT procedure, 961

creating/deleting snapshot baselines, 963–964

Database Control managing, 961–963

DBMS_WORKLOAD_REPOSITORY managing, 961

DROP_SNAPSHOT procedure, 961

managing, 961

preserved snapshot set, 963

purging, 964–965

retention time period, 964

running ADDM, 885

setting snapshot interval to zero, 961

snapshot interval, 964

Time Periods Comparison feature, 1206

AWR_REPORT_HTML function, 967

AWR_REPORT_TEXT function, 967

awrrpt.sql script, 966, 967, 971

awrsqrrpt.sql script, 1184

B

background CPU time

V$SESS_TIME_MODEL view, 1206

background processes, 179, 180–186

archiver (ARChn) process, 181, 184

ASM background (ASMh) process, 185

ASM rebalance (ARBh) processes, 185

change-tracking writer (CTWR), 185

checkpoint (CKPT) process, 181, 183

database writer (DBWh) process, 180, 181–182

flashback data archiver (FBDA), 186

getting complete list of, 184

job queue coordination (CJQO) process, 181, 185

key Oracle background processes, 180

lock (LCKn) process, 186

log writer (LGWR) process, 180, 182–183

manageability monitor (MMON) process, 181, 185

manageability monitor light (MMNL) process, 181, 185

memory manager (MMAN) process, 181, 185

process monitor (PMON) process, 181, 183

rebalance master (RBAL) process, 185

recovery (RECO), 185

recovery writer (RVWR) process, 185

result cache background (RCBG), 186

system monitor (SMON) process, 181, 184

viewing all available processes, 186

background processes, UNIX, 75

BACKGROUND_DUMP_DEST parameter, 178

Backout feature see Flashback Transaction

Backout feature

BACKUP command, 792

BACKUP commands, RMAN, 755–757

backing up online with scripts, 775

BACKUP ARCHIVELOG ALL, 775

BACKUP AS BACKUPSET, 754

BACKUP AS BACKUPSET DATABASE, 755

BACKUP AS COMpressed

BACKUPSET, 780

BACKUP AS COPY, 735, 752, 754, 756, 757

BACKUP BACKUPSET, 754

BACKUP CONTROLFILE, 785

BACKUP CURRENT CONTROLFILE, 776

BACKUP DATABASE, 742, 755, 756, 774, 777

BACKUP DATABASE PLUS

ARCHIVELOG, 775

BACKUP DATAFILE, 777

BACKUP INCREMENTAL LEVEL, 778

BACKUP RECOVERY AREA, 739, 741

BACKUP RECOVERY FILES, 739

BACKUP TABLESPACE USERS, 777

BACKUP VALIDATE, 783, 784

cumulative backup, 756, 757

differential backup, 756, 757

DURATION clause, 777

FORMAT clause, 754

incremental backups, 756–757

KEEP clause, 780

KEEP FOREVER clause, 780, 781

KEEP UNTIL TIME clause, 780, 781

making multiple copies of backup sets, 754

RESTORE POINT clause, 780

resynchronizing recovery catalog, 769

specifying limits for backup duration, 777–778

specifying multiple copies, 754

backup files, RMAN, 178

backup formats, RMAN, 753

backup levels, 728

BACKUP OPTIMIZATION parameter, RMAN, 765

backup pieces, RMAN, 752, 754

backup retention policy parameters, RMAN, 762–763

backup schedules, 733

backup sets, RMAN, 752

BACKUP DATABASE command, 755

making multiple copies of, 754

RMAN storing metadata, 744

VALIDATE BACKUPSET command, 761, 783

Backup Solutions Program (BSP), 745

backup tags, RMAN, 753

backup, DBA role, 3, 6, 95

backups, 728–730

see also flash recovery area; recovery; RMAN; RMAN backups

architecture, 201–202

archivelog mode, 726–727
backing up control file, 784–785
backing up Oracle databases, 725–734
backup levels, 728
backup schedules, 733
benefits of tablespaces, 171
change-tracking writer (CTWR) process, 185
classifying backup media, Oracle Secure Backup, 789
cloning databases, 833–840
closed/cold backups, 728
consistent backups, 727
control file, 201
datafile backups, 728
disaster recovery, 798–800
Flashback Data Archive, 870–874
flashback recovery techniques, 202
Flashback techniques, 847–861
frequency of backups, 732
HARD initiative, 798
inconsistent backups, 727
incremental backups, 732, 733
incrementally updated backups, RMAN, 778
logical backups, 728
loss of data since previous backup, 726
manual database upgrade process, 437, 439
noarchivelog mode, 726–727
open backups, 726
open/warm/hot backups, 728
Oracle Secure Backup, 202, 785–790
partial database backups, 727
physical backups, 728
RAID, 92
redo logs, 201
redundancy set, maintaining, 730–731
RMAN, 774
service level agreements (SLAs), 731–732
setting record keep time in control file, 454
SHUTDOWN ABORT command, 504
strategies, 731–734
system change number (SCN), 200, 727
tablespace backups, 728
terminology, 726
testing backups, 730
undo records, 201
UNIX utilities, 76–77
upgrading with DBUA, 432, 433
user-managed backups, 201, 790–795
ways of performing physical backups, 725
whole database backups, 727, 728
BAD FILE parameter, 648
BAD parameter, SQL*Loader, 635
balanced tree index see B-tree indexes
bandwidth
performance monitoring, UNIX, 81
base recovery catalog, RMAN, 772
baselines
AWR baseline templates, 965–971
AWR snapshots, 961, 963–964
baseline metrics, 953–959
CREATE_BASELINE procedure, 963
DROP_BASELINE procedure, 964
performance statistics, 948
SQL plan baselines, 1080–1085
BASH (Bourne Again Shell), 45
see also shells, UNIX
.bash_profile file, 55, 410, 413
.bashrc file, 55
BASIC value, STATISTICS_LEVEL parameter, 462
batch command, UNIX/Linux, 78
batch jobs, 554
batch mode, UNIX, 54
BATCH option, 339
batch script, Windows, 126
BATCH_GROUP resource consumer group, 558
before-image records, 176, 197, 198, 356
BEGIN BACKUP command, 792
BEGIN statement, PL/SQL, 1241, 1242
BEGIN_SNAPSHOT parameter, ADDM, 883
benefits, ADDM findings, 880
BETWEEN operator, SQL, 1227
BFILE data type, 1222
BFT (bigfile tablespace), 172, 236–238
bin directory, ORACLE_HOME, 395
bin directory, UNIX, 48
binary compression feature, RMAN, 742
BINARY data types, 1222
binary dumps, data blocks, 167
binary files, 253, 254
binary large objects (BLOBs), 40
binary operations, relational algebra, 21
bind array, SQL*Loader, 634
bind peeking technique, parsing, 1087
bind variables
converting hard parse to soft parse, 1141
cursor sharing using, 1087
efficient SQL, 1075
identical SQL statements, 1134
optimizing library cache, 1138–1139
reducing parse time CPU usage, 1157
binding, 343
adative cursor sharing, 1087, 1088, 1089
bind-sensitive cursors, 1088
BINDSIZE parameter, SQL*Loader, 634
bitmap indexes, 301, 1071
bitmap join indexes (BJIs), 1069
BITMAP keyword, CREATE INDEX, 301
INDEX

BITMAP_AREA_SIZE parameter, 194
bitmaps, 172, 217
BJI (bitmap join index), 1069
Blackouts page, Database Control tool, 148
BLOB data type, 1222
block checking, 470, 471
block corruption, 167
block dumps, 167
block media recovery (BMR), 742, 808, 864–865
BLOCK option, RECOVER command, 864
block special files, UNIX, 57
block-change tracking, RMAN, 779
BLOCKED status, Oracle listener, 522
blocking locks, 351–352
blocking sessions, 354
BLOCKRECOVER command, RMAN, 864
blocks see data blocks
BLOCKSIZE clause, 223
BLOCKTERMINATOR variable, SQL, 130, 132
Boolean operators, SQL, 1227
Bouncing the database, 448
Bourne Again Shell (BASH), 45
Bourne shell, 45
Boyce-Codd normal form (BCNF), 33
bps column, iostat command, 82
branch blocks, B-tree index, 298
bread column, sar command, 83
BREAK command, SQL*Plus, 122
CLEAR BREAKS command, 115
BSP (Oracle Backup Solutions Program), 745
BTITLE command, SQL*Plus, 123, 124
B-tree indexes, 298, 301
BLEVEL column, 333
index organized tables, 278
using appropriate index types, 1071
buffer busy wait events, 1175–1177
buffer cache, 187, 188–189
aging out blocks, 1145
assign table or index to, 1147
buffer busy wait events, 1175
buffer cache hit ratio, 1144, 1145
buffer gets, 1144
consistent gets, 1144
database writer (DBWn) process, 199
DB block gets, 1144
description, 456
faster instance startup, 805
high buffer cache hit ratio, 1161
hit ratio, 190
how Oracle processes transactions, 197
logical reads, 1144
multiple database block sizes and, 189–190
physical reads, 1144
sizing, 1144–1145
increasing buffer cache size, 1145
nonstandard-sized, 456, 458
optimal buffer cache size, 1145
standard-sized, 456
specifying values for subcaches, 190
total size of, 190
tuning, 1144–1148
using multiple pools for buffer cache, 1146–1148
buffer cache pools, 188–189
behavior of, 457
keep pool, 189, 457
listing, 1145
main types, 189
recycle pool, 189, 457
setting size of default buffer pool, 457
using multiple block size feature, 223
buffer gets, 1144
buffer_gets column, V$SQL view, 1108
buffers
CLEAR BUFFER command, SQL*Plus, 115
dirty buffers, 188
free buffers, 188
least recently used (LRU) algorithm, 188
memory buffers, 187, 188
pinned buffers, 188
redo log buffer, 176, 187, 192–193
saving SQL buffer contents to file, 124
BUILD DEFERRED/IMMEDIATE clauses
CREATE MATERIALIZED VIEW statement, 319
BUILD procedure, DBMS_LOGMNR_D, 844
Burleson Consulting, 14
Business Copy XP, 746
business rules, 36, 37
application logic or integrity constraints, 306
BUSY_TICKS system usage statistic, 1181
bwrit column, sar command, 83
BYDAY/BYHOUR/BYXYZ keywords, jobs, 999
BYPASS procedure, 1123, 1125
bytes remaining tablespace alert, 226, 227
C
C shell, 45, 55
see also shells, UNIX
cache
buffer cache pools, 188–189
cache misses affecting performance, 192
Client Query Result Cache, 1125–1126
CLIENT_RESULT_CACHE_XYZ parameters, 458
creating cacheable function, 1124
creating SQL cache, 321
data block sizes and tablespaces, 171
data dictionary cache, 191
DB_XYZ parameters, 457, 458
library cache, 191
measuring library cache efficiency, 1137–1138
PL/SQL Function Result Cache, 1124–1125
result cache, 192, 1120–1126
result cache background (RCBG) process, 186
RESULT_CACHE_XYZ parameters, 464
SQL Query Result Cache, 1124
cache buffer chain latch free wait, 1180
cache recovery, 804, 806
Cache Sizes section, AWR reports, 968
calculus, relational, 22
calendar expression, Scheduler, 999
cancel-based recovery, 824
CANCEL_REPLAY procedure, 1215
CANCEL_SQL switch group, 561, 555
candidate keys, 26
CAN_REDEF_TABLE procedure, 937
capacity planning, DBA role, 6
capture process, Oracle Streams, 671
cardinality, ER modeling, 26, 28
caret (^) character, SQL, 1238
Cartesian product/join, SQL, 21, 1045, 1232
CASCADE CONSTRAINTS clause
DROP TABLESPACE statement, 225
CASCADE clause
DROP PROFILE statement, 553
DROP TABLE statement, 276
DROP USER statement, 548, 853
GATHER_DATABASE_STATS procedure, 1055
TRANSACTION_BACKOUT procedure, 380, 869
case command, UNIX/Linux, 74
case statement, SQL, 1066, 1230
cat operator, abstract data types, 1241
cat command, UNIX/Linux, 52, 56, 58
CATALOG command, RMAN, 752, 759, 768, 770
CATALOG START WITH command, 760, 770
catalog.sql script, 485, 486
data dictionary creation, 204
cataloging backups, RMAN, 770
catblock.sql script, 353
catdwgrd.sql script, 434, 442
catproc.sql script, 485, 486
catupgrd.sql script, 434, 438, 440
catuppst.sql script, 438, 439
CBO (Cost-Based Optimizer), 205, 1047–1053
  application developer knowledge, 1053
  automatic optimizer statistics collection, 209
  Automatic Tuning Optimizer (ATO), 1111
  Autotrace utility, 1098, 1099
  cost-based query optimization, 1044–1046
  cost model of optimizer, 1060
  dependence on correct statistics
  gathering, 1053
  drawbacks of CBO, 1053
  heavy data skew in table, 1066
  how CBO optimizes queries, 1051–1053
  IN list, 1066
  inefficiently written queries, 1065
  materialized views, 314
  normal mode, 1111
  Oracle version differences, 1053
  plan stability feature, 1077
  providing statistics to CBO, 1053–1056
  providing statistics to optimizer, 1047–1049
  query processing optimization phase, 1043
  query rewriting, 315
  rule-based optimization compared, 1047
  selectivity, 1065
  setting optimizer level, 1050–1051
  setting optimizer mode, 1049–1050
  specifying use of pending statistics, 463
  stored outlines improving SQL processing, 1077–1080
  storing optimizer statistics, 1054
  TKPROF utility output, 1104
  tuning mode, 1111, 1115
  understanding logic of, 1090
  views in query, 1065
WHERE clauses, 1065
cd command, UNIX/Linux, 48, 62
CDs
  installing Oracle software using staging
directory, 416
  Oracle Enterprise Edition CDs, 414–415
  using explicit command to load, 414
centralized configuration, Oracle Net
  Services, 512
centralized user authorization, 602
certification, DBA, 10, 11–13
chained rows, Flashback Transaction
  Query, 374
caches, Oracle Scheduler, 995, 996, 1008–1010
CHANGE command, SQL*Plus, 129
change management, 212–213
DBA role, 7
  Oracle Change Management Pack, 149, 949
  Schema page, Database Control, 147
change vectors, redo log files, 176
change-based SCN
  incomplete recovery using RMAN, 820
  user-managed incomplete recovery, 824
  change-tracking file, RMAN, 779
  change-tracking writer (CTWR) process, 185
  channel configuration parameters, RMAN, 764
  CHANNEL parameter, RMAN, 764
  channels, RMAN, 753
  CHAR/character data types, 1222
  character large objects (CLOBs), 40
  character sets, 481, 488
  character special files, UNIX, 57
  CHECK constraint, 37, 307, 313
  CHECK LOGICAL clause
  VALIDATE command, RMAN, 784
  CHECK_OBJECT procedure, 797
  checkpoint (CKPT) process, 175, 181, 183, 479
  CHECKPOINT clause, 271
  checkpoint completed wait event, 1177
  “checkpoint not complete” messages, 1188
  checkpoints
    automatic checkpoint tuning, 183, 932–933
    Fast Start Checkpointing, 805
  CHECK_PRIVILEGE function, 617
checksumming, detecting corruption, 470, 796
CHECKSYNTAX parameter, RMAN, 749
INDEX

Chen, Peter, 25
chgrp command, UNIX/Linux, 62
child nodes, B-tree index, 298
child processes, UNIX, 54
Childs, D.L., 20
chmod command, UNIX/Linux, 60, 61, 70
Choosing Database Backup Procedure window, DBUA, 432
chsh command, UNIX/Linux, 46
classes, object-oriented database model, 39
CLEAR command, SQL*Plus, 115
clear_pending_area procedure, 556
client authentication, 615
client host server, administrative domain, 786
client processes, Data Pump, 686
Client Query Result Cache, 1125–1126
client software
  Instant Client software, 519–520
  Oracle Client software, 517–519
client/server model, database connectivity, 511
CLIENT_ID_TRACE_ENABLE attribute, 1106
CLIENT_RESULT_CACHE view, 1126
CLIENT_RESULT_CACHE_XYZ parameters, 458, 1126
CLOB data type, 1222
cloning databases, 833–840
  Database Control, 148, 838–839
  Grid Control, 148
  Oracle software cloning, 148
closed backups, 728
  whole closed backups, 790–791
closed recovery, 807
CLOSE_WINDOW procedure, 1015
CLUSTER clause, CREATE TABLE, 295
Cluster Synchronization Service see CSS
Cluster wait class, 1163
clustered tables, 266
clusters, 295–296
  hash clusters, 296
  Oracle Real Application Clusters (RACs), 173
CMON (Connection Monitor) process, 532
COALESCE function, SQL, 1230
COALESCE option, ALTER INDEX, 935
COALESCE PARTITION command, 292
taoesing indexes online, SQL, 935
Godd, E.F., 20, 21, 29
cold backups see closed backups
collection types, ORDBMS model, 40
collections, abstract data types, 1240
COLSEP variable, SQL*Plus, 107
COLUMN command, SQL*Plus, 123
column groups, 1058, 1059
column specifications, data types, 36
COLUMNARRAYROWS parameter, SQL*Loader, 641
column-level object privileges, 572
column-level security, 312
column-level VPD, 585–586
columns
  adding to/dropping from tables, 271
  CLEAR COLUMNS command, 115
  creating tables, 265
  DBA_CONS_COLUMNS view, 311
  DBA_IND_COLUMNS view, 333
  DBA_TAB_COLUMNS view, 293, 332
t  default values for, 270
  indexing strategy, 1071
  listing table columns and specifications, 119
  ordering of columns in tables, 20
  partitioning, 281
  renaming, 272
  setting as unused, 271
  showing properties of, 123
  virtual columns, 270–271
COMMAND column, top command, 84
command files, SQL*Plus, 124–129
command interpreters, UNIX, 46
command line
  Data Pump Export utility using, 687
  command line arguments, UNIX
    executing shell scripts with, 70
  command line parameters, SQL*Loader, 633–636
command line utilities
  Data Pump components, 680
  command-line options, SQL*Plus, 113–115
commands
  listener commands, 522–523
  operating system file executing RMAN commands, 749
commands, list of see SQL*Plus commands, list of
commands, SQL*Plus see SQL*Plus commands, list of
commands, UNIX see UNIX commands
COMMENT parameter
  creating resource consumer groups, 557
  comments
    adding comments to scripts, SQL*Plus, 132
    init.ora file, 496
    SPFILE (server parameter file), 496
    using comments in SQL*Plus, 128
COMMENT parameter, CREATE_JOB procedure, 999
commit method, JDBC conn, 540
COMMIT statement, 133, 197, 263, 338–339, 1242
Commit wait class, 1163
COMMIT_LOGGING parameter, transactions, 339
committing transactions, 196, 197–198
  fast commit mechanism, 183
  log writer (LGWR) process, 199
  redo log buffer, 182
COMMIT_WAIT parameter, transactions, 339
common manageability infrastructure, 210–213
communication protocol, connect descriptors, 515
compaction phase, segment shrinking, 929
Companion CD, Oracle Enterprise Edition, 414
COMPARE function, DBMS_COMPARISION package, 989
Compare Periods Report, Database Control, 1206–1208
COMPARE_PERFORMANCE parameter, 1219
comparison operators, SQL, 1227
WHERE clause, 1224
COMPATIBLE parameter, 452
database compatibility level, 429
Pre-Upgrade Information Tool, 428
setting up Oracle Streams, 673
compensation transactions, 868
COMPLETE option, materialized views, 316
complete recovery, 807
composite indexes, 297, 298, 1072
composite keys, 31, 334
composite partitioning, 281, 287–290
COMPOSITE_LIMIT parameter, 549
comprehensive analysis mode, Segment Advisor, 930
COMPRESS clause, ALTER TABLE, 275, 276
COMPRESS keyword, 1076
compressed backups, RMAN, 780
compression
key-compressed indexes, 301
table compression, 274–276
tablespace compression, 274
COMPRESSION parameter
Data Pump Export utility, 691
populating external tables, 652
RMAN, 764
compression techniques, SQL tables, 1076
COMPUTE command, SQL*Plus, 123
COMPUTE STATISTICS option
ANALYZE command, 1055
CREATE INDEX statement, 299
CONCAT function, SQL, 1228
CONCATENATE clause, SQL*Loader, 630
concatenated indexes, 297, 1072
concurrency see data concurrency
Concurrency wait class, 1163
conditional branching, UNIX, 71–72
conditional control, PL/SQL, 1243
conditional functions, SQL, 1230
configuration
database management, 146, 148
tool-wide with Grid Control, 158
Oracle Configuration Management Pack, 149, 949
RMAN configuration parameters, 761–766
Setup page, Database Control tool, 148–149
Configuration Assistants window, 155
Configuration Manager
products installed with 11.1 release, 401
Configuration Options window, 419
CONFIGURE command, RMAN, 762
BACKUP COPIES option, 754
configuration parameters, 761–766
default device types, 753
CONFIGURE procedure, DBMS_SPM package, 1085
CONFIGURE_POOL procedure, 533
CONNECT BY clause, SQL, 1232
CONNECT CATALOG command, RMAN, 767
CONNECT command, SQL*Plus, 100
connect descriptors, Oracle networking, 514
connect identifiers, Oracle networking, 515
net service names, 525
CONNECT privilege, 616
CONNECT role, 430, 574
connect strings, Oracle networking, 515
TWO_TASK environment variable, 519
CONNECT_IDENTIFIER variable, SQL*Plus, 118, 119, 127
connection architecture, 512
connection broker, DRCP, 180
connection management call elapsed time, 1206
Connection Manager feature, 512
Connection Mode tab, DBCA, 488
Connection Monitor process (CMON), 532
connection naming
directory naming method, 534–537
easy connect naming method, 529–530
external naming method, 533–534
local naming method, 525–529
connection pooling, 180, 531–533
connectionless SQL*Plus session with
NOLOG, 101
connections
concurrent connection requests, 523
connecting to RMAN, 745–746
database links, 985–987
naming, 525
operating system authentication method, 99
securing network, 614
starting SQL*Plus session from command line, 98–100
CONNECTION_TIME_SCALE parameter, 1214
connectivity, 511
see also Oracle networking
database resident connection pooling(DRCP), 531–533
establishing Oracle connectivity, 516–517
Instant Client software, 519–520
Java Database Connectivity see JDBC
naming, 525
net service names, 525
Oracle Client software, 517–519
Oracle Internet Directory (OID), 535
Oracle listener see listeners
Oracle Net Services, 511–512, 516
Oracle networking, 513–516
web applications connecting to Oracle, 513
INDEX

CONNECT_TIME parameter, 549
CONNECT_TIME_FAILOVER parameter, 523
consistency see data consistency
consistency property, transactions, 340
consistent backups, 727
consistent gets, 1144
CONSTANT parameter, SQL*Loader, 636
constraints
built-in database constraints, 37
CASCADE CONSTRAINTS clause, 225
CHECK constraint, 307
DBA_CONS_COLUMNS view, 311
DBA_CONSTRAINTS view, 310
deferrable constraints, 310
disable VALIDATE command, 309
domain constraints, 37
dropping tables, 276
ENABLE VALIDATE command, 309
ENABLED VALIDATED constraints, 316
ensuring data integrity, 36
immediate constraints, 310
integrity constraints, 306–310
NOT NULL constraint, 306, 307
orderid_refconstraint, 285
primary key constraints, 306
referential integrity constraints, 225, 308
RELY constraints, 310
SQL*Loader direct-path loading, 641
temporary tables, 277
UNIQUE constraint, 307
CONSTRAINTS parameter, export utility, 692
consumer groups see resource consumer groups
CONSUMER_GROUP parameter, 557
consumption, Oracle Streams architecture, 671
CONTENT parameter
Data Pump Export utility, 692, 704
Data Pump Import utility, 708
custom context switches, 80
custom contexts, 536, 537
continuation characters, SQL*Plus, 102, 133
CONTINUE_CLIENT parameter, Data Pump, 703, 713
CONTINUEIF clause, SQL*Loader, 630
CONTINUOUS_MINE procedure, 847
control files, 173, 174–175
auto-backups, flash recovery area, 735
backing up control file, 784–785
backing up with RMAN, 765–766, 776
backup and recovery architecture, 201
backup guidelines, 729
checkpoint (CKPT) process, 175
correlation parameters contained, 447
creating database, 481
creating/locating OMF files, 251, 926
database files, 398
database integrity, 175
default file locations, 738
flash recovery area, 738
generating names of all control files, 175
managing RMAN, 744
multiplexing, 453, 455
naming conventions, 398
OMF file-naming conventions, 249, 923
Oracle Managed Files (OMF), 250, 924
creating/locating OMF files, 252
recovering from loss of control files, 824–828
RMAN repository data (metadata), 766
setting record keep time before overwriting, 454
specifying default location for OMF files, 455
SQL*Loader, 628–636
system change number, 174
$CONTROLFILE view, 175
whole closed backups, 790
whole database backups, 791
CONTROL parameter, SQL*Loader, 633
control structures, PL/SQL, 1243
CONTROLFILE parameter, RMAN, 765, 766
backing up recovery catalog, 770
BACKUP command, 785
CREATE command, 826, 827
RESTORE command, 825
CONTROL_FILE_RECORD_KEEP_TIME parameter, 454, 770
CONTROL_FILES parameter, 453
Oracle Managed Files (OMF), 250
controllers, disk I/O, 1159
CONTROL_MANAGEMENT_PACK_ACCESS parameter, 459, 882
conventional data loading
direct-path loading compared, 640
SQL*Loader, 627, 628, 639
CONVERGE procedure,
DBMS_COMPARISION, 990
conversion functions, Oracle data types, 1223
CONVERT TABLESPACE command, 721
COPY command, Oracle, 757
COPY command, RMAN, 758
resynchronizing recovery catalog, 769
COPY command, SQL*Plus, 132–133
copy command, Windows, 790
COPYCOMMIT variable, SQL*Plus, 107
COPY_FILE procedure, 253, 991
copying files, RMAN, 752–753, 754
copying files, UNIX, 59, 79
COPY_TABLE_DEPENDENTS procedure, 939
coraenv script, 424, 425
core directory, ADR, 178
correlated subqueries, SQL, 1237
corruption
data blocks, 167
database corruption detection, 795–798
detecting physical/logical corruption, RMAN, 783
enabling detection of, 470
monitoring and verifying RMAN jobs, 782
repairing datafiles with RMAN, 742
corruption-checking parameters, 470–472
Cost-Based Optimizer see CBO
cost-based query optimization, 1044–1046
COUNT function, SQL, 1229
cp command, UNIX/Linux, 59
physical database backups, 725, 790
cpio command, UNIX/Linux, 76, 77
CPU column, top command, 84
CPU method
creating plan directives, 560, 561
Database Resource Manager, 555
CPU performance, 1153–1158
CPU usage
causes of intensive CPU usage, 1153
cost model of Oracle optimizer, 1060
CPU units used by processes, 1154
determining session-level CPU usage, 1155
eliminating wait event contention, 1208
enforcing per-session CPU limits, 564–565
finding inefficient SQL, 1109, 1110
identifying high CPU users, 1154
parse time CPU usage, 1156–1157
performance monitoring, UNIX, 80
production database problems, 554
recursive CPU usage, 1157
reducing parse time CPU usage, 1157
run queue length, 1153
sar command output showing, 1153
SQL Trace tool showing, 1100
system performance, 1203
system usage problems, 1188
tuning shared pool, 1133
uses of CPU time, 1155–1158
V$SESSTAT view, 1203
CPU_MTH parameter, 557, 560
CPU_PER_CALL parameter, 549
CPU_PER_SESSION parameter, 549
CPUSPEED statistic, 1061
CPUSPEEDNW statistic, 1061
cpu_time column, V$SQL view, 1108
Crash recovery, Oracle, 804–805
CREATE ANY DIRECTORY privilege, 683
CREATE ANY TABLE privilege, 268
CREATE BIGFILE statement, 237
CREATE CATALOG command, RMAN, 768
CREATE CLUSTER statement, 295, 296
CREATE CONTROLFILE statement, 826, 827
CREATE DATABASE statement, 460–481
DEFAULT TABLESPACE clause, 236
SYSAUX clause, 239
create directory, 97
create directory, 397
CREATE DISKGROUP command, 915
CREATE FLASHBACK ARCHIVE statement, 871
CREATE GLOBAL SCRIPT command, RMAN, 750
CREATE INDEX statement, 299–300
BITMAP keyword, 301
B-tree index, 298
COMPUTE STATISTICS option, 299
GLOBAL PARTITION BY option, 302
INVISIBLE clause, 304
PARTITION BY HASH option, 303
PRIMARY KEY constraint, 300
CREATE JOB privilege, Scheduler, 997
CREATE MATERIALIZED VIEW statement, 318–319
CREATE option
SPOOL command, SQL*Plus, 120
STORE command, SQL*Plus, 116
CREATE OR REPLACE VIEW statement, 313
CREATE OUTLINE statement, 1079
CREATE PROCEDURE statement
AUTHID clause, 573
CREATE PROFILE statement, 548
Create Role Administrators page, Database Control, 150
Create Role Properties page, Database Control, 150
CREATE ROLE statement, 575, 576
CREATE SCHEMA statement, 264, 265
CREATE SCRIPT statement, RMAN, 750
CREATE SESSION privilege, 568, 569
creating users, 545
CREATE SYNONYM statement, 324, 326
CREATE TABLE privilege, 268
CREATE TABLE statement, 269
CLUSTER clause, 295
CREATE TABLE AS SELECT (CTAS) command, 273
creating external table layer, 647
ENABLE ROW MOVEMENT clause, 286
ENCRYPT clause, 269
FOREIGN KEY REFERENCES clause, 285
INCLUDING clause, 279, 280
INTERVAL clause, 283
LOGGING clause, 227
ORGANIZATION INDEX phrase, 279
PARTITION BY HASH clause, 283
PARTITION BY LIST clause, 284
PARTITION BY RANGE clause, 282
PARTITION BY REFERENCE clause, 285
PARTITIONED BY SYSTEM clause, 287
PCTTHRESHOLD clause, 279, 280
PRIMARY KEY constraint, 300, 306
STORE IN clause, 283
SUBPARTITION BY HASH clause, 288
SUBPARTITION BY LIST clause, 288
SUBPARTITION BY RANGE clause, 289, 290
UNIQUE constraint, 300
VALUES LESS THAN clause, 281
CREATE TABLESPACE statement, 218
AUTOEXTEND ON clause, 230
creating permanent tablespaces, 219
creating temporary tablespaces, 230–231
<table>
<thead>
<tr>
<th>Clause/Keyword</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCRYPTION clause</td>
<td>242, 610</td>
</tr>
<tr>
<td>EXTENT MANAGEMENT clause</td>
<td>230</td>
</tr>
<tr>
<td>EXTENT MANAGEMENT LOCAL clause</td>
<td>220</td>
</tr>
<tr>
<td>NOLOGGING clause</td>
<td>227</td>
</tr>
<tr>
<td>SIZE clause</td>
<td>230</td>
</tr>
<tr>
<td>TABLESPACE GROUP clause</td>
<td>233</td>
</tr>
<tr>
<td>TEMPORARY clause</td>
<td>230</td>
</tr>
<tr>
<td>UNDO keyword</td>
<td>358</td>
</tr>
<tr>
<td>UNIFORM SIZE clause</td>
<td>230</td>
</tr>
<tr>
<td>CREATE TRIGGER statement</td>
<td>328</td>
</tr>
<tr>
<td>CREATE TYPE statement</td>
<td>1239</td>
</tr>
<tr>
<td>CREATE UNDO TABLESPACE statement</td>
<td>218, 219</td>
</tr>
<tr>
<td>AUTOEXTEND keyword</td>
<td>358</td>
</tr>
<tr>
<td>RETENTION GUARANTEE clause</td>
<td>460</td>
</tr>
<tr>
<td>CREATE TRIGGER statement</td>
<td>328</td>
</tr>
<tr>
<td>CREATE UNQIUE INDEX statement</td>
<td>299</td>
</tr>
<tr>
<td>CREATE USER statement</td>
<td>544–547</td>
</tr>
<tr>
<td>DEFAULT TABLESPACE clause</td>
<td>547</td>
</tr>
<tr>
<td>IDENTIFIED BY clause</td>
<td>544</td>
</tr>
<tr>
<td>QUOTA clause</td>
<td>546</td>
</tr>
<tr>
<td>RMAN clause</td>
<td>767</td>
</tr>
<tr>
<td>CREATE VIEW statement/privilege</td>
<td>312</td>
</tr>
<tr>
<td>CREATE VIRTUAL CATALOG command</td>
<td>773</td>
</tr>
<tr>
<td>CREATE_ACL procedure</td>
<td>616</td>
</tr>
<tr>
<td>CREATE_ANALYSIS_TASK procedure</td>
<td>1218</td>
</tr>
<tr>
<td>CREATE_BASELINE procedure</td>
<td>963</td>
</tr>
<tr>
<td>CREATE_BASELINE_TEMPLATE procedure</td>
<td>965</td>
</tr>
<tr>
<td>CREATE_CHAIN procedure</td>
<td>1009</td>
</tr>
<tr>
<td>CREATE_COMPARISON procedure</td>
<td>988</td>
</tr>
<tr>
<td>create_consumer_group procedure</td>
<td>557</td>
</tr>
<tr>
<td>CREATE_CREDENTIALIAL procedure</td>
<td>1005</td>
</tr>
<tr>
<td>CREATE_DIAGNOSTIC_TASK procedure</td>
<td>1036</td>
</tr>
<tr>
<td>CREATE_EXTENDED_STATS function</td>
<td>1059, 1060</td>
</tr>
<tr>
<td>CREATE_INDEX_COST procedure</td>
<td>298, 299</td>
</tr>
<tr>
<td>CREATE_JOB procedure</td>
<td>998</td>
</tr>
<tr>
<td>attributes</td>
<td>999</td>
</tr>
<tr>
<td>events</td>
<td>1011</td>
</tr>
<tr>
<td>external jobs</td>
<td>1005</td>
</tr>
<tr>
<td>lightweight jobs</td>
<td>1001, 1002</td>
</tr>
<tr>
<td>programs</td>
<td>1007</td>
</tr>
<tr>
<td>CREATE_JOB_CLASS procedure</td>
<td>1012</td>
</tr>
<tr>
<td>create_pending_area procedure</td>
<td>556</td>
</tr>
<tr>
<td>CREATE_PLAN procedure</td>
<td>560</td>
</tr>
<tr>
<td>CREATE_PLAN_DIRECTIVE procedure</td>
<td>561, 565</td>
</tr>
<tr>
<td>CREATE_POLICY_GROUP procedure</td>
<td>584</td>
</tr>
<tr>
<td>CREATE_PROGRAM procedure</td>
<td>1006</td>
</tr>
<tr>
<td>CREATE_REPORT procedure</td>
<td>883</td>
</tr>
<tr>
<td>CREATE_SCHEDULE procedure</td>
<td>1007, 1008</td>
</tr>
<tr>
<td>CREATE_SNAPSHOT procedure</td>
<td>961</td>
</tr>
<tr>
<td>CREATE_SQLSET procedure</td>
<td>1217</td>
</tr>
<tr>
<td>CREATE_SQLWKLD procedure</td>
<td>323</td>
</tr>
<tr>
<td>createStatement method, JDBC conn</td>
<td>539</td>
</tr>
<tr>
<td>CREATE_STGTAB_SQLSET procedure</td>
<td>1218</td>
</tr>
<tr>
<td>CREATE_STORED_OUTLINES parameter</td>
<td>1078, 1079</td>
</tr>
<tr>
<td>CREATE_TABLE_COST procedure</td>
<td>268</td>
</tr>
<tr>
<td>CREATE_TASK procedure</td>
<td>323, 890, 978</td>
</tr>
<tr>
<td>CREATE_TUNING_TASK procedure</td>
<td>1113</td>
</tr>
<tr>
<td>CREATE_WINDOW procedure</td>
<td>1014, 1015</td>
</tr>
<tr>
<td>Creation Options window, DBCA, 488</td>
<td></td>
</tr>
<tr>
<td>CREDENTIAL_NAME attribute, Scheduler, 1005</td>
<td></td>
</tr>
<tr>
<td>Credentials window, Database Upgrade Assistant, 432</td>
<td></td>
</tr>
<tr>
<td>critical alert thresholds, 226, 952, 954, 956</td>
<td></td>
</tr>
<tr>
<td>Critical Patch Updates, 617</td>
<td></td>
</tr>
<tr>
<td>critical_value attribute, 227</td>
<td></td>
</tr>
<tr>
<td>crontab command, 77–78</td>
<td></td>
</tr>
<tr>
<td>CROSSCHECK command, RMAN, 758, 761, 783</td>
<td></td>
</tr>
<tr>
<td>CRSTCTL utility, 903</td>
<td></td>
</tr>
<tr>
<td>csh (C shell), 45, 55</td>
<td></td>
</tr>
<tr>
<td>.cshrc file, UNIX, 55</td>
<td></td>
</tr>
<tr>
<td>CSS (Cluster Synchronization Service), 902–904</td>
<td></td>
</tr>
<tr>
<td>CTAS (CREATE TABLE AS SELECT) command, 273</td>
<td></td>
</tr>
<tr>
<td>deriving data from existing tables, 657</td>
<td></td>
</tr>
<tr>
<td>LONG columns, 133</td>
<td></td>
</tr>
<tr>
<td>managing logging of redo data, 227</td>
<td></td>
</tr>
<tr>
<td>using with large tables, 132</td>
<td></td>
</tr>
<tr>
<td>writing to external tables, 651</td>
<td></td>
</tr>
<tr>
<td>CUBE operator, SQL, 1235</td>
<td></td>
</tr>
<tr>
<td>cumulative backup, RMAN, 756, 757</td>
<td></td>
</tr>
<tr>
<td>cumulative statistics, 948</td>
<td></td>
</tr>
<tr>
<td>current incarnation, database recovery, 823</td>
<td></td>
</tr>
<tr>
<td>current_user attribute, 580</td>
<td></td>
</tr>
<tr>
<td>currval pseudo-column, sequences, 327</td>
<td></td>
</tr>
<tr>
<td>cursors</td>
<td></td>
</tr>
<tr>
<td>bind-sensitive cursors, 1088</td>
<td></td>
</tr>
<tr>
<td>cursor sharing, 466, 1087–1090</td>
<td></td>
</tr>
<tr>
<td>description, 343</td>
<td></td>
</tr>
<tr>
<td>maximum number of cursors in session, 194</td>
<td></td>
</tr>
<tr>
<td>open cursors, 194, 456, 1141</td>
<td></td>
</tr>
<tr>
<td>preventing early deallocation of cursors, 1141</td>
<td></td>
</tr>
<tr>
<td>cursors, PL/SQL, 1245–1247</td>
<td></td>
</tr>
<tr>
<td>CURSOR_SHARING parameter, 466, 1087, 1138, 1139, 1141, 1209</td>
<td></td>
</tr>
<tr>
<td>CURSOR_SPACE_FOR_TIME parameter, 1140, 1141</td>
<td></td>
</tr>
<tr>
<td>Custom installation, Oracle software, 417</td>
<td></td>
</tr>
<tr>
<td>custom option, installing Oracle Client, 518</td>
<td></td>
</tr>
<tr>
<td>custom.rsp response file template, 422</td>
<td></td>
</tr>
<tr>
<td>cut command, UNIX/Linux, 66</td>
<td></td>
</tr>
</tbody>
</table>

**D**

Data
- database triggers ensuring validity of, 37
- modifying data on disk, 181
- separating table and index data, 170

Data access see database access

Data anomalies, 29

Denormalization, 34
data blocks, 166–169
  allocating to objects, 172
bigfile tablespace (BFT), 236
binary dumps, 167
block corruption, 167
block dumps, 167
buffer busy wait events, 1176
changing block size, 467
checking for corrupted data blocks, 471
creating tablespaces with nonstandard block sizes, 223
DB_BLOCK_SIZE parameter, 466
detecting data block corruption, 795–798
determining size of, 166–167
dumping contents, 167, 168
extents, 166, 169
free space section, 167
identifying file and block IDs, 168
inner workings, 167–169
keeping track of space in blocks, 172
multiple data block sizes, 167
multiple sizes and buffer cache, 189–190
online database block-size changes, 943–944
operating system disk block size, 166
querying data, 466
repairing corrupted data blocks, 864
row data section, 167
segment space management, 217–218
setting block size, 466
size units, 166
specifying maximum number read during full table scan, 467
tablespaces and block sizes, 171–172
using multiple block size feature, 223
data buffer cache see buffer cache
data center disasters, 802
data concurrency, 198–200, 341–342
  allowing DDL locks to wait for DML locks, 350
data consistency and, 346
dirty reads problem, 341
explicit locking in Oracle, 351–352
explicit table locking, 350–351
implementing concurrency control, 347–355
isolation levels, 342–346
isolation property, 340
locking, 341, 347, 348–350
locks affecting, 386
lost-update problem, 341
managing Oracle locks, 353–355
multiversion concurrency control system, 347
nonrepeatable (fuzzy) read problem, 342
phantom reads problem, 341
serializable schedules, 342
time-stamping methods, 347
validation methods, 347
data consistency, 198–200
read consistency, 199
redo log files, 176
system monitor (SMON) process, 184
transactions, 196, 340, 341, 342, 346
transaction-set consistent data, 199
undo management, 200
undo segments, 199
Workspace Manager, 386
data corruption, 864–866
data decryption, 604
data dictionary, 191, 203, 204
  extracting, LogMiner utility, 843
  monitoring database status, 507
  protecting, database security, 613
data dictionary cache, 191, 1134–1135
data dictionary locks, 351
data dictionary objects, 485
data dictionary tables, 203, 204
data dictionary views, 204
  see also DBA views; V$ views
ADDM related, 893
AWR snapshots, 959
INDEX_STATS view, 334
listing, 203
managing advisory framework, 980
managing AWR statistics with, 971
managing Database Resource Manager, 566
managing tables, 292–295
managing tablespaces, 243–246
managing undo space information, 365
managing users/roles/privileges, 577
metrics and alerts, 958–959
using dictionary views for SQL tuning, 1108, 1110
viewing object information, 329
data encryption, 603–608
  ENCRYPT keyword, 604
  encrypting table columns, 607–608
  encryption algorithms, 608
  generating master encryption key, 607
  Oracle Wallet, 604, 605–606
data extraction see ETL (extraction, transformation, loading)
Data Guard see Oracle Data Guard
data integrity, 36, 37
data loading see ETL (extraction, transformation, loading)
data manipulation statements, 313, 314
Data Migration Assistant, 427
data modeling, 25, 27, 38
Data Movement page, Database Control, 148
DATA pages, Oracle process component, 1190
DATA parameter, SQL*Loader, 634
data protection, 798–800
  Oracle Streams, 672
Data Pump utilities (Export, Import), 677
  accessing Data Pump utilities, 678
  API creating export/import jobs, 715
  attaching to running Data Pump job, 679, 688
  benefits of Data Pump technology, 678–679
  compatibility, 677
components, 680
correcting human error, 802
Data Pump technology, 677–686
data-access methods, 680–681
DBMS_DATAPUMP package, 680
DBMS_METADATA package, 680
description, 808
directory objects, 681–684
estimating space requirements, 679
Export parameters, 689–704
  ADD_FILE, 702, 703, 704
  ATTACH, 700, 701, 702, 703
  COMPRESSION, 691
  CONTENT, 692, 704
  CONTINUE_CLIENT, 703
  DATA_OPTIONS, 694
  DIRECTORY, 690
  DUMPFILE, 690, 705
  encryption parameters, 694–696, 698
  estimate parameters, 696–697
  EXCLUDE, 692–693, 704
  EXIT_CLIENT, 703, 704
  export filtering parameters, 692–694
  export mode parameters, 691
  file and directory parameters, 690–691
  FILESIZEx, 690, 705
  FLASHBACK_SCN, 699
  FLASHBACK_TIME, 700
  FULL, 688
  HELP, 703, 704
  INCLUDE, 692–693
  interactive mode parameters, 701–704
  job parameters, 699–701
  JOB_NAME, 699, 705
  KILL_JOB, 703, 704
  LOGFILE, 690
  NETWORK_LINK, 698, 717
  NOLOGFILE, 691
  PARALLEL, 700–705
  PARFILE, 690
  QUERY, 694, 704
  REMAP_DATA, 693
  REUSE_DUMPPFILE, 691
  SAMPLE, 694
  SCHEMAS, 688, 705
  START_JOB, 703, 704
  STATUS, 699, 703, 704
  STOP_JOB, 702, 703, 704
  TABLES, 688
  TABLESPACES, 688
  TRANSPORTABLE, 694
  TRANSPORT_FULL_CHECK, 691, 717, 721
  TRANSPORT_TABLESPACES, 688, 718
  USERID, 718
Export utility, 207
  backup guidelines, 730
  creating export dump file, 704
dpdump directory containing files, 397
export prompt, interactive mode, 688
exporting metadata using, 721
exporting tables from schema, 705
generating transportable tablespace
  set, 717
initiating network export, 698
interactive commands, 703
logical backups, 728
methods, 687–688
modes, 688–689
read-only database, 698
schema mode, 688, 705
using parameter file, 704
files, 681–685
fine-grained data import capability, 679
Import parameters, 705–713
  CONTENT, 708
  CONTINUE_CLIENT, 713
  DATA_OPTIONS, 710
  DIRECTORY, 706
  DUMPFILE, 706
  EXCLUDE, 708
  EXIT_CLIENT, 713
  file and directory parameters, 706–707
  filtering parameters, 708
  FLASHBACK_XYZ parameters, 713
  FULL, 708
  HELP, 713
  import mode parameters, 708–709
  INCLUDE, 708
  interactive import parameters, 713
  job parameters, 708
  JOB_NAME, 708
  KILL_JOB, 713
  LOGFILE, 706
  NETWORK_LINK, 711–713
  NOLOGFILE, 706
  PARALLEL, 708, 713
  PARFILE, 706
  QUERY, 708
  remapping parameters, 709–711
  REUSE_DATAFILES, 707
  SCHEMAS, 708
  SQLFILE, 706–707
  START_JOB, 713
  STATUS, 708, 713
  STOP_JOB, 713
  TABLE_EXISTS_ACTION, 708
  TABLES, 708
  TABLESPACES, 708
  TRANSFORM, 710–711
  TRANSPORTABLE, 710
  TRANSPORT_XYZ parameters, 705, 708
Import utility, 207
  disabling constraints, 308
  import modes, 705
  import prompt, interactive mode, 688
  import types, 705
  importing metadata using, 723
  performing transportable tablespace
  import, 719–720
SQLFILE parameter, 681
managing/monitoring database, 213  
manual upgrade process, 427  
mechanics of Data Pump job, 685–686  
metadata filtering, 692  
monitoring Data Pump jobs, 713–714  
network mode of operation, 679  
nonprivileged users using, 683  
order of precedence for file locations, 684–685  
parallel execution, 678  
performance, 678  
performing exports and imports, 686–713  
privileges, 685  
processes, 685–686  
restarting jobs, 678  
roles, 574  
tasks performed by Data Pump technology, 677  
transportable tablespaces, 171, 716–723  
uses for, 679–680  
using Data Pump API, 715  
Data Recovery Advisor, 211, 829–833, 1023  
ADVISE FAILURE command, 831  
data block corruption, 167  
data repair, 803  
failure properties, 830  
LIST FAILURE command, 830  
REPAIR commands, 832, 833  
data redefinition online, 935–941  
data reorganization online, 933–935  
Database Control, 933  
SQL commands performing, 934–935  
Data Reorganization page, Database Control, 934  
data replication, Oracle Streams, 670  
data storage technologies, 93–95  
RAID systems for disks, 88–93  
data transfer element (dte), 789  
data transformation see transforming data  
data types  
abstract data types, 1239–1241  
column specifications, 36  
Oracle data types, 1222–1223  
ORDBMS model, 40  
SQL*Loader control file, 632  
user-defined object types, 264  
XMLType, 1249  
data warehousing  
Automatic Workload Repository (AWR), 210  
bitmap join indexes (BJI), 1069  
building, 27  
DB_BLOCK_SIZE parameter, 466  
detail tables, 314  
external tables, 280, 646  
indexing strategy, 1071  
loading, Oracle Streams, 670  
program global area (PGA), 187  
table compression, 1076  
transportable tablespaces, 716  
using indexes, 296  
database access, 567–586  
see also authorization  
application contexts, 580, 581  
auditing database usage, 586–596  
authenticating users, 596–602  
DBA views managing users/roles/privileges, 577  
definer’s rights, 573  
denying users access to database, 548  
fine-grained data access, 578–586  
application contexts, 579–581  
column-level VPD, 585–586  
fine-grained access control, 582–585  
invoker’s rights, 573  
privileges, 567–574  
object privileges, 570–573  
system privileges, 567–570  
restricting database access, 501–502  
roles, 574–577  
SQL Access Advisor, 212  
views and stored procedures, 577  
database administration commands, SQL*Plus, 134–135  
database administrator see DBA  
database alerts see alerts  
database architecture, 165–178  
database auditing, 612  
database authentication see authentication  
database availability, 171  
database backups, 201  
database buffers  
see also buffer cache  
committing transactions, 198  
least recently used (LRU) algorithm, 181  
modifying data via Oracle memory, 181  
database concurrency see data concurrency  
Database Configuration Assistant see DBCA  
database connections  
OID making, 535–536  
setting upper limit for OS processes, 455  
database connectivity see connectivity  
database consistency see data consistency  
Database Content page, DBCA, 487  
Database Control, 137, 139, 140–153  
accessing, 143–144  
accessing MTTR Advisor, 981  
Alerts table, 953  
automatic optimizer statistics collection, 899  
cloning databases, 838–839  
cloning Oracle home, 148  
Compare Periods Report, 1206–1208  
configuring and using, 140–143  
configuring automatic SQL tuning parameters, 1119  
configuring automatically, 140  
configuring Flashback Database, 855, 856  
configuring manually, 141–143  
creating database links, 987  
creating roles, 150  
Data Grid, OEM alternative, 206
Data Pump Export and Import operations, 688
database management, 146
database usage metrics, 151–153
DBMS_ADVISOR package and, 320
default port, 140
default URL, 140
default URL, 141
default URL, 141
default port, 140
emca utility, 141
end-to-end tracing, 1107
estimating table size before creating, 266
examining database feature-usage statistics, 152
examining database performance, 1195–1201, 1206–1208
examining SQL response time with, 1183
emca utility, 141
examinations database feature-usage statistics, 152
examinations database performance, 1195–1201, 1206–1208
examining SQL response time with, 1183
GATHER_STATS_JOB, 899
invoking SQL Access Advisor, 320–323
linking to MetaLink, 150
logging in, 144
login screen, 142
managing advisory framework, 979
managing alerts, 954–955
managing ASM instances, 907
managing ASM operations, 900
managing AWR snapshots, 961–963
managing database, 213
managing session locks, 354–355
managing users, 567
migrating databases to ASM, 920–921
online database reorganization with, 933
policy-based configuration framework, 151
running ADDM using, 893
setting alert thresholds, 954
setting notification rules, 955
starting, 490
SYSSMAN (super administrator account), 148
tracing individual user sessions, 1105
upgrading with DBUA, 432
versions, 137
viewing ADDM reports, 890–891
Database Control Memory Advisor see Memory Advisor
Database Control pages
Administrators page, 148
Advisor Central page, 150, 979
All Metrics page, 950
AWR page, 962
Availability page, 146
Blackouts page, 148
Data Movement page, 148
Data Reorganization page, 934
Database Performance page, 1197–1200
Dictionary Comparisons page, 147
Edit Thresholds page, 954, 955
Hang Analysis page, 1192–1194
home page, 145, 1195–1197
Manage Policy Library page, 151
Management Pack Access page, 149
Notification Methods page, 148
Patching Setup page, 148
Performance Data Report page, 1201–1202
Performance page, 145–146
performing RMAN backup and recovery tasks, 813
physical database backups, 725, 726
Policy Violations page, 151
Related Links section, 150
Schema page, 147
Segment Advisor page, 931
Segment Advisor Recommendations page, 980
Server page, 146–147
Setup page, 148–149
Software and Support page, 148
database corruption detection, 795–798
database creation, 474–493
authorizations, 446
creating file system, 444–445
creating parameter files, 446–474
Database Configuration Assistant (DBCA), 486–493
default memory management option, 196
ensuring sufficient memory allocation, 445
installing Oracle software, 444
introduction, 443
locating files, 445
manual database creation, 474–486
Oracle Managed Files (OMF), 250–253, 924–927
server parameter file (SPFILE), 493–497
setting OS environment variables, 446
sizing file system, 444–445
Database Credentials window, 432, 487
database design
attribute dependence on primary key, 31, 33
business rules and data integrity, 36
DBA role, 3, 7–8
designing different types of tables, 35
determining the need for a new ER model, 23–24
designing tables, 28–29
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
determining the need for a new ER model, 23–24
database directories, creating, 410
database failures, 801
  data center disasters, 802
  data repair, 803–804
  hardware-related recovery, 802
  human error, 802, 803
  media failures, 802, 803
  Oracle recovery process, 804–809
  recovery with RMAN, 809–814
  system failures, 802
  Transparent Application Failover feature, 802
Database File Locations window, 155, 487
database files, 398–400
  making transactions permanent, 181
database hangs, 1186–1194
  abnormal increase in process size, 1190–1191
  archiver process stuck, 1187–1188
  bad statistics, 1191
  collecting information during database hang, 1191
  gathering error messages, 1193
  getting systemstate dump, 1193
  locking issues, 1189
  severe contention for resources, 1188
  shared pool problems, 1191
  system usage problems, 1188
  using Database Control's Hang Analysis page, 1192–1194
  using hanganalyze utility, 1193
database hit ratios, 1161–1162
Database Identification window, 155, 487
database incarnations, recovery through, 822
database installation see installing Oracle Database 11g
database instance names, 514
database instances see instances
database integrity see integrity
database jobs, Oracle Scheduler, 995
database links, 985–987
  DBA_DB_LINKS view, 329
  Related Links, Database Control, 150
database load affecting performance, 1205
database maintenance, quiescing databases, 505
database management
  see also OEM (Oracle Enterprise Manager)
  automatic database management, 208–209
database management tools, 137
database metrics see metrics
database mode, ADDM, 882
database models, 38–41
  nonrelational database models, 20
database mounted statement, 484
database names, global, 514
database objects, 987–991
  comparing data, 987–989
  configuring, Oracle Secure Backup, 789
  converging data, 990–991
  initial extent, 169
  segments, 169
  storage allocation to, 222
database operations see also Oracle processes
Database page, Grid Control, 153
database parameter files see parameter files
database passwords
  DBCA changing passwords for default users, 490–491
Database Performance page, Database Control, 1197–1200
database quiescing, 505, 944–945
database recovery see recovery
Database Replay, 213, 1209–1216
  change management, 7
  database management, 148
database resident connection pooling (DRCP), 180, 531–533
Database Resource Manager, 208, 554–567, 941–943
  activating, 565
  allocating scarce resources with Scheduler, 996
  data dictionary views managing, 566
  deactivating, 566
  limiting transactions with operation queueing, 942
  limiting execution times for transactions, 942–943
  managing, 555
  managing resources, 554
  managing undo space information, 365
  OEM administering, 566–567
  pending area, 556, 562
  plan directives, 942
  privileges for, 555
  resource allocation methods, 555
  resource consumer groups, 554, 555, 556, 557–559
  assigning users to, 562–565
  automatic assignment to session, 564
  enforcing per-session CPU and I/O limits, 564–565
  managing job classes, 1011
  resource plan directives, 555, 556, 560–562
  resource plans, 554, 555, 556, 559–560
  steps when starting to use, 556
  switching long-running transactions, 942
database schemas, 20
database security see security
database server
  copying files with, 991–992
database service names
  connect descriptors, 515
  Oracle networking, 514
database sessions, terminating, 75
database shutdown command, 492
database sizing, 37
database startup triggers, 591
database storage
  creating database directories, preinstallation, 410
  implementing physical database design, 37
Database Storage window, 488
database structures see Oracle database structures
Database Templates window, 487
database transaction management, 337
database transactions see transactions
database triggers see triggers
database types, 9–10
Database Upgrade Assistant see DBUA
database usage metrics, Database Control, 151–153
Database Usage Statistics property sheet, 152, 153
Database Vault, 401, 430
database wait statistics see wait statistics
database writer (DBWn), 180, 181–182, 183
  how Oracle processes transactions, 197
  starting Oracle instance, 479
  write ahead protocol and, 199
DATABASE_PROPERTIES view, 544
bigfile tablespace information, 238
monitoring current status of instance, 507
databases
  see also Oracle database; relational databases
  audit parameters, 450
  auditing database usage, 586–596
  backing up Oracle databases see backups
  bouncing, 448
  changing into read-only mode, 502
  changing to restricted mode, 501
  cloning, 833–840
  communicating with database, 205–207
  copying files between, 253–255
  creating stored outlines for, 1079
  dropping, 506
  migrating to ASM, 919–921
  monitoring database status, 507
  mounting database, 482
  Oracle XML DB, 1248–1252
  preserving database performance, 1080
  quiescing, 505
  recovering Oracle databases see recovery
  restoring pre-upgrade database, 433
  reverse engineering, 38
  shutting down database from SQL*Plus, 502–505
  starting database from SQL*Plus, 497–499
  suspending databases, 505, 945
  variable naming database connected to, 127
  database-sizing spreadsheets, 392
  DATA_CACHE parameter, SQL*Loader, 641
datafile backups, 728
  DATAFILE clause
  creating Sysaux tablespace, 239
  creating tablespaces, 219
  Oracle Managed Files (OMF), 247, 253
datafiles, 173–174, 394
  accidentally dropping datafiles, 248
  adding within OMF file system, 253
  allocating data blocks to objects, 172
  backing up with RMAN, 777
  converting to match endian format, 721
  database files, 398
  DBA_DATA_FILES view, 245
  dropping, 806
  dumping data block, 167
  expanding tablespaces, 223
  extent allocation/deallocation, 220
  flash recovery area, 735
  Flashback Database restoring, 853
  free and used space in, 172
  increasing size of, 174
  making plain copy of, 757
  making whole closed backups, 790
  mount points, 394, 404
  names and locations of, 174
  naming conventions, 398
  OFA guidelines, 396
  Oracle Managed Files (OMF), 247, 249, 250, 923, 924
  recovering datafiles, 818–820
  recovering datafiles without backup, 828–829
  repairing corrupt datafiles, 742
  RESIZE clause, 219
  restoring vs. recovering, 806
  REUSE_DATAFILES parameter, 707
  separation from logical objects, 174
  setting location for, 239
  sizing for database creation, 444
  specifying default location for, 455
  SQL*Loader utility, 628
  tablespaces, 166, 219–220
  tablespaces and, 170, 173
  V$DATAFILE view, 246
data-flow diagrams (DFDs), 23
  DATA_OPTIONS parameter, Data Pump, 694, 710
DATA_PUMP_DIR objects, 682, 684, 685
datasets, Oracle Secure Backup, 789
date and time data types, 1223
date command, UNIX/Linux, 48
DATE data type, 1223
date functions, SQL, 1229
DATE variable, SQL*Plus, 119, 127
dates, 449, 453
DB block gets, 1144
db file scattered read wait event, 1166, 1167, 1168, 1177, 1204
db file sequential read wait event, 1165, 1166, 1167, 1168, 1178, 1204
db file single write wait event, 1204
DB time metric, 878, 1206
DBA (database administrator)
  asking experts (non-DBA), 16
  avoid making problems worse, 17
  certification, 11–13
  changing user’s password, 547
  connecting to Oracle database, 99
  daily routine, 15–16
  database design role, 7–8
  database security with multiple DBAs, 613
  development DBA, 8
  different organizational roles, 8
  general advice, 16–17
  hands-on learning, 10
  help, knowing when to ask for, 16
  improving SQL processing, 1075–1080
  job classifications, 8
  managing users, 544
  performance tuning tasks, 1086–1087
  production database problems, 554
  production DBA, 8
  resources, 13–14
  responsibility of DBA, 3–8
  security role, 4
  system administration and Oracle DBA, 12–13
  system management role, 5–7
  terminating database sessions, 75
  thinking outside the box, 17
  training, 10, 11, 12, 13, 14, 15
  UNIX/Linux for DBA, 43–95
dba (default name) see OSDBA group
DBA role, 574
DBA views, 204
  DBA_ADVISOR_ACTIONS, 894, 980
  DBA_ADVISOR_DEF_PARAMETERS, 884
  DBA_ADVISOR_EXECUTIONS, 1120, 1220
  DBA_ADVISOR_FINDINGS, 893, 980, 1220
  DBA_ADVISOR_PARAMETERS, 980
  DBA_ADVISOR_RATIONALE, 894, 980
  DBA_ADVISOR_RECOMMENDATIONS, 893, 978, 980
  DBA_ADVISOR_SQLXYZ views, 1120, 1220
  DBA_ADVISOR_TASKS, 980, 1220
  DBA_ALL_TABLES, 330
  DBA_AUDIT_TRAIL, 588, 596
  DBA_AUTO_SEGADV_CTL, 980
  DBA_AUTOTASK_XYZ views, 1020, 1022
  DBA_BLOCKERS, 351, 354
  DBA_COL_PRIVS, 577
  DBA_COMMON_AUDIT_TRAIL, 594, 596
  DBA_COMPARISION_XYZ views, 989
  DBA_CONS_COLUMNS, 311
  DBA_CONSTRAINTS, 310
  DBA_DATA_FILES, 245
  DBA_DATAPUMP_XYZ views, 713, 714
  DBA_DB_LINKS, 329
  DBA_ENABLED_TRACES, 1107
  DBA_EXTENTS, 255
  DBA_EXTERNAL_TABLES, 330
  DBA_FGA_AUDIT_TRAIL, 594, 596
  DBA_FLASHBACK_ARCHIVE, 872
  DBA_FLASHBACK_TRANSACTION_XYZ views, 870
  DBA_FREE_SPACE, 243, 850
  DBA_HIST_ACTIVE_SESS_HISTORY, 971, 972, 1169, 1186
  DBA_HIST_BASELINE, 971
  DBA_HIST_SNAPSHOT, 963, 971
  DBA_HIST_SYMMETRIC_HISTORY, 958
  DBA_HIST_WR_CONTROL, 971
  DBA_HIST_XYZ views, 952
  DBA_IND_COLUMNS, 333
  DBA_INDEXES, 333
  DBA_MVIEWS, 333
  DBA_OBJECTS, 329, 1146, 1147
  DBA_OUTSTANDING_ALERTS, 740, 957, 958
  DBA_PART_TABLES, 331
  DBA_PROFILES, 258
  DBA_RECYCLEBIN, 850
  DBA_REDEFINITION_ERRORS, 940
  DBA_REGISTRY, 427
  DBA_RESUMABLE, 386
  DBA_ROLE_PRIVS, 577
  DBA_ROLES, 577
  DBA_RSRC_CONSUMER_GROUP_PRIVS, 566
  DBA_RSRC_CONSUMER_GROUPS, 557, 559
  DBA_RSRC_PLANS, 566
  DBA_SCHEDULER_JOB_XYZ views, 1018
  DBA_SCHEDULER_JOBS, 931, 997, 1001, 1018, 1047
  DBA_SCHEDULER_XYZ views, 1018
  DBA_SEGMENTS, 244, 255
  DBA_SEQUENCES, 329
  DBA_SERVER_REGISTRY, 427, 429, 440
  DBA_SQL_MANAGEMENT_CONFIG, 1085
  DBA_SQL_PATCHES, 1038
  DBA_SQL_PLAN_BASELINES, 1083
  DBA_SQL_PROFILES, 1117
  DBA_SYS_PRIVS, 577
  DBA_TAB_COL_STATISTICS, 1049, 1087
  DBA_TAB_COLUMNS, 292, 330
  DBA_TABSPACE_GROUPS, 235, 246
  DBA_TABSPACES, 238, 243, 365, 610
  DBA_TAB_MODIFICATIONS, 331, 1047
  DBA_TAB_PARTITIONS, 292, 330
  DBA_TAB_PRIVS, 577
  DBA_TAB_STATISTICS table, 1049
  DBA_TEMP_FILES, 230, 246
  DBA_THRESHOLDS, 956, 958, 959
SET_CONSUMER_GROUP_MAPPING
procedure, 564
SET_CONSUMER_MAPPING_PRI
procedure, 564
SET_INITIAL_CONSUMER_GROUP
procedure, 563
SUBMIT_PENDING_AREA procedure, 562
VALIDATE_PENDING_AREA procedure, 562
DBMS_RESULT_CACHE package, 1123, 1124, 1125
DBMS_RESUMABLE package, 385, 386
DBMS_RLS package, 584
DBMS_RULE_ADM package, 1009
DBMS_SCHEDULER package, 994
administering Scheduler jobs, 1000
ALTER_ATTRIBUTES procedure, 1013
CLOSE_WINDOW procedure, 1015
CREATE_CHAIN procedure, 1009
CREATE_CREDENTIAL procedure, 1005
CREATE_JOB procedure, 998, 999, 1001, 1002, 1005, 1007, 1011
CREATE_JOB_CLASS procedure, 1012
CREATE_PROGRAM procedure, 1006
CREATE_SCHEDULE procedure, 1007, 1008
CREATE_WINDOW procedure, 1014
DEFINE_CHAIN_RULE procedure, 1009
DEFINE_CHAIN_STEP procedure, 1009
DISABLE procedure, 1016, 1007
DROP_JOB procedure, 1000
DROP_JOB_CLASS procedure, 1012
DROP_PROGRAM procedure, 1007
DROP_SCHEDULE procedure, 1008
DROP_WINDOW procedure, 1016
ENABLE procedure, 1006, 1007, 1009
GET_SCHEDULER_ATTRIBUTE
procedure, 1017
LOGGING_XYZ values, 1012, 1019
OPEN_WINDOW procedure, 1015
PURGE_LOG procedure, 1012
RUN_CHAIN procedure, 1010
RUN_JOB procedure, 1000
SET_ATTRIBUTE procedure, 1005, 1008, 1018
SET_ATTRIBUTE_NULL procedure, 1017
SET_ATTRIBUTES procedure, 1016
SET_SCHEDULER_ATTRIBUTE
procedure, 1017
STOP_JOB procedure, 1000
DBMS_SERVER_ALERT package, 227, 955–956, 957
DBMS_SERVER_REGISTRY view, 439
DBMS_SESSION package, 1101, 1106, 1107
DBMS_SHARED_POOL package, 1138, 1142
DBMS_SPACE package, 255–256
ASA_RECOMMENDATIONS function, 980
CREATE_INDEX_COST procedure, 298, 299
estimating size of index, 298–299
estimating space requirements, 268
finding unused space in segments, 928
FREE_BLOCKS procedure, 255
SPACE_USAGE procedure, 255, 268
UNUSED_SPACE procedure, 255
DBMS_SPACE_ADMIN package, 218
DBMS_SPM package, 1081, 1082, 1083, 1085
DBMS_SQLDIAG package, 1035–1038
DBMS_SQLPA package, 1217
DBMS_SQLTUNE package
ACCEPT_SQL_PROFILE procedure, 1115
ALTER_SQL_PROFILE procedure, 1115
configuring automatic SQL tuning, 1117–1118
CREATE_ANALYSIS_TASK procedure, 1218
CREATE_SQLSET procedure, 1217
CREATE_STGTAB_SQLSET procedure, 1218
CREATE_TUNING_TASK procedure, 1113
EXECUTE_ANALYSIS_TASK procedure,
1218, 1219
EXECUTE_TUNING_TASK procedure,
1114, 1117
EXPORT_TUNING_TASK procedure, 1117
interpreting automatic SQL tuning
reports, 1119
managing SQL profiles, 1115
PACK_STGTAB_SQLSET procedure, 1218
performing automatic SQL tuning,
1113–1114
REPORT_ANALYSIS_TASK function, 1219
REPORT_AUTO_TUNING_TASK
function, 1119
REPORT_TUNING_TASK procedure, 1114
running SQL Tuning Advisor, 1113–1115
SELECT_CURSOR_CACHE procedure, 1217
SET_TUNING_TASK_PARAMETERS
procedure, 1117
SYS_AUTO_SQL_TUNING_TASK
procedure, 1118
UNPACK_STGTAB_SQLSET procedure, 1218
DBMS_STATS package
automatic optimizer statistics collection, 898
AUTO_SAMPLE_SIZE procedure, 1055
collecting statistics, 1053–1056
CREATE_EXTENDED_STATS function,
1059, 1060
DELETE_PENDING_STATS procedure, 1057
DROP_EXTENDED_STATS function, 1059
EXPORT_PENDING_STATS procedure, 1057
frequency for refreshing statistics, 1086
GATHER_DATABASE_STATS procedure,
1055–1056, 1062, 1063
GATHER_DATABASE_STATS_JOB_PROC
procedure, 1217
GATHER_DICTIONARY_STATS
procedure, 1063
GATHER_FIXED_OBJECTS_STATS
procedure, 1062
gathering statistics, 1048
GATHER_TABLE_STATS procedure,
1059, 1060
GATHER_XYZ_STATISTICS procedures, 1086
GATHER_XYZ_STATS procedures, 1054
GET_PREFS procedure, 1056
managing/monitoring database, 213
manually collecting optimizer statistics, 900
METHOD_OPT attribute, 1086
PUBLISH_PENDING_STATS procedure, 1057
SET_TABLE_PREFS function, 1057
DBMS_STORAGE_MAP package, 994
DBMS_STREAMS_ADM package, 674
DBMS_STREAMS_AUTH package, 674
DBMS_SYSTEM package, 1102, 1175
DBMS_TRANSACTION package, 380
DBMS_TTS package, 717, 721
DBMS_WORKLOAD_CAPTURE package, 1210
ADD_FILTER procedure, 1210
CANCEL_REPLAY procedure, 1215
FINISH_CAPTURE procedure, 1211
GET_REPLAY_INFO function, 1215
INITIALIZE_REPLAY procedure, 1213
PREPARE_REPLAY procedure, 1214
PROCESS_CAPTURE procedure, 1211
REMAP_CONNECTION procedure, 1214
REPLAY_REPORT function, 1215
START_CAPTURE procedure, 1211
START_REPLAY procedure, 1214
DBMS_WORKLOAD_REPOSITORY package
AWR_REPORT_XYZ functions, 967
configuring ADDM, 881
CREATE_BASELINE procedure, 963
CREATE_BASELINE_TEMPLATE procedure, 965
CREATE_SNAPSHOT procedure, 961
DROP_BASELINE procedure, 964
DROP_SNAPSHOT procedure, 961
managing AWR snapshots, 961
modifying snapshot settings, 964
MODIFY_SNAPSHOT_SETTINGS procedure, 882
DBMS_XPLAN package, 1083, 1084, 1092
db_name attribute, USERENV namespace, 580
DB_NAME parameter, 398, 451, 452, 446
db_name_restore.sh script, 432
DB_nK_CACHE_SIZE parameter, 190, 196, 458, 943
DB_RECOVERY_FILE_DEST parameter, 469, 738
creating ASM databases, 919
flash recovery, 400, 410, 737, 738, 739
OMF, 247, 249, 250, 252, 923, 926
DB_RECOVERY_FILE_DEST_SIZE parameter, 469, 741, 919
flash recovery area, 249, 250, 400, 737, 923
DB_RECYCLE_CACHE_SIZE parameter, 189, 195, 458
DB_SECUREFILE parameter, 472
dbshut.sh script, 423, 499, 500
DBSNMP account, 490, 596
dbstart.sh script, 423, 499, 500
DBUA (Database Upgrade Assistant), 428, 429, 430–433
creating Sysaux tablespace, 239
managing/monitoring database, 213
DB_ULTRA_SAFE parameter, 470, 796
DB_UNIQUE_NAME parameter, 451
DBVERIFY utility, 797
DBWn see database writer
DB_WRITER_PROCESSES parameter, 455, 1179
SPFILE, 182
dd command, UNIX/Linux, 76, 725, 752, 790
DDL (data definition language)
DBMS_METADATA package extracting, 294
DDL statements, 22
executing SQL statements, JDBC, 539–540
locks, 350
pending transactions, 134
resumable database operations, 383
SQL statements, 264
transaction processing of, 337, 338
triggers, 591, 592
using ROLLBACK command with, 272
ddl_lock_timeout parameter, 350
DDL_LOG table, 592
DDL_LOG_TRIG trigger, 592
deadlocks, 340, 346, 352
DEALLOCATE UNUSED option, 220, 928
decision-support system (DSS), 9
declarative referential integrity, 36
DECLARE statement, PL/SQL, 1241, 1245
DECODER function, SQL, 1230
DECRIPT keyword, 608
decryption, data, 604
dedicated server architecture, 512
dedicated server configuration, 180
default accounts, 491
default auditing, 588
default buffer pool, 189, 1146
DEFAULT DEVICE TYPE parameter, RMAN, 763
DEFAULT DIRECTORY clause, 648
default file location, 738
default permanent tablespaces, 235–236
creating/managing users, 544
database creation, 444, 482
default profile, users, 550–551
default role, 574
DEFAULT TABLESPACE clause, 236, 547
default tablespaces, 237, 544
default temporary tablespaces, 232, 444, 482
creating/managing users, 544
default trace directory (UDUMP), 168
DEFAULT_CONSUMER_GROUP, 558, 563, 1011
DEFAULTIF parameter, SQL*Loader, 642
DEFAULT_JOB_CLASS, Scheduler, 1011
DEFAULT_MAINTENANCE_PLAN, 559, 1022
DEFAULT_PLAN resource plan, 559
deferrable constraints, 310
deferred statistics publishing, 1056–1057
DEFINE command, SQL*Plus, 107, 126, 127
DEFINE_CHAIN procedure, 1009
DEFINE_CHAIN_RULE procedure, 1009
definer’s rights, 573
degree of parallelism see parallelism
DEL command, SQL*Plus, 131
delete anomaly, 29, 32, 33
DELETE clause, MERGE statement, 660
DELETE command, RMAN, 758
  EXPIRED option, 758
  OBSOLETE option, 741, 758, 759
  SCRIPT option, 750
DELETE statement, 263, 272, 287
  PL/SQL, 1242
  SQL, 1225–1226
DELETE_CATALOG_ROLE, 569
DELETE_PENDING_STATS procedure, 1057
DELETE_TASK procedure, 890
deleting files, UNIX, 63
DELETION POLICY option, RMAN, 766
DELIMITED BY clause, 648
delimiters, SQL*Loader, 632
delta values, 948
denormalization, 29, 34
dependencies, transportable tablespaces, 716, 717
Deployment Procedure Manager, 148
Deployments page, Grid Control, 159
DESC keyword, ORDER BY clause, SQL, 1226
DESCRIBE command, 293
  Scheduler managing external jobs, 1003
  SQL*Plus, 119
  viewing object information, SQL, 329
detached jobs, Oracle Scheduler, 996
detail tables, materialized views, 314
/dev directory, UNIX, 63
/dev/null, UNIX, 56
development databases, 9
development DBA, 8
device column, iostat command, 82
device files, UNIX, 63
DEVICE TYPE parameter, RMAN, 763, 764, 765
devices, Oracle Secure Backup, 789
df command, UNIX/Linux, 81, 86, 219
DFDs (data-flow diagrams), 23
Diag Alert/Incident/Trace directories, ADR, 1024, 1026
diagnostic framework, 947
diagnostic pack, 459
Diagnostic Summary, Database Control, 145
DIAGNOSTIC_DEST parameter, 178, 449
  ADR, 396, 1023
  OMF, 926
diagnostics
  see also ADDM (Automatic Database Diagnostic Monitor)
  ADR, 178, 211, 396, 1022, 1023–1024
  ADRCI, 1022, 1024–1026
application knowledge for diagnosis, 1182
Data Recovery Advisor, 803, 829, 1023
fault diagnosability infrastructure, 210–211, 1022–1038
Health Monitor, 1022, 1032–1035
incident packaging service (IPS), 1022, 1026–1028
Oracle Diagnostics Pack, 149, 949
performance- and diagnostics-related
  parameters, 461–468
SQL Repair Advisor, 1023, 1035–1038
SQL Test Case Builder, 1023, 1038
Support Workbench, 1022, 1028–1032
dictionary cache, 191, 1134–1135
Dictionary Comparisons page, Database Control, 147
dictionary-managed tablespaces, 217
dictionary tables, 1062–1063
DICTIONARY value, extent management, 221
DICTIONARY_ACCESSIBILITY parameter, 613
diff command, UNIX/Linux, 53
difference operation, 21
differential backup, RMAN, 756, 757
dimensions
  MODEL clause creating multidimensional arrays, 668, 670
DIRECT clause, SQL*Loader, 641
direct hand-off to dispatcher, 512
DIRECT parameter, SQL*Loader, 634
Direct Path API, 680
direct path read/write wait events, 1178
directories, Oracle
  administrative directories, 397
  creating directories for database files, 399
  creating directories, preinstallation, 409–410
directory structure, OFA guidelines, 395
  mount points, 394
  naming conventions, OFA guidelines, 394
  Oracle base, 395
  Oracle home, 395, 396
  Oracle Inventory directory, 396
directories, UNIX
  bin directory, 48
  changing directories, 48
  creating, 62
device files, 63
directory management, 62
file types, 57
home directory, 46, 63
indicating file is directory, 60
individual directories described, 63
listing files in directory, 58
locating directories, 52
mount points, 86
navigating directory structure, 62
present working directory, 49
removing directories, 62
root directory, 63
system configuration files, 63
temporary files, 63
directory administration tools, OID, 535
Directory Information Tree (DIT), 536
directory management, UNIX, 62
directory naming context, 537
directory naming method, 525, 534–537
directory objects, 178, 682
creating external table layer, 648–649
Data Pump utilities using, 681–685
DIRECTORY parameter, Data Pump, 683, 684, 690, 706
directory privileges, 571
directory services, 534
directory structure, OFA-compliant, 399
directory structure, UNIX, 47
DIRECTORY_PATH parameter, NAMES, 530
direct-path loading, SQL*Loader, 627, 628, 639–642
dirty buffers, 188
dirty reads problem, 341, 344, 345
disable no validate state, 309
DISABLE procedure
   DBMS_AUTO_TASK_ADMIN, 1021, 1118
   DBMS_SCHEDULER, 1007, 1016
disable validate state, 309
disaster recovery, 798–800
   see also recovery
DISCARD FILE parameter, 648
DISCARD parameter, SQL*Loader, 635
DISCARDMAX parameter, SQL*Loader, 635
discrete transactions, Oracle, 380
disk allocation/layout, OFA, 393
disk blocks, 166
disk cache for tape, flash recovery area, 735
disk configuration strategies, 85–88, 93–95
   RAID systems, 88–93
disk fragmentation, ASM, 901
disk groups, ASM see ASM disk groups
disk I/O see I/O
disk mirroring see mirroring
DISK PARALLELISM parameter, RMAN, 764, 765
disk partitioning, 87
disk reads, 1109
disk space, 403
disk storage requirements, Oracle Database 11g, 392
disk storage, UNIX, 85–88
   performance monitoring, 81
   RAID systems, 88–93
disk striping, 87, 88
Disk-Based Backup and Recovery, 734
disk_reads column, V$SQL view, 1108
DISK_REPAIR_TIME attribute, 908, 910
disks
   configuring physical disks, 88
cost of disk I/O, 186
disk I/O, 1159
   modifying data on, 181
   monitoring disk usage, UNIX, 86
   RAID systems, 88–93
   recovering from damaged disk drive, 849
DISMOUNT FORCE clause, ALTER DISKGROUP, 913
dispatchers, 512
   shared server processes, 180
DISPLAY environment variable, DBCA, 486
DISPLAY function, DBMS_XPLANNER package, 1092
DISPLAY variable, 411, 412, 414, 416, 422
DISPLAY_AWR function, 1092
DISPLAY_SQL_PLAN_BASELINE function, 1083, 1084
DISTINCT operation, indexing strategy, 1071
distinguished names (DNs), 536, 537
distributed locks, 351
distribution files, OFA guidelines, 399
distribution functions, SQL, 1232
DIT (Directory Information Tree), OID, 536
DML (data manipulation language)
   DML statements, 22, 263
   executing SQL statements, JDBC, 539–540
   indexing strategy, 1071
   locks, 349–350
   making DML changes permanent, 133
   PL/SQL, 1242
   resumable database operations, 383
   row- and table-level locks, 349
   transaction processing of, 337, 338
   triggers, 590–591
   VERSIONS_OPERATION pseudo-column, 371
   view showing DML changes, 331
DMn processes, Data Pump, 685
DNs (distinguished names), 536, 537
documentation review, Oracle Database 11g, 392
dollar sign ($) character, SQL, 1238
domain component, DNs, 537
domain constraints, 37
domains, 20, 514
DOUBLE data type, 1222
double failure protection mode, Oracle Data Guard, 800
downgrading from Oracle Database 11g, 441
downloading Oracle software, 415–416
   Oracle Client software, 517
dpdump directory, 397
d Predicate predicate, 583
DRCP (database resident connection pooling), 180, 531–533
DriverManager class, JDBC, 538
drivers see JDBC drivers
DROP CATALOG command, RMAN, 768, 772, 774
DROP command, ALTER TABLE statement, 271
DROP DATABASE command, 506
DROP DATAFILE command, 806
DROP FLASHBACK ARCHIVE command, 872
DROP MATERIALIZED VIEW command, 320
DROP OUTLINE command, 1079
DROP PARTITION command, 291, 302
DROP PROFILE command, 553
DROP ROLE command, 577
DROP STORAGE command, 220
DROP SYNONYM command, 326
DROP TABLE command, 116, 224, 276, 548, 849
DROP TABLE PURGE command, 116
DROP TABLESPACE command, 224, 225, 248, 364
DROP UNUSED COLUMNS command, 271
DROP USER command, 547–548
  CASCADE option, 548, 853
DROP VIEW command, 313
DROP_ACL procedure, 617
DROP_BASELINE procedure, 964
DROP_EXTENDED_STATS function, 1059
DROP_JOB procedure, 1000
DROP_JOB_CLASS procedure, 1012
dropped tables, restoring, 851–852
dropping databases, 506
dumping data block contents, 167, 168
dumping data files, 691
  COMPRESSION parameter, Data Pump, 691
  creating export dump file, 704
  Data Pump utilities, 678, 680, 681
  importing metadata from, 719
  matching degree of parallelism, 700
  REUSE_DUMPFILE parameter, Data Pump, 691
DUMPFILE parameter, Data Pump, 690, 700, 705, 706
dumping data block contents, 167, 168
duplexing, redo log groups, 983
DUPLICATE command, RMAN, 810, 834–837
durability property, transactions, 340
Dw nn processes, Data Pump, 686
dynamic data sampling, ATO, 1112
dynamic initialization parameters, 177
dynamic parameters, 448, 493, 496–497
dynamic performance tables, 203, 204
collecting fixed object statistics, 1062
dynamic performance views, 203, 204
  see also $V$ views
  automatic performance tuning compared, 1131–1132
database metrics, 950, 959
temporary statistics, 959
dynamic resource management, 941–943
dynamic sampling, 1063
dynamic security policy, 584
dynamic service registration, 183
  Oracle PMON process, 521
dynamic-access predicates, row-level security, 583
  E
  easy connect method, 98, 100, 206
easy connect naming method, 525, 529–530
echo command, UNIX/Linux, 48, 53, 55
  ECHO variable, SQL*Plus, 107
ed command, SQL*Plus, 106
edit Thresholds page, Database Control, 954, 955
  EDITFILE variable, SQL*Plus, 107
editing files with vi editor, UNIX, 63–65
  editing within SQL*Plus, 129–134
  EDITOR variable, SQL*Plus, 128
editors, SQL*Plus default, 124
  egrep command, UNIX/Linux, 66
  elapsed_time column, V$SQL view, 1108
  elementstape libraries, Oracle Secure Backup, 789
  else keyword, UNIX, 72
  Emacs text editor, UNIX, 65
e-mail notifications, Database Control, 148
  embedded SQL statements, 262
  emca utility, 141
  emca.rsp response file template, 422
  emctl utility, 143, 156, 157
  EMPHASIS method, resource plans, 560, 561
  emulators, UNIX, 46
  enable no validate state, 309
  enable parameter, ADD_POLICY procedure, 594
  ENABLE procedure
    DBMS_AUTO_TASK_ADMIN, 1021, 1118
    DBMS_SCHEDULER, 1006, 1007, 1009
  ENABLE QUERY REWRITE clause, 319
  ENABLE RESUMABLE clause, 384, 385
  ENABLE ROW MOVEMENT clause, 286, 929
  ENABLE TRIGGERS clause, 377
  enable validate state, 309
  ENABLE_AT_SYSTEM_CHANGE_NUMBER procedure, 369
  ENABLE_AT_TIME procedure, 368
  ENABLED attribute, CREATE_JOB procedure, 999
  ENABLED VALIDATED constraints, 316
  encapsulation, 39
  ENCLOSED BY clause, SQL*Loader, 632
  ENCRYPT clause, creating tables, 269
  ENCRYPT keyword, 604, 607–608
  encrypted passwords, database authentication, 601
  encrypted tablespaces, 240–243
  encryption
    data decryption, 604
data encryption, 603–608
dual-mode encryption, 763

generating master encryption key, 607

password-based encryption, 763

tablespace encryption, 608–610

transparent data encryption, 604–608

transparent encryption, 763

encryption algorithms, 608

ENCRYPTION clause, creating tablespaces, 242, 610

ENCRYPTION parameter

Data Pump Export utility, 695

populating external tables, 652

RMAN, 763, 764

cryptocurrency wallet

creating Oracle Wallet, 241–242

encryption, RMAN, 781, 782

ENCRYPTION_ALGORITHM parameter, Data Pump, 695

ENCRYPTION_MODE parameter, Data Pump, 695

ENCRYPTION_PASSWORD parameter, Data Pump, 695, 696, 698

ENCRYPTION_WALLET_LOCATION parameter, 241, 609

END BACKUP command, 792

End of Installation window, 420

END statement, PL/SQL, 1241

END_DATE attribute

CREATE_JOB procedure, 999

CREATE_WINDOW procedure, 1014

endianness format, 720–723

END_SNAPSHOT parameter, ADDM, 883

to-end tracing, 1105–1107

ENFORCED mode, 316

enforced value, 465

enqueue waits event, 1179

Enterprise Edition, 417, 422

Enterprise Manager, 15

enterprise user security, 603–611

data encryption, 603–608

LDAP, 603

shared schemas, 603

Single Sign-On feature, 603

tablespace encryption, 608–610

transforming ER diagrams into relational tables, 35

entity-relationship (ER) modeling, 24–26

business rules and data integrity, 36

ER modeling tools, 34

entryID attribute, USERENV namespace, 580

env command, UNIX/Linux, 54

evironment variables

see also SQL*Plus environment variables

OEM versions managing, 139

Oracle user’s home directory, 413

setting, post-installation, 424

setting, preinstallation, 410–413

setting for database creation, 446

UNIX, 54, 55

equi joins, 1066, 1233

error handling

Java programs, 540

PL/SQL, 1242

error logging

autonomous transactions, 381

SQL*Plus error logging, 111–112

error messages, database hangs, 1193

ersors

benefits of RMAN, 742

common resumable errors, 383

data block corruption, 167

Flashback error correction using undo data, 366–368

normal program conclusion without, 338

ORA-00257: Archiver error, 741

ORA-00376: file # cannot be read ..., 868

ORA-01031: insufficient privileges error, 938

ORA-01078: failure to process system parameters, 478

ORA-01152: file # was not restored ..., 867

ORA-01194: file # needs more recovery ..., 866

ORA-01536: space quota error, 383

ORA-01555: snapshot-too-old error, 209, 356, 359, 364

ORA-01588: must use RESETLOGS option ..., 866

ORA-01589: must use RESETLOGS or ..., 866

ORA-01628: maximum extents error, 383

ORA-01653: out of space error, 383

ORA-01756: quoted string not properly terminated, 132

ORA-15110: no diskgroups mounted, 906

ORA-19804: cannot reclaim string bytes ..., 741

ORA-19809: limit exceeded for recovery file, 741

ORA-19815: WARNING: db_recovery_file_dest_size, 740

ORA-29701: unable to connect to Cluster Manager, 905

ORA-30032: the statement has timed out, 384

ORA-30036: unable to extend segment, 385

ORA-30393: a query block in the statement did not write, 316

ORA-4031: out of shared pool memory, 894

recovery errors, 866–870

REWRITE_OR_ERROR hint, 315

sec_protocol_error_xyz_action parameters, 615

SHOW ERRORS command, 118

standard error, UNIX, 56

viewing error alerts, Support Workbench, 1029

ERRORS parameter, SQL*Loader, 634
esac keyword, case command, UNIX/Linux, 74
ESTIMATE parameter, Data Pump, 696
Estimate Table Size page, Database Control, 266
ESTIMATE_ONLY parameter, Data Pump, 697
/etc directory, UNIX, 63, 67
ETL (extraction, transformation, loading), 625, 626
see also transforming data
data loading, 627
external tables, loading data using, 645–656
multitable inserts, 660–662
Oracle Streams, 671–675
populating external tables, 650
SQL*Loader utility, 627–645
evaluation mode, SQL Access Advisor, 321
event class metrics, 951
event management and notification, Oracle Streams, 670
event metrics, 951
events
event 10046, tracing SQL code, 1174
Oracle Streams, 670
setting events, caution when, 1193
wait events, 1163
events, Oracle Scheduler, 995, 1010–1011
EVOLVE_SQL_PLAN_BASELINE function, 1083
EXCEPTION statement, PL/SQL, 1241, 1243
exceptions, UTL_FILE package, 258
EXCLUDE parameter
Data Pump Export utility, 692–693, 704
Data Pump Import utility, 708
excluded addresses, securing network, 614
exclusive locks, 199, 349
executable files
components of Oracle process, 1190
Oracle home directory, 177, 395
EXECUTE command, SQL*Plus, 121
execute permission, UNIX files, 59
EXECUTE privileges, Scheduler, 997
EXECUTE SCRIPT command, RMAN, 749
EXECUTE_ANALYSIS_TASK procedure, 1218, 1219
EXECUTE_CATALOG_ROLE, 569, 717
EXECUTE_DIAGNOSTIC_TASK procedure, 1037
eexecuteQuery method, JDBC, 539
EXECUTE_TASK procedure, 324, 890, 978
EXECUTE_TUNING_TASK procedure, 1114, 1117
eexecuteUpdate method, JDBC, 539
execution history, ATO, 1112
execution phase
SQL processing, 343, 1133
TKPROF utility output, 1104
execution plan generation phase, 1044
execution plans, 343
Autotrace utility, SQL, 1097
EXPLAIN PLAN tool, 1090–1095
query processing, 1043
query with index, 1098
after analyzing table, 1099
query without index, 1097
SQL plan baselines, 1080–1085
SQL Plan Management (SPM), 1080–1087
TKPROF utility output, 1105
using hints to influence, 1067–1068
execution stage, query processing, 1046
execution time, query optimization, 1043
evaluation mode, SQL Access Advisor, 321
execution times, limiting, 942
EXECUTION_DAYS_TO_EXPIRE parameter, 1118
EXECUTION_TYPE parameter, 1219
EX_FAIL/EX_FTL, SQL*Loader return codes, 639
EXISTS operator, subqueries, SQL, 1237
existsNode operator, SQL/XML, 1249, 1251
exit code, finding, 69
exit command, RMAN, 745
exit command, UNIX/Linux, 48
EXIT_CLIENT parameter, Data Pump, 703, 704, 713
exp utility, 680
expdp utility, 678, 680, 687
EXP_FULL_DATABASE privilege, 685
EXP_FULL_DATABASE role, Data Pump, 574, 703
EXPIRATION parameter, AWR, 966
EXPIRATION_DATE parameter, Scheduler, 1003
expired account, database authentication, 597
EXPLAIN parameter, TKPROF utility, 1102
EXPLAIN PLAN tool, SQL, 319, 1090–1095
indexing strategy, 1070
monitoring index usage, 304
EXPLAIN PLANs
Autotrace utility, 1095, 1096, 1098
comparing SQL statements, 1127
RESULTCACHE hint, 1121
SQL Trace tool using, 1100
EXPLAIN_MVIEW procedure, 317
EXPLAIN_REWRITE procedure, 316, 317
explicit capture, Oracle Streams, 671
explicit cursors, 1245
explicit locking, 350–352
export command, UNIX/Linux, 51, 53, 54
export modes, Data Pump, 688–689
export parameters, Data Pump, 689–704
export prompt, Data Pump, 688
export utilities
CONSTRAINTS parameter, 692
continued support for, 677
FILE parameter, 690
GRANTS parameter, 692
INDEXES parameter, 692
manual upgrade process, 427
Export utility see under Data Pump utilities
(Export, Import)
EXPORT_PENDING_STATS procedure, 1057
EXPORT_SQL_TESTCASE_DIR BY_XYZ
function, 1038
EXPORT_TUNING_TASK procedure, 1117
EXPRESSION parameter, SQL*Loader, 636
expression statistics, 1059
EX_SUCC, SQL*Loader return codes, 639
extended optimizer statistics, 1058–1060
EXTENT MANAGEMENT clause, 230
EXTENT MANAGEMENT LOCAL clause, 220
extents, 166, 169
allocating data blocks to objects, 172
amalgamating free extents, 184
ASM mirroring, 914
AUTOALLOCATE option, 216, 221, 222
deallocating unused space, 268
default for tablespace extent
management, 216
default number of, 221
extent allocation/deallocation, 220–222
extent management, 219, 221
extent sizing, 216, 219
determining sizing, 220–222
temporary tablespaces, 230
INITIAL_EXTENT parameter, 221
NEXT_EXTENT parameter, 221, 222
operating system files, 220
performance, 220
segments, 166, 169
UNIFORM option, 216, 221
UNIFORM SIZE clause, 221
using bigfile tablespaces, 237
external authentication, 601–602
external authorization, roles, 576
external data loading, SQL*Loader, 627
external jobs
local/remote external jobs, 1002
Oracle Scheduler, 996, 1002–1006
eXternal naming method, 525, 533–534
external redundancy level, ASM, 909, 914
external tables, 280
creating external table layer, 646–649
access drivers, 648
directory objects and locations, 648–649
ETL components, 626
existence of, 645
indexing, 646
loading data using, 645–656
manual collection of statistics required, 1054
populating external tables, 649–652
SQL*Loader, 625, 646
using external tables, 652–653
using SQL*Loader with, 653–656
writing to external tables, 651–652
external tables feature, 207
Data Pump data access, 680
data warehousing, 646
SQL*Loader, 627
external_name attribute, USERENV
namespace, 580
EXTERNAL_TABLE parameter, 653
EXTPROC functionality, PL/SQL, 614
extract operator, SQL/XML, 1249, 1251
extracting data see ETL (extraction,
transformation, loading)
extрактиValue operator, SQL/XML, 1249, 1251
EX.Warn, SQL*Loader return codes, 639
EZ Connect string, DRCP connection, 532
EZCONNECT method, 530
F
FAILED_LOGIN_ATTEMPTS parameter, 550,
599, 612
FAILGROUP keyword, ASM disk groups, 915
failure code, SQL*Loader return codes, 639
failure grouping, Data Recovery Advisor, 830
failure groups, ASM disk groups, 914
failure priority, Data Recovery Advisor, 830
failure status, Data Recovery Advisor, 830
failures, database, 801–804
Transparent Application Failover, 802
FAN (Fast Application Notification) events, 101
fast commit mechanism, 183, 199
fast mirror resync feature, ASM, 908–909
FAST option, materialized views, 317, 318, 319
Fast Start Checkpointing, 805
Fast Start Fault Recovery, 804, 805
FAST_START_MTTR_TARGET parameter, 805
automatic checkpoint tuning, 933
MTTR Advisor, 981
redo log sizing, 1205
fatal error code, SQL*Loader return codes, 639
fault diagnosability infrastructure, 210–211,
1022–1038
Automatic Diagnostic Repository, 211, 1022,
1023–1024
ADRCI, 211, 1022, 1024–1026
Data Recovery Advisor, 211, 1023
Health Monitor, 1022, 1032–1035
health monitor, 211
incident packaging service (IPS), 211, 1022,
1026–1028
SQL Repair Advisor, 1023, 1035–1038
SQL Test Case Builder, 211, 1023, 1038
Support Workbench, 211, 1022, 1028–1032
FCLOSE/FCLOSE_ALL procedures,
UTL_FILE, 258
FEEDBACK variable, SQL*Plus, 107, 108
FETCH command, explicit cursors, PL/SQL, 1245
Fetch operation, TKPROF utility, 1104
fetching, SQL processing steps, 1133
FGAC (fine-grained access control), 578,
582–585
DBA_FGA_AUDIT_TRAIL, 596
index

fgrep command, UNIX/Linux, 66
field list, SQL*Loader, 629
Fifth Normal Form (5NF), 34
FILE ARRIVAL event, 1010, 1011
file deletion policy, flash recovery area, 740
file directory, UTL_FILE package creating, 256
file locations, flash recovery area, 738
file management
  see also ASM (Automatic Storage Management)
  Oracle Managed Files (OMF), 247–253, 922–927
file mapping, 993–994
file metrics, 951
FILE parameter, export utility, 690
file systems
  alias for file system directory, 178
database creation, 444–445
disk configuration strategies, 87
Oracle Managed Files (OMF) managing, 247
filemap.ora file, 993
FILE_MAPPING parameter, 993, 994
FILENAME parameter, TKPROF utility, 1102
filename.ist file, 120
filenames, ASM, 917–918
file-related parameters, 453–454
files
  administrative files, 397
  alert log file, 177
  backup files, 178
c control files, 173, 174–175
copying files between databases, 253–255
database files, 398–400
datafiles, 173–174
FCLOSE procedure, 258
FCLOSE_ALL procedure, 258
FOPEN function, 257
initialization files, 173
locating for database creation, 445
naming conventions, OFA guidelines, 394
network administration files, 173
operating system files, 256–259
password file, 177
product files, 397
recovering, SQL*Plus, 134
redo log files, 173, 175–176
setting file permissions, preinstallation, 409
SPFILE (server parameter file), 177
trace files, 178
UTL_FILE package, 256–259
files, UNIX, 57–62
  changing filename, 59
  changing group, 62
  comparing files, 53
copying, 59
creating file without data, 63
device files, 63
directory management, 62
editing files with vi, 65
editing text with vi, 63
file types, 57
indicating file is directory, 60
joining files, 67
linking files, 57–58
listing files in directory, 58
locating files, 52
location of executable files, 49
locations and paths, 47
managing files, 58–59
moving around files, 65
moving file location, 59
outputting columns, 66
pattern matching, 65
permissions, 59–62
protecting files from overwriting, 57
reading contents of files, 52
removing directories containing files, 62
removing duplicate lines of text, 68
removing files, 59
sending and receiving files using FTP, 79
shell scripts, 68–74
sorting text, 68
special files, 57
system configuration files, 63
temporary files, 63
text extraction utilities, 65
viewing contents of, 58
FILESIZE parameter, Data Pump, 690, 705
FILE_TYPE, UTL_FILE package, 257
Filter Options page, SQL Access Advisor, 322
filtering
  metadata filtering, 692
WHERE clause, SQL 1066, 1226
find command, UNIX/Linux, 52
findings, ADDM, 880
ADDM reports, 888, 891
c fine-grained auditing, 586, 593–596
c fine-grained data access, 578–586
c application contexts, 579–581
c column-level VPD, 585–586
c fine-grained access control (FGAC), 578, 582–585
c fine-grained network access control, 615–618
c fine-grained recovery, 840–847
FINISH_CAPTURE procedure, 1211
FINISH_REDEF_TABLE procedure, 940, 941
Finnegan, Peter, 614
firewalls, 614
First Normal Form (1NF), 30–31
FIRST_ROWS value, OPTIMIZER_MODE, 1050
FIRST_ROWS(n) hint, 1067
FIX_CONTROL parameter, 1036
FIXED clause, external table layer, 647
c fixed dictionary tables, 1062
c fixed record format, SQL*Loader, 631
c fixed SQL plan baselines, 1083
c fixed value thresholds, 954
FIXED_DATE parameter, 449
flash recovery area, 201, 734–741
 Automatic Disk-Based Backup and Recovery, 734
 backing up, 739
 configuring, 737
 contents of, 735
 control files, 738
 creating, 400, 736–739
 creating, preinstallation, 410
 database creation log, 485
 DB_RECOVERY_FILE_DEST parameters, 249
 default file location, 738
 description, 468
 disabling, 737
 dynamically defining, 737
 Flashback Database, 854, 857, 858
 LOG_ARCHIVE_DEST_10 destination, 736
 managing, 740–741
 managing/monitoring database, 213
 OFA guidelines, 396
 Oracle Managed Files (OMF), 734, 738
 out-of-space warning and critical alerts, 741
 parameters, setting up, 739
 redo log files, 738
 sizing, 736
 sizing for database creation, 445
 specifying default location for, 469
 specifying size of, 469
 upgrading with DBUA, 432
 Flashback Data Archive feature, 202, 870–874
 Flashback techniques, 848
 flashback data archiver (FBDA) process, 186
 Flashback Database, 202, 469, 853–861
 block media recovery (BMR), 864
 brief comment, 367
 configuring, 854–856
 data repair, 803
 database-level Flashback techniques, 848
 disabling, 856–857
 enabling, 855
 example using, 859–861
 flash recovery area, 735, 854, 857, 858
 flashback database logs, 854
 limitations, 861
 privileges, 857
 restore points, 863
 RVWR (recovery writer), 857
 storage limits, 857–858
 FLASHBACK DATABASE statement, 854, 857
 Flashback Database logs
 flash recovery area, 857
 Flashback Drop, 202, 276, 848, 849–853
 description, 367, 376, 377
 flashback features, 200
 using flashback features for auditing, 593
 flashback levels, 848
 flashback logs, flash recovery area, 735
 Flashback Query, 202, 366, 367–368
 correcting human error, 802
 RETENTION GUARANTEE clause, 374
 row-level Flashback techniques, 848
 using flashback features for auditing, 593
 flashback recovery techniques, 202
 Flashback Table, 202, 366, 376–378, 848
 FLASHBACK TABLE statement, 377, 378, 852
 Flashback techniques, 808, 848
 FLASHBACK TO BEFORE DROP statement, 116
 Flashback Transaction, 366, 379–380
 Flashback Transaction Backout, 202, 848,
 868–870
 Flashback Transaction Query, 366, 372–375,
 593, 848
 Flashback Versions Query, 366, 369–372,
 375–376, 593, 848
 FLASHBACK_RETENTION_TARGET parameter, 857
 flashbacks
 DBMS_FLASHBACK package, 368–369
 error correction using undo data, 366–368
 Oracle Database 11g features, 366
 specifying flashback time, 469
 undo tablespaces, 367
 UNDO_RETENTION parameter, 359
 FLASHBACK_SCN parameter, Data Pump,
 699, 713
 FLASHBACK_TIME parameter, Data Pump,
 700, 713
 FLASHBACK_TRANSACTION_QUERY view,
 372, 373
 FLOAT data type, 1222
 flow control structures, UNIX, 71–74
 FLUSH clause, ALTER SYSTEM, 1135
 FLUSH procedure, DBMS_RESULT_CACHE, 1123
 FLUSH variable, SQL*Plus, 108
 FMON process, file mapping, 993
 FMPUTL process, file mapping, 993
 footers, SQL*Plus, 123
 FOPEN function, UTL_FILE package, 257
 FOR LOOP statement, PL/SQL, 1244, 1246
 FORCE attribute, Scheduler jobs, 1000
 FORCE option
 refreshing materialized views, 317
 starting ASM instances, 908
 FORCE value
 CURSOR_SHARING parameter, 466, 1139
 QUERY_REWRITE_ENABLED parameter, 315
 for-do-done loop, UNIX, 73
 FOREIGN KEY REFERENCES clause, 285
 foreign keys, 35, 36, 308, 1071
 FORMAT clause
 BACKUP command, RMAN, 754
 SET SERVEROUTPUT command,
 SQL*Plus, 109
 FORMAT parameter
 converting datafiles to match endian
 format, 721
 RMAN backup file locations, 755
 formatting, SQL*Plus, 118, 122–124
 forName method, java.lang, 538
%FOUND attribute, PL/SQL, 1246
Fourth Normal Form (4NF), 34
fractured block problem, whole open backups, 792
fragmentation, segment space management, 928
free buffer waits event, 1178
free buffers, 188
free space, 255–256
alerts, 226
database writer (DBWn) process, 199
datafiles, 219
DBA_FREE_SPACE view, 243
extent allocation/deallocation, 220
preinstallation checks, 403, 404
Recycle Bin, 244
segment space management, 217
free space section, data blocks, 167
free, vmstat utility, 82
FREE_BLOCKS procedure, DBMS_SPACE, 255
freelists, segment space management, 217
FREQ keyword, Oracle Scheduler, 999
FROM clause, SELECT statement
necessity for dual table in Oracle’s SQL, 102
subqueries, 263
FTP (File Transfer Protocol), 79
full export mode, Data Pump, 688
FULL hint, 1067
FULL parameter, Data Pump, 688, 708
full table scans
avoiding unnecessary full table scans, 1075
guidelines for creating indexes, 297
INDEX_FFS hint, 1068
query optimization, 1052
fully qualified name, 452
function-based indexes, 302, 1066, 1072
function privileges, 571
functional dependence, 29, 33
functions, SQL see SQL functions
fuzzy-read problem, data concurrency, 342
G
Gather Statistics Wizard, 1195
GATHER_DATABASE_STATS procedure, 1055–1056, 1062, 1063
GATHER_DATABASE_STATS_JOB_PROC procedure, 899
GATHER_DICTIONARY_STATS procedure, 1063
GATHER_FIXED_OBJECTS_STATS procedure, 1054, 1062
GATHERING_MODE parameter, 1060
GATHER_STATS_JOB, 899–900, 1047–1049, 1065
GATHER_SYSTEM_STATS procedure, 1060–1062
GATHER_TABLE_STATS procedure, 1059, 1060
GATHER_XYZ_STATISTICS procedures, 1086
GATHER_XYZ_STATS procedures, 1054
GENERATED ALWAYS AS clause, 270
GET command, SQL*Plus, 106, 128
get command, UNIX/Linux, 79, 80
getConnection method, JDBC drivers, 538
GET_DDL procedure, 295
GET_FILE procedure, 253, 254, 992
GET_LINE procedure, 257
GET_PREFS procedure, 1056
GET_REPLAY_INFO function, 1215
GET_REPORT function, 884
GET_RUN_REPORT function, 1033
GET_SCHEDULER_ATTRIBUTE procedure, 1017
GET_SYSTEM_CHANGE procedure, 369
GET_TASK_REPORT procedure, 890, 978
GET_THRESHOLD procedure, 956
glance command, UNIX/Linux, 85
GlimpsePlus package, performance monitoring, 84
global authorization, roles, 576
global database names, Oracle networking, 514
global partitioned indexes, 302, 1073
global preferences file, SQL*Plus, 110
GLOBAL QUERY REWRITE privilege, 317
GLOBAL SCRIPT command, RMAN, 750
GLOBALLY clause, CREATE ROLE, 576
GLOBAL_NAMES parameter, Oracle Streams, 673
glogin.sql file, 110, 111
gpm command, UNIX/Linux, 85
GRANT ANY OBJECT privilege, 569, 573
GRANT ANY PRIVILEGE privilege, 569
GRANT command, SQL*Plus, 136
GRANT statement, 567, 568–569, 572
GRANT_ADMIN_PRIVILEGE procedure, 674
granting privileges, 612, 618
to users with UTL_FILE package, 257
granting roles, recovery catalog, 767
GRANTS parameter, export utility, 682
granular recovery techniques, 840–847
LogMiner utility, 841–847
tablespace point-in-time recovery, 840–841
granularity
autonomous transactions, 381
Data Pump technology, 679
locks, 348
GRANULARITY attribute, 1055
grep command, UNIX/Linux, 49, 65, 75
Grid Control, 139, 153–161
alerts, 160
analyzing page performance, 160
components of Grid Control framework, 154
configuring automatic SQL tuning parameters, 1119
connecting to, 157
critical patch advisories, 159
Database Control, OEM alternative, 206
database management, 158, 213
deployments summary, 159
enterprise configuration management, 158
features, 158–159
grouping targets, 159
installing, 154–156
logging into, 157
Management Repository, 154
managing enterprise using, 159
managing groups, 161
monitoring and managing enterprise configuration, 158
monitoring application servers, 160
monitoring host performance, 160
monitoring host/databases/services, 154
monitoring system with, 159–161
monitoring web applications, 160
obtaining host and database configuration information, 158
OEM Management Agent, 154, 156
Oracle Management Service (OMS), 154, 157
Oracle software cloning, 148
resource center, 159
status information, 159
transaction performance, 160
upgrading with DBUA, 432
user interface for, 154
versions, 137
viewing ADDM reports, 890–892
Grid Control pages, 159–160
Database page, 153
GROUP BY clauses
guidelines for creating indexes, 297
indexing strategy, 1071
program global area (PGA), 193
SQL, 1234–1236
group commits, log writer (LGWR) process, 199
grouping operations, SQL, 1234–1236
groups, Grid Control, 159, 161
groups, UNIX, 62, 406–408
guaranteed restore points, 862, 863
guaranteed undo retention, 362–365

H
handler_xyz parameters, ADD_POLICY procedure, 594, 595
Hang Analysis page, Database Control, 355, 1192–1194
hanganalyze utility, database hangs, 1193
hangs see database hangs
HARD (Hardware Assisted Resilient Data), 95, 798
hash clusters, 296
hash joins, 1052, 1068
hash partitioning, 283–284, 288
HASH_AREA_SIZE parameter, 194
HASHKEYS value, CREATE CLUSTER, 296
hash-partitioned global indexes, 303
HAVING clause, 1067
HAVING operator, GROUP BY clause, SQL, 1236
head command, UNIX/Linux, 65
headers, SQL*Plus, 123
HEADING variable, SQL*Plus, 108
Health Monitor, 211, 1022, 1032–1035
heap, 1190
heap indexes, 300
heap-organized tables, 265, 1072
help
man command, UNIX/Linux, 50
showing help topics, SQL*Plus, 106
HELP command, Data Pump, 703, 704, 713
help command, lsnrctl utility, 522
HELP INDEX command, SQL*Plus, 106
Help page, Grid Control, 159
heuristic strategies, query processing, 1046
hidden Oracle parameters, 177
hierarchical queries, SQL, 1232
High Availability, Database Control Performance page, 145
high redundancy level, ASM, 909, 914
high water mark see HWM
High Water Marks page, 153
high-availability systems, 798–799
hints
NO_RESULT_CACHE hint, 1122
optimizer hints, 1051
optimizing queries, 205
RESULT_CACHE hint, 1121, 1122
REWRITE_OR_ERROR hint, 315
using hints to influence execution plans, 1067–1068
histogram functions, SQL, 1232
histograms, 1086–1087
history
Active Session History (ASH), 210, 971–975
history command, UNIX/Linux, 49
hit ratios
buffer cache hit ratio, 190, 1144
database hit ratios, 1161–1162
latch hit ratio, 1179
library cache hit ratio, 1137
not relying on, 1182
HM_RUN command, 1034
Home Details window, Oracle Universal Installer, 417
home directories
naming conventions, OFA guidelines, 394
home directory, UNIX, 46, 63
HOME shell variable, UNIX, 54
host attribute, USERENV namespace, 580
Host chart, Database Performance page, 1199
HOST command, SQL*Plus, 105, 106, 119
Host CPU chart, Database Control, 1196
Host home page, Grid Control, 160
host names
  connect descriptors, 515
  precedence order for evaluating, 617
host parameter
  easy connect naming method, 529
hosts, Oracle Secure Backup, 786, 788
hot backups see open backups
hot restore, RMAN, 816
hot spares, RAID systems, 88
hot swapping, RAID systems, 88
Hotsos, DBA resources, 14
HR (human resources) schema, 1221
HWM (high water mark)
  Database Control, 151, 153
determining free space, 255
  online segment shrinking, 929
  reclaiming unused space below HWM, 935
hyphen pair notation (--), SQL*Plus, 128
hypothetical ranks and distribution functions, SQL, 1232

IDENTIFIED BY clause, 544, 547, 575
IDENTIFIED EXTERNALLY clause, 576
IDENTIFIED GLOBALLY clause, 576
IDENTIFIED USING clause, role
  authorization, 575
identifiers see keys
idle events, 1181
idle time method, Database Resource Manager, 555
Idle wait class, 1163
IDLE_TICKS system usage statistic, 1181
IDLE_TIME parameter, 549, 553
idle-time-limit resource directive, 560, 561
IE (information exchange) schema, 1222
iee (import-export element), Oracle Secure Backup, 789
INDEXFILE parameter, 706
IMPORT CATALOG command, RMAN, 771, 772
import modes, Data Pump, 705
import parameters, Data Pump, 705–713
import prompt, Data Pump interactive mode, 688
import types, Data Pump, 705
import utilities, 677
  see also Data Pump utilities (Export, Import)
IMPORT_FULL_DATABASE role, Data Pump, 705
IN clause, SQL subqueries, 1067
in memory metrics, 951
IN operator, SQL, 1227
INACTIVITY TIMEOUT parameter, DRCP, 532, 533
INCARNATION option, LIST command, RMAN, 823
incarnations, database, 822
incident, 1026
incident packages
  creating, Support Workbench, 1030–1031
  Diag Incident directory, ADR, 1024
incident packaging service (IPS), 211, 1022, 1026–1028
INCLUDE parameter, Data Pump, 692–693, 708
INCLUDING clause, CREATE TABLE, 279, 280
INCLUDING CONTENTS clause, DROP TABLESPACE, 224, 225, 853
incomplete recovery, 807, 820–824
inconsistent backups, 727
incremental backups, 732, 733
  flash recovery area, 735
  RMAN, 742, 756–757, 778, 779
INDEX hint, 1067
index key compression, 1076
index-organized tables see IOTs
index range scans, 1095
index scans, query optimization, 1052
index segments, 169
index skip scan feature, 1072
indexes
  ALTER INDEX REBUILD command, 217
  bitmap indexes, 1071
  bitmap join indexes (BJI), 1069
  coalescing indexes online, 935
  column data with low cardinality, 1071
  concatenated indexes, 1072
  creating indexes online, 935
  creating tablespaces first, 215
data block sizes and tablespaces, 171
efficient SQL indexing strategy, 1070–1073
EXPLAIN PLAN tool examples, 1093, 1094
function-based indexes, 1066, 1072
index-only plans, 1071
index-organized tables (IOTs), 1072
monitoring index usage, 1073
moving from development to production, 1070
partitioned indexes, 1073
primary indexes, 1070
rebuilding indexes online, 934
rebuilding indexes regularly, 1089
removing unnecessary indexes, 1073
reverse key indexes, 1072
secondary indexes, 1070, 1071
separating table and index data, 170
sizing for database creation, 444
SQL*Loader utility, 644
using appropriate index types, 1071–1073
views in query preventing use of, 1065
what to index, 1071
when to use indexes, 1070–1071
INDEXES parameter, 692
indexes, Oracle, 296–300
bitmap indexes, 301
B-tree index, 298, 301
creating an index, 299–300
DBA_IND_COLUMNS view, 333
DBA_INDEXES view, 333
dropping tables, 276
estimating size of index, 298–299
extracting DDL statements for, 294
function-based indexes, 302
global partitioned indexes, 302
guidelines for creating, 297–298
heap indexes, 300
invisible indexes, 303–304
key-compressed indexes, 301
keys compared, 297
locally partitioned indexes, 303
maintaining, 305
materialized views, 314
monitoring usage, 304–305
OPTIMIZER_USE_INVISIBLE_INDEXES parameter, 463
Oracle index schemes, 298
partitioned indexes, 302–303
performance enhancement
   recommendations, 320
performance trade-off using, 296, 301
rebuilding, 305
reverse-key indexes, 301
special types of, 300–304
SQL Access Advisor recommendations, 303
transparency of, 296
types, 297
INDEX_FFS hint, 1068
INDEXFILE parameter, 706
index-organized tables see IOTs
INDEX_STATS view, 334
INFILE parameter, SQL*Loader, 628, 630
InfiniBand, 94
Information Lifecycle Management (ILM)
   applications, 874
inheritance, 39
init.cssd script, 904
init.ora file
   see also initialization files; PFILE
   activating parameter limits in user profiles, 553
   automatic service registration, 521
   backup guidelines, 729
   cloning databases with RMAN, 834
   creating new database, 177
   creating OMF-based instance, 251
   database instance names, 514
   DB_BLOCK_SIZE parameter, 166
   manual database upgrade process, 437
   OMF redo log files, 250
   Oracle Managed Files (OMF), 250
   post-upgrade actions, 441
   REMOTE_OS_AUTHENT parameter, 615
   setting initialization parameters, post-installation, 424
   upgrading with DBUA, 430
init+asm.ora file, 905
INITCAP function, 657
initial extent, database objects, 169
Initial Options page, SQL Access Advisor, 322
INITIAL_EXTENT parameter, 221
initialization files, 173, 447, 448–449
   see also init.ora file; PFILE
   nesting initialization files, 453
   Oracle looking for correct file, 494, 497
   setting up flash recovery parameters, 739
initialization parameter file see PFILE
initialization parameters, 449–473
ADDM recommendations, 881
archivelog parameters, 459–460
audit-related parameters, 450–451
BACKGROUND_DUMP_DEST, 178
changing for session only, 262
corruption-checking parameters, 470–472
creating Oracle Managed Files, 922–923
detecting data block corruption, 796
DIAGNOSTIC_DEST, 178
dynamic initialization parameters, 177
file-related parameters, 453–454
memory allocation parameters, 456–459
modifying, 262
optimal settings for, 1129
Oracle licensing parameters, 461
Oracle Managed Files (OMF), 248–249, 454–455
performance- and diagnostics-related parameters, 461–468
Pre-Upgrade Information Tool, 428
process-related parameters, 455
recovery-related parameters, 468–470
security-related parameters, 472
session-related parameters, 456
setting security-related initialization parameters, 615
setting, post-installation, 424
undocumented initialization parameters, 473
undo-related parameters, 460–461
upgrading with DBUA, 430
V$SPPARAMETER data dictionary, 177
viewing current values, 473–474
Initialization Parameters page, DBCA, 487
initialization parameters, list of
AUDIT_FILE_DEST, 451
AUDIT_SYS_OPERATIONS, 451, 458
AUDIT_TRAIL, 450
CLIENT_RESULT_CACHE_XYZ parameters, 458
COMPATIBLE, 452
CONTROL_FILE_RECORD_KEEP_TIME, 454
CONTROL_FILES, 453
CONTROL_MANAGEMENT_PACK_ACCESS, 459
CURSOR_SHARING, 466
DB_BLOCK_CHECKING, 471
DB_BLOCK_CHECKSUM, 470
DB_BLOCK_SIZE, 466
DB_CACHE_SIZE, 457
DB_CREATE_FILE_DEST, 455
DB_CREATE_ONLINE_LOG_DEST_n, 455
DB_DOMAIN, 452
DB_FILE_MULTIBLOCK_READ_COUNT, 467
DB_FLASHBACK_RETENTION_TARGET, 469
DB_KEEP_CACHE_SIZE, 457
DB_LOST_WRITE_PROTECT, 470
DB_NAME, 451
DB_nK_CACHE_SIZE, 458
DB_RECOVERY_FILE_DEST, 469
DB_RECYCLE_CACHE_SIZE, 458
DB_SECUREFILE, 472
DB_ULTRA_SAFE, 470
DB_UNIQUE_NAME, 451
DB_WRITER PROCESSES, 455
DIAGNOSTIC_DEST, 449
FIXED_DATE, 449
IFILE, 453
INSTANCE_XYZ parameters, 452
LARGE_POOL_SIZE, 459
LDAP_DIRECTORY_SYSAUTH, 451
LICENSE_MAX_SESSIONS, 461
LICENSE_MAX_USERS, 461
LOG_ARCHIVE_DEST_n, 459
LOG_ARCHIVE_FORMAT, 460
MEMORY_MAX_TARGET, 456
MEMORY_TARGET, 457
NLS_DATE_FORMAT, 453
OPEN_CURSORS, 456
OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES, 462
OPTIMIZER_DYNAMIC_SAMPLING, 463
OPTIMIZER_FEATURES_ENABLE, 463
OPTIMIZER_MODE, 462
OPTIMIZER_USE_INVISIBLE_INDEXES, 463
OPTIMIZER_USE_PENDING_STATISTICS, 463
OPTIMIZER_USE_SQL_PLAN_BASELINES, 464
OS_AUTHENT_PREFIX, 472
PARALLEL_MAX_SERVERS, 467
PLSQL_CODE_TYPE, 464
PLSQL_OPTIMIZE_LEVEL, 468
PROCESSES, 455
QUERY_REWRITE_ENABLED, 465
QUERY_REWRITE_INTEGRITY, 465
REMOTE_LOGIN_PASSWORDFILE, 472
RESULT_CACHE_XYZ parameters, 464
RESUMABLE_TIMEOUT, 470
SEC_XYZ parameters, 465
SERVICE_NAME, 452
SQL_TRACE, 467
STATISTICS_LEVEL, 452
TIMED_STATISTICS, 468
UNDO_XYZ parameters, 460
UTL_FILE_DIR, 454
INITIALIZE_REPLAY procedure, 1213
INITRANS parameter, 346, 1176
inline stored functions, 1074–1075
inline view, 263
inner join, SQL, 1234
INPUT command, SQL*Plus, 130, 131
input/output (I/O)
ADDM determining optimal I/O performance, 884
assigning objects to multiple buffer pools, 189
asynchronous I/O for DBWn process, 182
block size and I/O performance, 1158
cost model of Oracle optimizer, 1060
cost of disk I/O, 186
data block sizes and tablespaces, 171
datafile scattered read wait event, 1177
disk I/O, 246, 1158–1159
disk I/O contention, 1160
distribution of I/O, 1159
disk I/O performance, 1158–1159
eliminating wait event contention, 1208
enforcing per-session I/O limits, 564–565
excessive reads and writes on some disks, 1160
finding inefficient SQL, 1109, 1110
measuring I/O performance, 1159–1161
performing text input and output, UTL_FILE, 258
rebalancing ASM disk groups, 916
reducing I/O contention, 1160
SAME guidelines for optimal disk usage, 1160
saving output to operating system, SQL*Plus, 120
saving user input in variable, SQL*Plus, 121
silent option, SQL*Plus output, 115
specifying directory for processing I/O, 454
system performance, 1204
system usage problems, 1188
tuning shared pool, 1133
viewing output screen by screen, SQL*Plus, 121
input/output redirection, UNIX, 56–57
input/output statistics, UNIX, 82
INSERT ALL statement, 661
insert anomaly, 29
INSERT clause, SQL*Loader, 629
INSERT FIRST statement, 661
INSERT INTO PARTITION clause, 287
INSERT parameter, TKPROF utility, 1102
INSERT statement, SQL, 1225
creating tables, 269
DML statements, 263
table functions, 662, 666
INSERT statement, PL/SQL, 1242
inserting data into tables, 660–662
Install window, Oracle Universal Installer, 420
Installation Guide, Oracle, 392
Installation Type window, Oracle Universal Installer, 417
installations
Instant Client software, 519–520
Oracle Client software, 518
Oracle Internet Directory (OID), 537
installing Oracle Database 11g, 414–422
disk storage requirements, 392
downloading Oracle software, 391, 414, 415–416
final checklist, 413–414
DISPLAY variable, 414
kernel parameters, 413
swap space, 413
temporary space, 413
X Window System emulation, 414
installing on multihomed computer, 398
installing Oracle software, 414–416, 420–422
creating response files, 421
installation types, 417
kernel parameters, 418
operating system requirements, 418
Oracle Universal Installer, 416–420
restarting installation process, 418
runInstaller script, 416
using response files, 421–422
using response files in silent mode, 421, 422
using response files in suppressed mode, 421, 422
using staging directory, 416
introduction, 391
logging in as oracle not root, 414
memory requirements, 393
OFA-compliant directory structure, 399
Optimal Flexible Architecture, 393–400
Oracle Enterprise Edition CDs, 414–415
Oracle home directory, 396
Oracle Installation Guide, 392
post-installation tasks, 422–425
Oracle owner tasks, 424–425
system administrator tasks, 423–424
preinstallation tasks, 400–413
checking preinstallation requirements, 400–401
Oracle owner tasks, 410–413
system administrator tasks, 401–410
products installed with 11.1 release, 401
README files, 392
Release Notes and Addendums, 392
reviewing documentation, 392
uninstalling Oracle, 425–426
instance attribute, USERENV namespace, 580
Instance Efficiency section, AWR reports, 969
instance failure, benefits of archivelog
mode, 726
instance mode, ADDM, 882
instance performance
analyzing instance performance, 1164–1181
analyzing waits with Active Session History, 1169
collecting wait event information, 1174–1175
Compare Periods Report, 1206
Database Control examining, 1195–1201
database hit ratios, 1161–1162
database load affecting, 1205
database wait statistics, 1162–1163
eliminating wait event contention, 1208
long-running transactions affecting, 1202
measuring instance performance, 1161–1194
memory affecting, 1205
objects with highest waits, 1170
obtaining wait information, 1167–1168
redo logs affecting, 1205
segment-level statistics, 1173
SQL statements affecting, 1194
using V$ tables for wait information, 1165–1166
V$ACTIVE_SESSION_HISTORY view, 1169
V$SESSION_WAIT_HISTORY view, 1168
wait classes and wait events, 1163
wait classes and wait-related views, 1171
wait events affecting, 1206
instance recovery
Oracle recovery process, 804–805
instance tuning, 1129–1130, 1194–1209
see also instance performance
INSTANCE_NAME parameter, 452
automatic service registration, 521
database instance names, 514
INSTANCE_NUMBER parameter, ADDM, 883
instances, 173
  see also instance performance
altered properties of, 262
creating OMNI-based instance, 251
database consistency on restart, 184
database instance names, 514
faster instance startup, 805
memory allocation, 187
object-oriented database model, 39
specifying database or ASM instance, 452
starting Oracle instance, 477–480
INSTANCE_TYPE parameter, 452
ASM, 905
Instant Client software, 518, 519–520
instant protection mode, Oracle Data Guard, 800
INSTR function, SQL, 1228
integrated systems management, OEM, 139
integrity, 306–310
control files, 175
integrity constraint states, 308–309
integrity constraints, 306–310
integrity rules, enforcing, 465
interactive mode, Data Pump, 687–688, 701–705
interactive mode, UNIX, 54
interinstance locking, 186
internal locks, 351
INTERNAL_XYZ resource plans, 559
International Oracle Users Group (IOUG), 13
interrupts, 80
intersection operation, 21
INTERSECTION operator, SQL, 1228
INTERVAL clause, CREATE TABLE, 283
INTERVAL keyword
collecting operating system statistics, 1061
jobs, Oracle Scheduler, 999
INTERVAL parameter, AWR snapshots, 961, 964, 965
interval partitioning, 282–283
interval-list partitioning, 289
interval-range partitioning, 290
INTO TABLE clause, SQL*Loader, 628
invalid objects
manual database upgrade process, 439
upgrading with DBUA, 431
INVENTORIES table
interpreting EXPLAIN PLAN output, 1093
Inventory Contents tab, Oracle Universal Installer, 426
inverse percentile functions, SQL, 1231
INVISIBLE clause, CREATE INDEX, 304
invisible indexes, 303–304, 463
invited addresses, securing network, 614
invoker’s rights, stored procedures, 573
I/O see input/output (I/O)
IOSEEKTIM statistic, 1061
iostat utility, UNIX, 82, 1159
IOTFRSPEED statistic, 1061
IOTs (index-organized tables), 265, 278–280
Flashback Transaction Query, 371, 374
IOWAIT.Ticks system usage statistic, 1182
IP (Internet Protocol) address, 47, 78
ip_address attribute, USERENV namespace, 580
IPS (incident packaging service), 211
IPS CREATE PACKAGE command, 1027
is_grant parameter, CREATE_ACL procedure, 616
ISO transaction standard, 342
isolation levels, transactions, 342–346
isolation property, transactions, 340
%ISOPEN attribute, PL/SQL, 1246
Ixora, 14

Java, Oracle and, 1252–1254
Java Database Connectivity see JDBC
Java pool, 187, 193, 1148
Java programs, error handling, 540
Java stored procedures, 1252
java.lang class, 538
JAVA_POOL_SIZE parameter, 193, 195
JDBC (Java Database Connectivity), 537–542, 1253
JDBC drivers, 538
JDBC Supplement, Instant Client packages, 520
Job Activity page, Grid Control, 159
job classes, Oracle Scheduler, 996, 1011–1013
job commands, RMAN, 757–758
job coordinator, Oracle Scheduler, 997
job queue coordination (CJQO) process, 181, 185
job scheduling, 77, 554
job table, Oracle Scheduler, 997
job worker processes, Oracle Scheduler, 997
JOB_ACTION attribute
CREATE_JOB procedure, 999
embedding jobs in chains, 1010
Scheduler managing external jobs, 1002
Scheduler managing programs, 1006
JOB_CLASS_NAME attribute,
CREATE_JOB, 1012
JOB_NAME attribute, CREATE_JOB, 999
JOB_NAME parameter, Data Pump, 699, 705, 708
JOB_QUEUE_PROCESSES parameter, 673, 1056
jobs, Data Pump viewing, 713, 714
jobs, Oracle Scheduler, 994
  administering, 1000
  assigning job priority levels, 996
  belonging to job classes, 1011
  CREATE_JOB procedure, 1005
  creating, 998–999
  creating event-based jobs, 1010
  default scheduler jobs, 1019
  embedding jobs in chains, 1010
  frequency (FREQ keyword), 999
INDEX

1295

- grouping sets of jobs, 996
- managing, 998–1000
- managing external jobs, 1002–1006
- monitoring Scheduler jobs, 1017–1019
- prioritizing jobs, 1016–1017
- purging job logs, 1019
- repeat interval (INTERVAL keyword), 999
- resource plans and windows, 1013
- specifying repeat interval, 999–1000
- types of Scheduler jobs, 995–996

JOB_TYPE attribute
- CREATE_JOB procedure, 999
- embedding jobs in chains, 1010
- Scheduler managing external jobs, 1002
- Scheduler managing programs, 1006

- join command, UNIX/Linux, 67
- join operations, 21
  - heuristic strategies for query processing, 1046
  - using correct joins in WHERE clauses, 1066
- join views, 313
- joins, 1233–1234
  - bitmap join indexes (BJI), 1069
  - Cartesian joins, 1068, 1232
  - CBO choosing join method, 1052
  - CBO choosing join order, 1053
  - equi joins, 1066, 1233
  - EXPLAIN PLAN tool examples, 1094
  - guidelines for creating indexes, 297
  - hash join, 1052, 1068
  - indexing strategy, 1071
  - inner join, 1234
  - lossless-join dependency, 34
  - MERGE join, 1068
  - natural join, 1233
  - nested loop join, 1052
  - outer join, 1234
  - selecting best join method, 1068
  - selecting best join order, 1070
  - self join, 1233
  - sort-merge join, 1052

- JVM (Java Virtual Machine), 1252

- keep buffer pool, 189, 1146, 1147
- KEEP clause, ALTER TABLESPACE, 231, 232
- keep pool, 457
- KEEP procedure, DBMS_SHARED_POOL, 1138
- KEEP XYZ clauses, BACKUP command, RMAN, 780, 781
- kernel, UNIX, 45
- preinstallation, 401, 402, 404–406, 413
- key-compressed indexes, 301
- keys, 26
  - indexes compared, 297
  - multivalued/composite keys, 31
- tables, 20
- keyword variables see shell variables, UNIX
- kill command, UNIX/Linux, 75, 620

- KILL SESSION command, 620
- KILL_JOB command, Data Pump, 686, 703, 704, 713
- KILL_SESSION switch group, 55, 56, 565
  - see also shells, UNIX

- L
- -L option (no-prompt logon), SQL*Plus, 115
- label-based security policy, 586
- large objects, ORDBMS model, 40
- large pool, 187, 193, 1148
- LARGE_POOL_SIZE parameter, 193, 195, 459
- LAST_DDL_TIME value, 329
- latch contention
  - eliminating contention, 1208, 1209
  - parsing, 1141
  - reducing parse-time CPU usage, 1175
  - soft parsing, 1141
  - tuning shared pool, 1133
  - wait events, 1206
- latch free wait events, 1179–1180
- latch hit ratio, 1179
- latch rate, 1183
- latches, 343, 351, 1179
  - library cache latch, 1180
  - shared pool latch, 1180
  - tuning shared pool, 1133
  - V$LATCH view, 1168
- LDAP (Lightweight Directory Access Protocol), 512, 534, 602, 603
- LDAP_DIRECTORY_SYSAUTH parameter, 451, 615
- LD_LIBRARY_PATH variable, 446, 475
- leaf blocks, B-tree index, 298
- least recently used (LRU) algorithm, 181, 188
- Legato Single Server Version (LSSV), 746
- LENGTH function, SQL, 1228
- LGWR see log writer
- library cache, 191
  - aging out SQL statements, 1134
  - avoiding ad hoc SQL, 1141
  - measuring library cache efficiency, 1137–1138
  - optimizing library cache, 1138–1142
  - pinning objects in shared pool, 1142–1143
  - tuning shared pool, 1133–1134
- library cache hit ratio, 1137
- library cache latch, 1180
- LICENSE_MAX_XYZ parameters, 461
- licensing, 149, 948
- Lightweight Directory Access Protocol see LDAP
- lightweight jobs, Oracle Scheduler, 996, 1000–1002
- LIKE operator, SQL, 1224, 1227
- limited analysis mode, Segment Advisor, 930
- limits.conf file, 406
- LINESIZE variable, SQL*Plus, 108
- linking files, UNIX, 57–58
- links see database links
Linux, 43, 44, 45

see also UNIX

Linux commands see UNIX commands

LIST ARCHIVELOG ALL command, RMAN, 761
LIST BACKUP command, RMAN, 760, 810
LIST CHAINED ROWS clause, 279
LIST command, RMAN, 810
LIST command, SQL*Plus, 128
LIST COPY command, RMAN, 760, 810
LIST FAILURE command, Data Recovery Advisor, 830
LIST GLOBAL SCRIPT NAMES command, RMAN, 761
LIST INCARNATION command, RMAN, 823
LIST partitioning, 284, 288–289
LIST SCRIPT NAMES command, RMAN, 750, 761

listener commands, 522–523

listener.ora file, 206, 520, 521, 523
ADMIN_RESTRICTIONS parameter, 614
cloning databases with RMAN, 835
dynamic service registration and, 183
failed attempt to stop, 524
hsnrctl checking listener status, 521
lsnrctl commands, 521–523
multiple listeners, 523
Oracle networking and, 520
QUEUESIZE parameter, 523
READY status, 522
script to start and stop, 500
securing, 614
setting password for, 524–525
setting queue size, 523
UNKNOWN status, 522
upgrading with DBUA, 432

listing commands, RMAN, 750–751
listing files, UNIX, 58
LIVE workspace, 387

In command, UNIX/Linux, 58
load balancing, ASM, 901
LOAD DATA keywords, SQL*Loader, 629
LOAD parameter, SQL*Loader, 634
Load Profile section, AWR reports, 969, 1205
LOAD WHEN clause, 648
loading data see ETL (extraction, transformation, loading)
LOAD_PLANS_FROM_CURSOR_CACHE function, 1082
LOAD_PLANS_FROM_SQLSET function, 1081
loads column, V$SQL view, 1109
load-then-transform method, ETL process, 626
LOBs (large objects)
  data block sizes and tablespaces, 171
  transportable tablespaces, 716
  treating large objects as SecureFiles, 472
local commands, SQL*Plus, 103
LOCAL extent management, 219, 220
local external jobs, 1002
local naming method, connections, 525–529
local partitioned indexes, 1073
local servers, copying files from/to, 254
LOCAL value, extent management, 221
localconfig command, 904
locally managed tablespaces, 172
  AUTOALLOCATE option, 219
  automatic segment space management, 219
  managing storage extents, 222
  MAXEXTENTS parameter, 221
  migrating from dictionary-managed, 217
  proactive space alerts, 226
  specifying default storage parameters, 220
  using bigfile tablespaces, 237
locally partitioned indexes, 302, 303
LOCATION parameter, external table layer, 648
location transparency
  Oracle Net Services features, 511
  using synonyms, 325
lock (LCKn) process, 186
lock conversion, 348
lock escalation, 347, 348
LOCK TABLE statement, 263, 350, 351
locking
  committing transactions, 339
  data concurrency, 199, 341
  database authentication, 597, 598
  exclusive lock mode, 199
  explicit locking in Oracle, 351–352
  explicit table locking, 350–351
  how Oracle processes transactions, 197
  interinstance locking, 186
  locking issues, 1189
  locking-related views, 353
  optimistic locking methods, 347
  Oracle locking, 347, 348
  page-level locking, 348
  pessimistic locking methods, 347
  queries, 349
  SELECT statement, 349
  serializable isolation level, 344, 346
  share lock mode, 199
temporary tables, 278
locks
  blocking locks, 351–352
data dictionary locks, 351
  Database Control managing session locks, 354–355
  DDL locks, 350
deadlocks, 352
distributed locks, 351
DML locks, 349–350
exclusive locks, 349
granularity, 348
identifying sessions holding locks, 353
identifying source of lock, 354
internal locks, 351
latches, 351
locks on suspended operations, 385
managing long transactions, 386
managing Oracle locks, 353, 355
Oracle lock types, 348–350
Oracle locks, 347
releasing locks, 348
row exclusive locks, 349
row-level locks, 349
table-level locks, 349
using SQL to analyze locks, 353–354
views analyzing locks, 354
log buffer space wait event, 1180
LOG FILE parameter, 648
log file switch wait event, 1180
log file sync wait event, 1181
log files
alert log file, 16, 177
archiving redo log files, SQL*Plus, 135
create materialized view log, 318
Data Pump utilities, 681
flashback database logs, 854
managing logging of redo data, 227
Pre-Upgrade Information Tool, 428
RMAN redirecting output to, 747
SQL*Loader utility, 638–639
LOG parameter
RMAN redirecting output to log files, 747
SQL*Loader control file, 635
log sequence-based recovery
Flashback Database example, 860
incomplete recovery using RMAN, 821
log%t_%s_%r format, archived redo logs, 823
log writer (LGWR), 180, 182–183
background process, 982
committing transactions, 198, 199
group commits, 199
how Oracle processes transactions, 197
redo log buffer, 192
starting Oracle instance, 479
LOG_ARCHIVE_DEST parameter, 738
LOG_ARCHIVE_DEST_n parameter, 459, 485, 492
flash recovery, 735, 736, 738, 739
Oracle Managed Files (OMF), 250, 925
Oracle Streams, 673
LOG_ARCHIVE_DUMPDEST parameter, 738
LOG_ARCHIVE_FORMAT parameter, 460, 492, 823
LOG_ARCHIVE_MAX_PROCESSES parameter, 184
LOG_ARCHIVE_MIN_SUCCEED_DEST parameter, 729
LOG_ARCHIVE_START parameter, 493
LOG_BUFFER parameter, 192, 196, 1180
Logfile parallel wait event, 1204
LOGFILE parameter, Data Pump, 690, 706
LOG_FILE_NAME_CONVERT parameter, RMAN, 834, 836
logging
default Oracle logging, 588
Flashback Transaction Query, 374
supplemental logging, 842
LOGGING clause, CREATE TABLE, 227
logging in/out, Oracle
  enabling/disabling password case sensitivity, 465
FAILED_LOGIN_ATTEMPTS parameter, 550, 599
locking accounts, 598
performance-related issues, 1142
sec_max_failed_login_attempts parameter, 615
security when logging in as different user, 620
settings from CREATE PROFILE statement, 548
Single Sign-On feature, 603
specifying maximum number of attempts at, 465
logging in/out, UNIX, 46, 48
remote login (Rlogin), 78
running processes after logging out, 75
logging in, SQL*Plus, 103, 115
LOGGING_LEVEL attribute, 1012, 1019
LOGGING_XYZ values, DBMS_SCHEDULER, 1012
LOG_HISTORY attribute, 1012
logical backups, 728
logical change record (LCR), Oracle Streams, 671
logical database design, 24–34
  converting into physical design, 34, 35
  entity-relationship (ER) modeling, 24–26
  ER modeling tools, 34
  normal forms, 29–34
  normalization, 28–29
  transforming ER diagrams into relational tables, 35
logical database structures, 165–172
  data blocks, 166–169
  extents, 166, 169
  links to datafiles, 174
  schema, 165
  segments, 166, 169
tablespace, 166, 170–172
logical DBA, 8
LOGICAL keyword, RMAN, 756
logical operators, SQL, 1227
logical reads, 1144
finding inefficient SQL, 1109
logical standby databases, 799
logical time stamp, 199
logical unit number (LUN), 901
Logical Volume Manager (LVM), 88, 170, 900
logical volume stripe sizes, disk I/O, 1159
logical volumes, UNIX, 88
LOGICAL_READS_PER_XYZ parameters, 549, 553
.login file, UNIX, 54, 55, 406
login screen, Database Control, 142
login scripts, changing preinstallation, 406
login.sql file, 110–111, 114, 119
LogMiner utility, 207, 841–847
analyzing redo logs, 845–847
archived redo logs, 184
correcting human error, 802
description, 808
extracting data dictionary, 843
LogMiner session, 844–845
naming transactions, 340
precision recovery using, 841
supplemental logging, 842–843
undoing SQL statements, 373
logon/logoff triggers, 591
Logout page, Grid Control, 159
logs see log files
LONG variable, SQL*Plus, 108
long-term backups, RMAN, 780
lookup tables, 35
LOOP/END LOOP statements, PL/SQL, 1243
looping, PL/SQL, 1243–1244
looping, UNIX, 72–73
lossless-join dependency, 5NF, 34
lost-update problem, 341, 345
LOWER function, SQL, 302, 1228
LOW_GROUP resource consumer group, 558, 565
LPAD function, SQL, 1228
lread column, sar command, 83
LRU (least recently used) algorithm, 181, 188
ls command, UNIX/Linux, 58, 59
LSNRCTL STATUS command, ASM, 906
lsnrctl utility, Oracle listener, 516, 520
  checking listener status, 521
  listener commands, 522–523
  set password clause, 524
LUN (logical unit number), 901
LVM (Logical Volume Manager), 88, 170, 900
lwrit column, sar command, 83
MANAGE SCHEDULER privilege, 997, 1013, 1015
manageability monitor see MMON
manageability monitor light (MMNL), 181, 185, 971
managed files
  Oracle Managed Files (OMF), 247–253
management advisors see advisors
management advisory framework see advisory framework
Management Agent, OEM, 154, 156
Management Options window
  Database Upgrade Assistant, 432
  DBCA creating database, 487
Management Pack Access page, Database Control, 149
Management Packs, licensing, 149
Management Repository, OEM, 154, 157, 159
Management Service (OMS), 154, 157
Management System page, Grid Control, 159
manager_policy security policy, 585
managing resources see resource management
managing users, 544, 619, 620
manual database creation, 474–486
  creating data dictionary objects, 485
  creating database, 480–485
  init.ora file, 475–477
  privileges, 475
  quick way to create database, 485–486
  setting OS variables, 474–475
starting Oracle instance, 477–480
manual database upgrade process, 427, 434–441
  backing up database, 437
  catdwgrd.sql script, 434, 442
  catupgrd.sql script, 434, 438, 440
  catuppst.sql script, 438, 439
  checking for invalid objects, 439
  copying init.ora (parameter) file, 437
  creating spool file, 434
  ending spool file, 441
  list of steps for upgrading, 434
  ORACLE_HOME variable, 437
  password file, 437
  Post-Upgrade Status tool, 440–441
  Pre-Upgrade Information Tool, 435–437
  recompiling and validating invalidated objects, 439
  restarting instance, 438
restarting new database, 441
running post upgrade actions script, 439
running upgrade actions script, 438
starting up new database, 437–438
STARTUP UPGRADE command, 437
Sysaux tablespace, 438
upgrade and downgrade scripts, 434
upgrade.log spool file, 435
utlrp.sql script, 434, 439, 440
utlu111i.sql script, 434, 435
utlu111s.sql script, 434, 440
utluppset.sql script, 434
manual management mode, PGA memory, 195
manual optimizer statistics collection, 900
MANUAL parameter, segment space management, 217
manual PGA memory management, 894
manual segment shrinking, 928–929
manual shared memory management, 894
manually cloning databases, 839–840
many-to-many (M:M) relationship, 26
MAP_ALL procedure, 994
mapping data, Data Pump, 693, 709–711
mapping files, 993–994
mapping structures, 993
MARKUP command, SQL*Plus, 134
markup option (-M), SQL*Plus, 115
master encryption key, 607
master process, Data Pump, 685–686
master tables
Data Pump, 685, 686
materialized views, 314
materialized views, 314–320
aggregations, 315
creating, 317–320
data manipulation statements, 314
DBA_MVIEWS view, 333
dropping, 320
EXPLAIN_XYZ procedures, 317
improving SQL processing, 1077
indexes, 314
optimizer statistics, 320
optimizing, 315
privileges, 571
query rewriting, 315
refresh modes/options, 316
refreshing, 314, 316–317
rewrite integrity, 316
REWRITE_OR_ERROR hint, 315
SQL Access Advisor, 320–324
TUNE_MVIEW procedure, 317
using DBMS_MVIEW package, 317
view resolution, 314
MAX function, SQL, 1229
MAX_AUTO_SQL_PROFILES parameter, 1118
MAX_DUMP_FILE_SIZE parameter, 958, 1101
MAX_ESTIMATED_EXEC_TIME directive, 943
MAXEXTENTS parameter, 221
MAX_IDLE_BLOCKER_TIME parameter, 560, 561
MAX_IDLE_TIME parameter, 560, 561
maximum availability mode, Oracle Data Guard, 800
maximum extents errors, 383
maximum performance mode, Oracle Data Guard, 800
maximum protection mode, Oracle Data Guard, 800
MAX_LIFETIME_PER_SESSION parameter, 532
MAXSETSIZE parameter, RMAN backups, 781
MAX_SIZE parameter, DRCP, 533
MAXSIZE parameter, tablespaces, 224
MAX_SQL_PROFILES_PER_EXEC parameter, 1117
MAX_THINK_TIME parameter, DRCP, 533
MAXTHR statistic, 1061
MAX_USES parameter, Scheduler, 1003
MAX_USES_PER_SESSION parameter, DRCP, 532
MBRC statistic, 1061
MCP (Master Control Process), 685–686
md_backup/md_restore commands, 911, 912, 913
media corruption, detecting, 795
media failures, 802, 803
media families, Oracle Secure Backup, 789
Media Management Layer, (MML), 742, 744, 745
media management solutions, 745, 746
media managers
Oracle Secure Backup, 785
RMAN, 743
media recovery, 806–808
block media recovery (BMR), 808, 864–865
data repair, 804
media recovery scenarios
control file recovery, 824–828
datafile recovery, 818–820
without backup, 828–829
incomplete recovery, 820–824
tablespace recovery, 817–818
whole database recovery, 814–817
media server, 786
medium transport element (mte), 789
memory
see also buffers
cost of disk I/O, 186
creating SPFILE or PFILE from, 497
in memory metrics, 951
instance performance, 1205
least recently used (LRU) algorithm, 188
measuring process memory usage, 1190–1191
memory allocation, 187
modifying data, 181
operating system physical memory, 1158
program global area, 187, 193–196
SHOW PG command, 117
system global area (SGA), 187, 187–193
tuning buffer cache, 1144–1148
tuning Java pool, 1148
tuning large pool, 1148
tuning Oracle memory, 1132–1152
hard parsing and soft parsing, 1135–1143
tuning PGA memory, 1148–1152
tuning shared pool, 1133–1135
tuning streams pool, 1148
understanding main memory, 186–187
virtual memory, 1158
Memory Access Mode, 1198
Memory Advisor, 976, 1132, 1145, 1178
memory allocation, 194, 457
memory allocation parameters, 456–459
memory buffers, 187, 188
   CLEAR BUFFER command, SQL*Plus, 115
   writing data to disk, 183
memory management
   automatic, 456, 195–196, 894–897
   PGA memory management, 1148–1149
   operating system, 1186
memory management, UNIX, 81, 82
memory manager (MMAN), 181, 185
memory requirements
   ensuring sufficient memory allocation, 445
   estimating, preinstallation, 400
   installing Oracle Database 11g, 393
memory structures see Oracle memory structures
Memory window, DBCA, 488
MEMORY_MAX_TARGET parameter, 456, 895, 896, 897
MEMORY_REPORT function, 1123
MEMORY_TARGET parameter, 195, 457
   adjusting memory allocation, 1132
   automatic memory management, 456, 895, 896, 897
   defining maximum value of, 456
   ensuring sufficient memory allocation, 446
   managing result cache, 1120
   managing/monitoring database, 214
   sizing shared pool, 1142
   system global area (SGA), 187
MERGE join, 1068
MERGE PARTITION command, 291
MERGE statement, 263, 656, 658–660
upserts, 626, 658
workspaces, 388
merging
   sort-merge join, 1052
messages
   object-oriented database model, 39
   Oracle Streams Advanced Queuing, 670
   sending message to screen, SQL*Plus, 121
metadata
data dictionary, 203, 204
DBMS_METADATA package, 294, 680
exporting dictionary metadata for
tablespaces, 718
exporting using Data Pump, 721
exporting using external tables, 653
importing from dump file, 719
importing using Data Pump, 723
RMAN, 744, 766
metadata filtering, 692
METADATA_ONLY value, Data Pump, 691, 692, 695
MetaLink
   database security, 617
   DBA training, 15
   linking Database Control to, 150
METHOD_OPT attribute, 1055, 1086
methods
   object-oriented database model, 39
   ORDBMS model, 40
metric groups, 950
metrics, 950–952
   baseline metrics, 953–959
   data dictionary views, 958
midrange servers, 45
migrating databases to ASM, 919–921
Millsap, Cary, 1145, 1161
MIN function, SQL, 1229
MINEXTENTS parameter, 221, 957
MINIMIZE XYZ options, RMAN, 777
MIN_SIZE parameter DRCP, 533
MINUS operator, SQL, 1228
mirroring
   ASM mirroring, 901, 909, 914
   fast mirror resync feature, ASM, 908–909
   mirroring vs. multiplexing, 982
   RAID 0+1: striping and mirroring, 90
   RAID 1: mirroring, 89
MIXED_WORKLOAD_PLAN resource plan, 558
mkdir command, UNIX/Linux, 62
mkstore command, creating Oracle Wallet, 241, 609
MML (Media Management Layer), RMAN, 742, 744, 745
MMNL (manageability monitor light), 181, 185, 971
MMON (manageability monitor), 181, 185
   ABP process, 1022
   AWR statistics, 959
   database metrics, 950
   in memory metrics, 951
   managing ADDM, 881
   proactive tablespace alerts, 956
   running ADDM, 885
   saved metrics, 952
   tablespace space alerts, 226
MODE parameter, LOCK TABLE statement, 351
MODEL clause, transforming data, 656, 667–670
modeling, entity-relationship (ER), 24–26
modes
   archivelog mode, 184, 459
   normal mode, 502
   OPTIMIZER_MODE parameter, 462
   restricted database mode, 501
MODIFY_SNAPSHOT_SETTINGS procedure, 882
monitoring, 138
   DBA role, 5, 15
   efficient monitoring, 213–214
   system monitor (SMON) process, 184
MONITORING USAGE clause, 305
more command, UNIX/Linux, 52, 58
MOUNT option, STARTUP command
configuring Flashback Database, 855
starting ASM instances, 907
starting up database from SQL*Plus, 498
mount points, 86, 394
creating databases, 174
creating, preinstallation, 404
OFA guidelines, 394, 396, 399, 404
mounting database, 482, 492
MOVE command, 273, 275
MOVE TABLESPACE option, 935
moving-window aggregate functions, SQL, 1231
MREADTIM statistic, 1061
msps column, iostat command, 82
mte (medium transport element), 789
MTTR (mean time to recover), 805
MTTR Advisor, 976, 981, 1132
multicolumn statistics, 1058
multidimensional arrays, 668
multiplexing
backup guidelines, 729
Connection Manager feature, 512
flash recovery area, 735
mirroring vs. multiplexing, 982
online redo logs, 982
redo log files, 176, 182, 184, 444, 729
session multiplexing, 180
shared server architecture, 512
multitable inserts, 626, 660–662
MULTITHREADING parameter,
SQL*Loader, 641
multivalued dependency, 4NF, 34
multiversion concurrency control system, 347
manual upgrade process, 429
mv command, UNIX/Linux, 59

NAME parameter, ALTER SESSION, 385
NAMES parameter,
TRANSACTION_BACKOUT, 379
naming connections, 525
directory naming method, 534–537
easy connect naming method, 529–530
external naming method, 533–534
local naming method, 525–529
naming context, 536
NAS (Networked Attached Storage), 94
natural join, SQL, 21, 1233
NCA Configuration Assistant), 516, 528–529
NDMP (Network Data Management Protocol), 788
nested loop (NL) method, 1068
nested loop join, 1052
nested subquery, 263
nested table, abstract data types, 1240
Net Configuration Assistant see NCA
Net Service Name Configuration page, 528
net service names, 100, 525
Net Services see Oracle Net Services
netca.rsp response file template, 422
netstat utility, UNIX, 85
network administration files, 173
Network Configuration window, DBUA, 432
Network Data Management Protocol (NDMP), 788
network directory, ORACLE_HOME, 395
network export, Data Pump initiating, 698
Network Information Service (NIS), 533, 534
network mode, Data Pump, 679
network monitoring, UNIX, 85
network performance, 1160
Network wait class, 1163, 1164
Networked Attached Storage (NAS), 94
NetWorker, 745
networking see Oracle networking
NETWORK_LINK parameter, 698, 711–713, 717
networks
fine-grained network access control, 615–618
problems affecting performance, 1203
securing, 614
NEVER option, materialized views, 317
NEWFILE procedure, DBMS_LOGMNR, 844
NEW_LINE procedure, UTL_FILE, 257
NEWPAGE variable, SQL*Plus, 108
NEXT_EXTENT storage parameter, 221, 222
nextval pseudo-column, sequences, 327
NFS file systems, 394
NI column, top command, 84
NIS (Network Information Service), 533, 534
NLS_DATE_FORMAT parameter, 453
NLS_TERRITORY parameter, 453
NO LOGGING option, backup guidelines, 729
NO SALT option, ENCRYPT keyword, 608
no workload mode, 1061
noarchivelog mode, 176, 491, 492
see also archivelog mode
partial database backups, 794
reasons for using, 726
whole database backups, 790
NOAUDIT keyword, 589
nobody, verifying unprivileged user exists, 408
nocascade/nocascade_force options, 869
noclobber shell variable, 57
noconflict_only option, 869
nodes, B-tree index, 298
NOFILENAMECHECK option, RMAN, 835
nohup option, shell programs, 75
NOLOG option, connectionless SQL*Plus, 101
NOLOGFILE parameter, Data Pump, 691, 706
NOLOGGING clause
CREATE TABLESPACE statement, 227
CREATE TABLE AS SELECT command, 273
deriving data from existing tables, 657
SQL*Loader utility, 644
nologging option, redo log buffer, 193
NOMOUNT option, STARTUP command, 477
starting ASM instances, 907
starting up database from SQL*Plus, 498
NONE value, AUDIT_TRAIL parameter, 587
nonprefixed indexes, 1073
nonrepeatable (fuzzy) read problem
data concurrency, 342
isolation levels, 344, 345, 346
nonunique indexes, 297
no-prompt logon option (-L), SQL*Plus, 115
NO_RESULT_CACHE hint, 1122, 1125
normal forms, 29–34
Boyce-Codd normal form (BCNF), 33
Fifth Normal Form (5NF), 34
First Normal Form (1NF), 30–31
Fourth Normal Form (4NF), 34
Second Normal Form (2NF), 31–33
Third Normal Form (3NF), 33
normal mode, Oracle optimizer, 1111
NORMAL option, SHUTDOWN command, 502
normal program conclusion, 338
normal redundancy level, ASM, 909, 914
normalization, 28–29
attribute dependence on primary key, 31, 33
denormalization, 34
functional dependencies, 33
lossless-join dependency, 34
multivalued dependencies, 34
non-normalized data, 30
partial dependencies, 31
performance issues, 36
repeating groups, 30
NOT EXISTS operator, subqueries, SQL, 1237
NOT NULL constraint, 306, 307
%NOTFOUND attribute, PL/SQL, 1246
Notification Methods page, Database
Control, 148
notification rules, setting, 955
NOWAIT option, 339, 351
NOWORKLOAD keyword, 1061
null values, 269, 306
NULLIF parameter, SQL*Loader, 642
NUMBER data type, 1222
number functions, SQL, 1229
numberofxids parameter, 869
NUM_CPUS system usage statistic, 1181
numeric data types, 1222
NUMTXNS parameter, 379
NUMWIDTH variable, SQL*Plus, 108
NVL function, SQL, 1230
OFA guidelines, 393–394
administrative files, 397
ADR (Automatic Diagnostic Repository), 396
database files, 398–400
datafiles, 396
directories, 394, 395, 396
flash recovery area, 396, 400
installing on multihomed computer, 398
mount points, 394, 404
naming conventions, 393, 398
product files, 397
OFF option, SPOOL command, SQL*Plus, 120
OFFLINE clauses, tablespaces, 228, 229
OFFLINE IMMEDIATE clause, media failures, 803
OID (Oracle Internet Directory), 534–535
enterprise user security, 611
installing, 537
making database connections, 535–536
organization of, 536–537
OID option, Data Pump Import, 711
oinstall (default name), 407, 408
OL$/OL$HINTS/OL$NODES tables, 1078
OLTAP databases
block sizes and tablespaces, 172
DB_BLOCK_SIZE parameter, 466
indexing strategy, 1071
program global area (PGA), 193
system global area (SGA), 187
table compression, 275, 1076
using indexes, 296
write ahead protocol, 199
OMF (Oracle Managed Files), 247–253
accidentally dropping datafiles, 248
adding tablespaces, 927
adding tablespaces and datafiles, 253
benefits of using, 248, 922
bigfile tablespaces (BFTs), 236
control files, 250, 251, 252, 924
creating database, 250–253, 924–927
creating OMF files, 248–249, 251, 922–923
creating OMF-based instance, 251
creating Sysaux tablespace, 239, 252
creating/locating tablespaces, 252
datafiles, 250, 924
deleting unwanted files, 734
description, 174
file management, 922–927
file types, 250, 924
flash recovery area, 738
init.ora file, 250
initialization parameters, 248–249, 454–455, 922–923
limitations, 248
locating OMF files, 926–927
naming conventions, 922, 923
naming files, 248, 249
operating system files, 248
redo log files, 250, 252, 924
setting up file location parameters, 250, 924
small and test databases, 248
specifying default location for OMF files, 455
starting database instance, 925
OMF parameters
DB_CREATE_FILE_DEST parameter, 247, 249, 250, 923
DB_CREATE_ONLINE_LOG_DEST_n parameter, 247, 249, 250, 923
DB_RECOVERY_FILE_DEST parameter, 247, 249, 250, 923
OMS (Oracle Management Service), 154, 157
ON clause, BREAK command, SQL*Plus, 122
ON COMMIT DELETE/PRESERVE ROWS options, 278
ON COMMIT mode, refreshing materialized views, 316, 319
ON DEMAND mode, refreshing materialized views, 316
ON PREBUILT TABLE clause, 319
one-pass sort, 1149
one-to-many (1:M) relationship, 26
one-to-one (1:1) relationship, 26
online backups, RMAN, 742, 775
online capabilities, Oracle Database 11g, 933–945
data redefinition, 935–941
data reorganization, 933–935
database quiescing for online maintenance, 944–945
dynamic resource management, 941–943
online database block-size changes, 943–944
suspending database, 945
online redo log files, 175, 868, 981–985
creating database, 481
creating/locating OMF files, 926
flash recovery area, 735
making whole closed backups, 791
OMF file-naming conventions, 249, 923
online segment shrinking, 927–928
online table redefinition, 935, 936–941
activity during redefinition process, 939
checking for errors, 940
completing redefinition process, 940–941
copying dependent objects, 939
creating temporary tables, 937
errors occurring during, 941
redefining temporary tables, 938
synchronizing interim and source tables, 940
verifying eligibility of tables, 937
ONLY keyword, encrypting RMAN backups, 782
open account, database authentication, 597
open backups, 726, 728, 792–793
OPEN clause, explicit cursors, PL/SQL, 1245
OPEN option, STARTUP command, 499
open recovery, 807
OPEN_CURSORS parameter, 194, 456
OPEN_WINDOW procedure, 1015
operating system authentication, 601
connecting to RMAN, 746
database security, 612
Oracle Net, 602
operating system copies, RMAN, 752
operating system files
extents, 220
opening operating system file, 257
Oracle Managed Files (OMF), 248
tablespaces, 215
UTL_FILE package, 256–259
Operating System Statistics section, AWR reports, 970
operating systems
applying OS packages, 403
audit parameters, 450, 451
automatic database startup on OS restart, 499
bigfile tablespaces, 237
checking memory and physical space, 403
checking OS packages, 403
checking version, 402
collecting statistics, 1060–1062, 1086
creating additional accounts, 424
creating groups, 406–408
disk I/O performance, 1158
manual collection of statistics, 1054
memory management, 1186
evaluating performance, 1158
Oracle installation requirements, 418
preinstallation checks, 401
saving output to, 120
setting permissions, 613
setting upper limit for OS processes, 455
UNIX/Linux for DBA, 43–95
usage of commands from SQL*Plus, 105, 115, 119
verifying software, 402–403
operations
grouping SQL operations, 1234–1236
resumable database operations, 383, 385
table operations, 267
operator class, Oracle Secure Backup, 788
operators
set operators, 263
SQL/XML operators, 1249
operators, SQL, 1227–1228
LIKE operator, 1224, 1227
OPS$ prefix, authenticated usernames, 472
OPSS$ORACLE database account, 602
opt parameter, COPY command, SQL*Plus, 133
optimal mode operation, 194
optimal sort, 1149
optimistic locking methods, 347
optimization
see also query optimization
CBO (Cost-Based Optimizer), 1047–1053
cost-based query optimization, 1044–1046
materialized views, 315
PPLSQL_OPTIMIZE_LEVEL parameter, 468
optimization phase
query processing, 1043–1046
SQL processing steps, 1133
optimizer see CBO (Cost-Based Optimizer)
optimizer hints, 1051
optimizer statistics
automated tasks feature, 211
automatic collection, 209, 212, 213, 897–899
default values, 1050
description, 1049
dynamic collection of, 1050
extended optimizer statistics, 1058–1060
manual collection, 900
materialized views, 320
Oracle recommendation, 898
providing statistics to optimizer, 1047–1049
using OEM to collect optimizer statistics, 1064–1065
when manual collection is required, 1054
OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES parameter, 462, 1081
OPTIMIZER_DYNAMIC_SAMPLING parameter, 463, 1063
OPTIMIZER_FEATURES_ENABLE parameter, 315, 463, 1078, 1080, 1218, 1219
OPTIMIZER_MODE parameter, 462, 1049–1050, 1051, 1068
OPTIMIZER_USE_INVISIBLE_INDEXES parameter, 463
OPTIMIZER_USE_PENDING_STATISTICS parameter, 463, 1057
OPTIMIZER_USE_SQL_PLAN_BASELINES parameter, 464, 1082
OPTIM_PEEK_USER_BINDS parameter, 1087
OPTIONS attribute,
GATHER_DATABASE_STAT, 1055
OPTIONS clause, SQL*Loader, 633
OPTIONS parameter,
TRANSACTION_BACKOUT, 380, 869
OR REPLACE clause, 313
ORASAUTOTASK_HIGH_SUB__PLAN resource plan, 559
ORASAUTOTASK_SUB__PLAN resource plan, 558
Oracle
dual table, necessity for, 102
Java and, 1252–1254
product_user_profile table, 103, 104
setting environment for SQL*Plus, 98
Oracle administrative files see administrative files
Oracle Advanced Security, 535, 603, 618
Oracle advisors see advisors
Oracle Application Express, 401
Oracle Application Server Single Sign-On, 535
Oracle Backup Solutions Program (BSP), 745
Oracle Backup Web Interface, 785, 788
Oracle base directory, 395, 399, 409
Oracle blocks see data blocks
Oracle blocks, 395, 399, 409
Index

Oracle by Example (OBE), 14
Oracle Certification Program, 11
Oracle Certified Associate (OCA), 10, 12
Oracle Certified Master (OCM), 10, 12
Oracle Certified Professional (OCP), 10, 12
Oracle Change Management Pack, 149, 949
oracle class, Oracle Secure Backup, 788
Oracle Client, 391, 517–519
Oracle Cluster Synchronization Service (CSS), 902–904
Oracle Collaboration Suite, 535
Oracle Configuration Management Pack, 149, 949
Oracle Configuration Manager, 401, 1029
Oracle connectivity see connectivity
Oracle context, 536
Oracle data dictionary see data dictionary
Oracle Data Guard see also standby databases
avoiding data center disasters, 802
DB_LOST_WRITE_PROTECT parameter, 470
creating Oracle Wallet, 241
data protection modes, 800
high-availability systems, 798
standby databases and, 799–800
Oracle Data Guard Broker, 799
Oracle Data Migration Assistant, 427
Oracle Data Pump utility see Data Pump utilities (Export, Import)
Oracle data types, 1222–1223
Oracle database, 173
connecting to, 205
connecting using CONNECT command, 100
efficient managing and monitoring, 213–214
ensuring database compatibility, 452
preparing database for upgrading, 430
providing name for database service, 452
remotely connecting to, 98
script to start and stop, 500
setting database name, 451
setting environment for SQL*Plus, 98
starting SQL*Plus session from command line, 98–100
Oracle Database 11g
downgrading from, 441
downloading Oracle software, 415–416
initialization parameters, 449–473
viewing current values, 473–474
installing Oracle software, 416–420
OFA-compliant directory structure, 399
Oracle Enterprise Edition CDs, 414–415
sample schemas, 1221–1222
SGA and PGA memory allocations, 456
Oracle Database 11g architecture, 165–214
automated tasks feature, 211
automatic database management, 208–209
common manageability infrastructure, 210–213
communicating with database, 205–207
efficient managing and monitoring, 213–214
Oracle Database 11g CD/Client CD, 414
Oracle Database 11g installation see installing Oracle Database 11g
Oracle Database 11g upgrade see upgrading to Oracle Database 11g
Oracle Database Two-Day DBA course, 14
Oracle Database Configuration Assistant see DBCA
Oracle database connectivity see connectivity
Oracle database files see database files
Oracle Database Resource Manager see Database Resource Manager
Oracle database structures, 165–178
alert log file, 177
backup files, 178
tablespace, 173, 174–175
data blocks, 166–169
datafiles, 173–174
extents, 166, 169
initialization files, 173
physical database structures, 173–176
redo log files, 173, 175–176
segments, 166, 169
SPFILE (server parameter file), 177
tablespace, 166, 170–172
trace files, 178
Oracle database transactions see transactions
Oracle database upgrade see upgrading to Oracle Database 11g
Oracle Database Vault, 401, 430
Oracle datafiles see datafiles
Oracle DBA see DBA (database administrator)
Oracle Diagnostic Pack, 149, 949
Oracle Directory Manager, 516, 535
Oracle directory replication server, 535
Oracle directory server, 535
Oracle Enterprise Edition CDs, 414
Oracle Enterprise Manager see OEM
Oracle Enterprise Manager CD, 414
Oracle FAQ, 14
Oracle home directory, 395–396, 399, 410, 417
Oracle iLearning, 11
Oracle indexes see indexes, Oracle
Oracle installation see installing Oracle Database 11g
Oracle installer see Oracle Universal Installer
Oracle instances see instances
Oracle Internet Directory see OID
Oracle Inventory directory, 396, 409
Oracle Inventory group, 407
Oracle Label Security feature, 586
Oracle licensing parameters, 461
Oracle listener see listeners
Oracle LogMiner utility see LogMiner utility
Oracle Managed Files see OMF
Oracle Management Service (OMS), 154, 157
Oracle memory see memory
Oracle memory structures, 186–196
  automatic memory management, 195–196
  database buffer cache, 187, 188–189
  Java pool, 187, 193
  large pool, 187, 193
  memory management configuration options, 196
  multiple database block sizes and buffer cache, 189–190
  program global area (PGA), 187, 193–196
  redo log buffer, 187, 192–193
  shared pool, 187, 191–192
  Streams pool, 187, 193
  system global area (SGA), 187, 187–193
  understanding main memory, 186–187
Oracle MetaLink, 15
Oracle Names, OID and, 535
Oracle Net, 511, 513, 516
  connecting to Oracle, 205
  copying files between databases, 253
  operating system authentication, 602
Oracle Net Configuration Assistant (NCA), 516, 528–529
Oracle Net configuration files, 412
Oracle Net Listener see listeners
Oracle Net Manager, 516, 537
Oracle Net Services, 206, 511–512, 513
  configuring, post-installation, 425
  Oracle components using OID, 535
  Oracle home directory, 395
  tools, 516
Oracle Network Foundation Layer, 513
Oracle networking, 513–516
  see also connectivity
  LDAP-compliant directory server, 512
  listener and, 520
Oracle online learning program, 11
Oracle optimizer see CBO (Cost-Based Optimizer)
Oracle owner
  post-installation tasks, 424–425
  preinstallation tasks, 413–410
Oracle performance statistics see performance statistics, Oracle
Oracle PMON process, 521
Oracle Policy Manager, 586
Oracle processes, 179–186
  archiver, 184
  ASM background (ASMB), 185
  ASM rebalance (ARBn), 185
  background processes, 179, 180–186
  change-tracking writer (CTWR), 185
  checkpoint (CKPT), 183
  continuous processes, 179
  database writer (DBWn), 181–182
  flashback data archiver (FBDA), 186
  job queue coordination (CJQO), 185
  lock (LCKn), 186
  log writer (LGWR), 182–183
  manageability monitor (MMON), 185
  manageability monitor light (MMNL), 185
  memory manager (MMAN), 185
  process monitor (PMON), 183
  rebalance master (RBAL) process, 185
  recoverer (RECO), 185
  recovery writer (RVWR) process, 185
  result cache background (RCBG), 186
  server processes, 179–180
  system monitor (SMON), 184
  user and server process interaction, 179
  user processes, 179
Oracle Protocol Support, 513
Oracle Real Application Clusters see RACs
Oracle recovery process see recovery
Oracle Resource Manager, 147, 996
Oracle Scheduler see Scheduler
Oracle schemas see schemas, Oracle
Oracle Secure Backup, 202, 785–790
Oracle sequences see sequences
Oracle server software, installing, 416–420
Oracle SQL Developer, 136, 401
Oracle Storage Compatibility Program (OSCP), 95
Oracle Streams, 671–675
  avoiding data center disasters, 802
  description, 193
  high-availability systems, 798
Oracle synonyms see synonyms
Oracle tables see tables, Oracle
Oracle tablespaces see tablespaces
Oracle Technology Network (OTN), 13
Oracle tools, 213–214
Oracle transaction management, 337
Oracle transactions see transactions
Oracle triggers see triggers
Oracle Tuning Pack, 149, 949
Oracle Universal Installer
  choosing Basic/Advanced Installation, 416
  End of Installation window, 420
  Install window, 420
  installing Grid Control, 155
  installing OEM Management Agent, 156
  installing OID, 537
  installing Oracle server software, 415
  installing Oracle software, 416–420
  installing Oracle software using response files, 421–422
  invoking with runInstaller script, 416
  prerequisite checks, 418
  restarting installation process, 418
  Select Configuration Options window, 419
  Select Installation Type window, 417
  Specify Home Details window, 417
  Summary window, 419
  uninstalling Oracle, 426
  Welcome window, 416, 417
Oracle University, 11
Oracle upgrade see upgrading to Oracle Database 11g
oracle user
creating Oracle software owner, 408
installing Oracle Database 11g, 414
privileges, 409
setting environment variables, 413
tasks, post-installation, 424
Oracle views see views
Oracle Wallet
creating, 241–242
creating encrypted tablespaces, 242–243
creating with OWM, 605–606
creating, tablesapce encryption, 609
data encryption, 604
ENCRIPTION_WALLET_LOCATION parameter, 241
opening and closing, 606
Oracle Wallet Manager (OWM), 241, 605–606
Oracle Warehouse Builder (OWB), 401, 627
Oracle Web Conference (OWC), 15
Oracle XML DB, 1248–1252
Oracle-Base, 14
ORACLE_BASE variable, 71, 395, 396
setting OS environment variables, 446
setting preinstallation, 409, 411
ORACLE_DATAPUMP access driver, 627, 648, 650, 651
ORACLE_HOME directory, 177, 401, 447
ORACLE_HOME variable
creating database using SQL*Plus, 475
manual database upgrade process, 437
Oracle home directory, 395
remotely connecting to Oracle database, 98
setting environment for SQL*Plus, 98
setting environment variables, preinstallation, 411, 413
setting OS environment variables, 446
upgrading with DBUA, 430
ORACLE_HOSTNAME variable, 398
ORACLE_LOADER access driver, 627, 648
ORACLE_PATH variable, 125
OracleSchedulerExecutionAgent service, 1005
ORACLE_SID variable, 71, 412
creating database using SQL*Plus, 475
DB_NAME parameter, 451
registering database in recovery catalog, 769
setting for ASM instances, 906
setting OS environment variables, 446
specifying database name at STARTUP, 499
starting session from command line, 98
starting SQL*Plus sessions, 99
oradebug utility, 1174
oraenv script, 424, 425
ORAENV_ASK variable, 412
orainst.loc file, 421
orainstRoot.sh script, 155, 156, 420
OracleInventory (Oracle Inventory directory), 396, 409
ORAINVENTORY group, 407, 408
ORAKILL utility, 621
OraPub, 14
orapwd command, 600
oratab file, 411, 423, 424, 425, 430
ORDBMS (object-relational database management system), 19, 38, 39–40
ORDER BY clause, 193, 297, 1071, 1226
ORDER BY command, 263
ORDERED hint, 1067
orderid_refconstraint constraint, 285
O_RELEASE variable, SQL*Plus, 128
ORGANIZATION EXTERNAL clause, 647
ORGANIZATION INDEX clause, 279
OS value, AUDIT_TRAIL parameter, 587, 588
OSASM group, 407, 902
OS_AUTHENT_PREFIX parameter, 472, 601, 612
OSCP (Oracle Storage Compatibility Program), 95
OSDBA group, 407, 408
OSOPER group, 407, 408
OS_USER attribute, USERENV namespace, 579, 580
Other wait class, 1163
OTHER_GROUPS resource consumer group, 558, 562
out-of-space errors, 383
OUT option, SPOOL command, SQL*Plus, 120
outer join, SQL, 1234
outlines, stored, 1077–1080
OUTLN account, 490, 596
OUTLN user, stored outlines, 1078
out-of-space alerts, 226
output see input/output (I/O)
O_VERSION variable, SQL*Plus, 128
overwriting, protecting files from, 57
OWB (Oracle Warehouse Builder), 401, 627
OWM (Oracle Wallet Manager), 241, 605–606
ownership
creating Oracle software owner user, 408–409

P
package privileges, 571
packages
applying/checking OS packages, 403
executing, SQL*Plus, 121
GlancePlus package, 84
PL/SQL, 1247
packs
Server Manageability Packs, 459
PACK_STGTAB_SQLSET procedure, 1218
page command, UNIX/Linux, 53
page ins/outs, memory management, 81
page-level outs, 348
pages
analyzing page performance, Grid Control, 160
swapping data, 1158
PAGESIZE variable, SQL*Plus, 108
paging, 1188, 1205
PARALLEL parameter, Data Pump, 703, 713
parallel degree limit method, 555
parallel execution, 662, 1085
PARALLEL option, 273, 657
PARALLEL parameter
  Data Pump, 678, 700–705, 708
  populating external tables, 652
  SQL*Loader control file, 635
parallel processes, setting number of, 467
parallel recovery feature, 807
PARALLEL_DEGREE_LIMIT_MTH parameter, 560
parallelism
  changing degree of parallelism, 700
  creating plan directives, 560
  Data Pump technology, 677
  degree of parallelism parameters, 615
  production database problems, 554
PARALLEL_MAX_SERVERS parameter, 467
parameter files
  Data Pump Export utility, 687, 690, 704
  database creation, 446–448
  initialization parameter file (PFILE), 447–448
parameters
  Data Pump Export parameters, 689–704
  Data Pump Import parameters, 705–713
  dynamic parameters, 448
  hidden Oracle parameters, 177
  setting security-related initialization parameters, 615
  SHOW PARAMETERS command, 117
  static parameters, 448
parent nodes, B-tree index, 298
PARFILE parameter
  Data Pump, 687, 690, 706
  SQL*Loader utility, 637
parity, RAID, 90
Parse column, TKPROF utility, 1104
parse information, 1135–1136
parse tree, 1043, 1133
parsing
  application scalability, 1141–1142
  bind peeking technique, 1087
  deriving parse information, 1135
  hard parsing, 191, 1135, 1139
  converting to soft parsing, 1141
  latch contention, 1141
  parse-time CPU usage, 1156–1157
  soft parsing, 191, 1135
  stages of SQL processing, 343
  using parsed statement, 1134
parsing stage
  bind variables, 1075
  query processing, 1043
  SQL processing steps, 1133
  partial database backups, 727, 794
  partial dependencies, 2NF, 31
  partial execution of SQL statements
    Automatic Tuning Optimizer (ATO), 1112
    partial mode, ADDM, 882
    PARTIAL option, DURATION clause,
      RMAN, 777
    PARTITION BY HASH clause, 283, 303
    PARTITION BY LIST clause, 284
    PARTITION BY RANGE clause, 282
    PARTITION BY REFERENCE clause, 285
    partition pruning, 280
    PARTITIONED BY SYSTEM clause, 287
partitioned indexes, 302–303, 1073
partitioned tables, 266, 280–292
  adding partitions, 291
  archiving data, 281
  coalescing partitions, 292
  creating, 281
  DBA_PART_TABLES view, 331
  DBA_TAB_PARTITIONS view, 292, 330
  dropping partitions, 291
  exchanging partitions, 291
  improving SQL processing, 1076
  merging partitions, 291
  partition independence, 281
  partition maintenance operations, 290–292
  performance, 280, 281
  renaming partitions, 291
  splitting partitions, 291
partitioning, 280
  composite partitioning, 281, 287–290
  disk partitioning, 87
  hash partitioning, 283–284
  interval partitioning, 282–283
  interval-list partitioning, 289
  interval-range partitioning, 290
  list partitioning, 284
  range partitioning, 281–282, 285
  range-hash partitioning, 288
  range-list partitioning, 288–289
  reference partitioning, 284–286
  system partitioning, 287
  transition point, 282
  VALUES LESS THAN clause, 281
  virtual column-based partitioning, 286
partitioning keys, 281
partitions
  MODEL clause creating multidimensional arrays, 668, 670
passwd command, UNIX/Linux, 49
passwd file, UNIX, 67
PASSWORD command, SQL*Plus, 547
PASSWORD FILE option, RMAN, 835
password management function, 552
password protection features, 552
PASSWORD value, Data Pump, 695, 696
password verification function, 552
password-based encryption, 763
  encrypting RMAN backups, 782
password-related parameters, user profiles, 549, 550
passwords
  see also authentication; security
case sensitivity, 465, 491, 597–598
changing another user’s password temporarily, 620
changing passwords, 611
changing user’s password, 547
database security, 611
DBCA changing passwords for default users, 490–491
default values, 490
encrypted passwords, 601
ENCRYPTION_PASSWORD parameter, Data Pump, 696
expiration, 599
hard-coding user passwords, 611
managing, 596–597
password aging and expiration policies, 612
password authentication, 602
password file, 177
  backup guidelines, 729
database security, 611
  manual database upgrade process, 437
orapwd command, 600
REMOTE_LOGIN_PASSWORDFILE parameter, 472, 599
requirements for strong passwords, 552
resetting passwords, post-upgrade, 441
secure password support, 598
setting password for listener, 524–525
SYS user default password, 475
SYSTEM account default password, 475
Password_Grace_Time parameter, 475
PASSWORD_XYZ parameters, 550
paste command, UNIX/Linux, 67
Patch Advisor, database management, 148
patches
  ACCEPT_SQL_PATCH procedure, 1038
  applying manually, 151
  applying patch sets, 1131
  applying, post-installation, 424
  checking for latest security patches, 617
  Critical Patch Updates, 617
  DBA_SQL_PATCHES view, 1038
  DROP_SQL_PATCH procedure, 1038
  Grid Control, 159
  linking to MetaLink, 150
  preinstallation checks, 401
Patching Setup page, Database Control, 148, 151
PATH shell variable, UNIX, 54
PATH variable, setting, 411, 446
paths, UNIX, 47, 125
pattern matching, SQL*Plus, 129
pattern matching, UNIX, 65
pattern-recognition, grep command, UNIX, 49
PAUSE command, SQL*Plus, 121
PAUSE variable, SQL*Plus, 108
PCTFREE parameter, 217, 222, 1176
PCTSPACE option, Data Pump Import, 711
PCTTHRESHOLD clause, 279, 280
PCTUSED parameter, 217, 222
pending area, Database Resource Manager, 556, 562
pending statistics, making public, 1057
percent full alert, 226
percent sign (%) character, SQL, 1224
percentage of maximum thresholds, 954
performance
  see also indexes; instance performance;
    optimization; system performance; tuning
abnormal increase in process size, 1190–1191
analyzing performance using ADDM, 1183–1184
ASH analyzing recent session activity, 1186
ASM disk groups, 914, 916
Automatic Database Diagnostic Monitor (ADDM), 209
avoiding ad hoc SQL, 1141
AWR statistics for SQL statements, 1184
benefits of tablespaces, 171
bind variables, 1134
buffer cache sizing, 188
cache misses affecting, 192
collecting trace statistics, 1100
confusing symptoms and causes, 1183
cost of disk I/O, 186
Data Pump technology, 678
Database Control examining, 1195–1201
database design, 36
database hangs, 1186–1194
database hit ratios, 1161–1162
database wait statistics, 1162–1163
delays due to shared pool problems, 1191
disk configuration strategies, 87
dynamic performance (VS) views, 204
dynamic performance tables, 204
extent sizing, 216
extents, 220
inline stored functions, 1074–1075
isolation levels, 344
licensing performance tools, 948
locking issues, 1189
logging on and off affecting, 1142
measuring I/O performance, 1159–1161
measuring process memory usage, 1190–1191
minimizing downtime, 6
monitoring host with Grid Control, 160
monitoring system with Grid Control, 160
network performance, 1160
operating system memory management, 1186
Oracle Enterprise Manager (OEM), 206
paging affecting, 1205
partitioned tables, 280, 281
preserving database performance, 1080
problems due to bad statistics, 1191
RAID systems, 88
reclaiming unused space, 1090
reducing I/O contention, 1160
SAME guidelines for optimal disk usage, 1160
scalability, 1141
segment space management, 218
severe contention for resources, 1188
SQL Performance Analyzer, 213, 1216–1220
swapping affecting, 1205
system global area (SGA), 187
system usage problems, 1188
temporary tables, 277
timing conversion to new version, 1131
tracing entire instance, 1107
write classes and wait events, 1163
writing efficient SQL, 1065–1075

period (.) character, regular expressions, SQL, 1238
period-over-period comparison functions, SQL, 1231
permanent files, flash recovery area, 735
permanent tablespaces, 172
creating database, 482
default permanent tablespaces, 235–236
description, 215
permissions, UNIX files, 59–62
setting, database security, 613
setting, preinstallation, 409
pessimistic locking methods, 347
PFILE (initialization parameter file), 446, 447–448
see also init.ora file; initialization files; SPFILE
changing parameter values, 493
changing parameters dynamically, 447
comments in init.ora file, 496
creating init.ora file, 475–477
from SPFILE, 495
creating SPFILE from, 494
creating SPFILE or PFILE from memory, 497
init.ora file not saved in default location, 478
modifying, 495
Oracle looking for correct initialization file, 494
quick way to create database, 485
reading init.ora file, 473
setting archivelog-related parameters, 491
using init.ora file as well as SPFILE, 495
pfile directory, 397
PGA (program global area), 187, 193–196
automatic memory management, 195–196, 894, 896
automatic PGA memory management, 194, 894
managing/monitoring database, 214
manual PGA memory management, 894
MEMORY_TARGET parameter, 457
total memory allocated to, 1190
description, 1130
database security, 613
database tuning, 1041–1043
see also tuning
adaptive cursor sharing, 1087–1090
ADDM and, 877
avoiding improper use of views, 1075
CBO (Cost-Based Optimizer), 1047–1053
database design, 1042
DBA improving SQL processing, 1075–1080
DBA role, 5, 1086–1087
identifying inefficiency in SQL statements, 1127
instance tuning, 1129–1130, 1194–1209
see also instance performance
query optimization, 1047–1065
query processing optimization, 1043–1046
reactive performance tuning, 1042
removing unnecessary indexes, 1073
SQL Plan Management (SPM), 1080–1087
tuning-related advisors, 976
using result cache, 1120–1126
performance tuning tools, SQL, 1090–1105
Autotrace facility, 1095–1099
EXPLAIN PLAN tool, 1090–1095
SQL Trace utility, 1099–1102
TKPROF utility, 1102–1105
performance advisors, ADDM, 881
Performance Analysis section, Database Control, 1197
performance- and diagnostics-related parameters, 461–468
Performance Data Report page, Database Control, 1201–1202
performance monitoring, UNIX, 80–85
analyzing read/write operations, 83
bandwidth, 81
CPU usage, 80
disk storage, 81
GlancePlus, 84
iostat command, 82
memory management, 81
memory use, 82
monitoring network with netstat utility, 85
monitoring performance with top command, 84
monitoring system with GlancePlus, 84
netstat utility, 85
sar command, 83
top command, 84
viewing input/output statistics, 82
vmstat utility, 82
Performance page, Database Control, 145–146
performance statistics, 959
AWR, 210, 877, 878, 960
performance statistics, Oracle, 948–952
performance tuning, 1041–1043
see also tuning
adaptive cursor sharing, 1087–1090
ADDM and, 877
avoiding improper use of views, 1075
CBO (Cost-Based Optimizer), 1047–1053
database design, 1042
DBA improving SQL processing, 1075–1080
DBA role, 5, 1086–1087
PGA memory, 1148–1152, 1205
PGA_AGGREGATE_TARGET parameter, 194, 195
automatic memory management, 896, 897, 1148
db file sequential read wait event, 1178
setting value of, 1149–1152
PGA_TARGET parameter, 895, 1150, 1205
phantom reads problem, 341, 344, 345, 346
physical database backups, 725, 728
physical database design, 34–37
physical database structures, 173–176
control files, 173, 174–175
datafiles, 173–174
redo log files, 173, 175–176
physical reads, 1144
physical standby databases, 799
PID (process ID), UNIX, 74
PID column, top command, 84
ping command, Oracle connectivity, 516
pinhits, determining number of, 1137
pinned buffers, 188
pinning objects in shared pool, 1142–1143
pipe (|) command, UNIX/Linux, 52, 57
pipelining, table functions, 662, 666
plan directives see resource plan directives
plan stability feature, 1077
PLAN_LIST attribute, 1083
PLAN_RETENTION_WEEKS parameter, SMB, 1085
plans see execution plans
PLAN_TABLE table, 1090, 1091, 1095
PL/SQL, 97, 1241–1248
blocks, 1241
conditional control, 1243
creating cacheable function, 1124
cursors, 1245–1247
declaring variables, 1241
displaying output on screen, 109
ending SQL and PL/SQL commands, 103
error handling, 1242
explicit cursors, 1245
functions, 1247
implicit cursors, 1245
libraries, 464, 468
looping constructs, 1243–1244
packages, 1247
procedures, 1247
records, 1244
%ROWTYPE attribute, 1242
show errors command, 111
terminating PL/SQL block, 102
%TYPE attribute, 1242
writing executable statements, 1242
PL/SQL execution elapsed time, 1206
PL/SQL Function Result Cache, 1124–1125
PL/SQL statements
BEGIN, 1241, 1242
COMMIT, 1242
DECLARE, 1241
DELETE, 1242
DML statements, 1242
END, 1241
EXCEPTION, 1241, 1243
FETCH, 1245
FOR LOOP, 1244
IF-THEN, 1243
INSERT, 1242
LOOP/END LOOP, 1243
RAISE, 1243
SELECT, 1242
UPDATE, 1242
WHILE LOOP, 1244
PLSQL_CODE_TYPE parameter, 464
PLSQL_OPTIMIZE_LEVEL parameter, 468
PLSTRACE role, 1096
plustrace.sql script, 1096
PM (product media) schema, 1221
PMON (process monitor), 181, 183, 479, 521
PMON timer idle event, 1181
point-in-time recovery (PITR), 802, 823, 853
tablespace point-in-time recovery (TSPITR), 808, 840–841
policy-based configuration framework, Database Control, 151
policy functions, 582–585
policy groups, 586
Policy Violations page, Database Control, 151
policy_name parameter, ADD_POLICY, 594
POLICY_TYPE parameter, ADD_POLICY, 584
polymorphism, 39
pooling see connection pooling
port number, connect descriptors, 515
port parameter, easy connect naming
method, 529
portlist.ini file, 157
ports, 140, 614
POSITION clause, SQL*Loader, 632
post upgrade actions script, 439
post-installation tasks, Oracle Database 11g, 422–425
Oracle owner tasks, 424–425
system administrator tasks, 423–424
Post-Upgrade Status tool, 429, 440–441
POWER clause, REBUILD command, 916
predicated variables, SQL*Plus, 127
predicate columns, 1071
predicates, 583
Preferences page, Grid Control, 159
preferred mirror read feature, ASM, 909
prefixed indexes, 1073
preinstallation tasks, Oracle Database 11g, 400–413
Oracle owner tasks, 410–413
system administrator tasks see system administrator
PreparedStatement object, 539
INDEX

PREPARE_REPLAY procedure, 1214
Pre-Upgrade Information Tool, 428–429, 435–437
PREVIEW option, RESTORE command, RMAN, 811
PRI column, top command, 84
primary indexes, 297, 1070
primary key constraints, 300, 306
primary key method, 936
primary keys, 26, 31, 33, 35, 36
distinguished names (DNs), 536
guidelines for creating indexes, 297
principal parameter, CREATE_ACL, 616
PRINT parameter, TKPROF utility, 1102
PRINT SCRIPT command, RMAN, 750, 751
PRINTPRETTY_SQL procedure, 846
private data, 1190
private database links, 985–986
private SQL area, PGA memory, 194
private synonyms, 324, 326
PRIVATE_SGA parameter, 549
privilege parameter, CREATE_ACL, 616
PRIVILEGE variable, SQL*Plus, 119, 128
privileged connections, SQL*Plus, 99
privileged users, 599
privilege-level audit, 587
privileges, 567
CHECK_PRIVILEGE function, 617
controlling database access, 567–574
CREATE ANY TABLE, 268
CREATE SESSION, 545
CREATE TABLE, 268
CREATE VIEW, 312
creating materialized views, 317
creating users, 545
Data Pump privileges, 685
DBA views managing, 577
directory privileges, 571
function privileges, 571
granting object privileges, 367
granting privileges, 567
database security, 612
through roles, 618
to PUBLIC, 569
to roles, 575
to users, 135, 257
how Oracle processes transactions, 197
manual database creation, 475
materialized view privileges, 571
object privileges, 570–573
package privileges, 571
procedure privileges, 571
sequence privileges, 571
showing privileges in prompt, 119
SQL*Plus security and Oracle, 103
SYSASM privilege, 570
SYSDBA privilege, 475, 570
SYSOPER privilege, 570
system privileges, 567–570
table privileges, 571
UNLIMITED TABLESPACE, 268
view privileges, 571
views and stored procedures managing, 577
privileges script, 420
proactive tuning, 1042
problem findings, ADDM, 880
procedure privileges, 571
procedures
see also stored procedures
executing, SQL*Plus, 121
Java stored procedures, 1252
PL/SQL functions compared, 1247
process monitor (PMON), 181, 183, 479, 521
process number, deriving, 621
process-related parameters, 455
PROCESS_CAPTURE procedure, 1211
processes, 179
abnormal increase in process size, 1190–1191
components of Oracle process, 1190
CPU units used by processes, 1154
freeing up resources from dead processes, 183
measuring process memory usage, 1190–1191
setting number of parallel processes, 467
specifying number of writer processes, 455
system usage problems, 1188
PROCESSES parameter, 455
processes, Oracle see Oracle processes
processes, UNIX see UNIX processes
product files, 394, 397
production databases, 9, 554
product_user_profile table, 103, 104, 105, 493
disabling role using, 576
enabling role using, 577
restricting SQL*Plus usage, 618
.profile file, UNIX, 54, 55, 406
profiles see user profiles
program global area see PGA
PROGRAM_ACTION attribute, 1002, 1006
programs, Oracle Scheduler, 995, 1006–1007
PROGRAM_TYPE attribute, 1002, 1006
projection operations, 21, 1046, 1223
PROMPT command, SQL*Plus, 121
prompts, SQL*Plus, 99, 115, 118, 119
prompts, UNIX, 51
properties
ACID properties, transactions, 340
object-oriented database model, 39
showing properties of columns, SQL*Plus, 123
protocol address, connect descriptors, 515
protocols
communication protocol, 515
Lightweight Directory Access Protocol, 534
write ahead protocol, 182, 199
Provisioning Software Library, 148
proxy authentication, 602
proxy copies, RMAN, 753
prvtsch.plb script, 1003, 1006
ps command, UNIX/Linux, 74, 75, 81, 903
PS1 environment variable, 51
pseudo-columns, Flashback Versions
query rewrite phase, optimizing processing, 1044
QUERY REWRITE privilege, 317
query rewriting, 315, 465, 578
QUERY_REWRITE_ENABLED parameter, 315, 465, 1078
QUERY_REWRITE_INTEGRITY parameter, 316, 465
queue size, setting in listener.ora, 523
queues
  ALERT_QUE queue, 954
  operation queuing, 942
  system usage problems, 1188
QUEUESIZE parameter, listener, 523
QUERY parameter, Data Pump, 694, 704, 708
query optimization, 1043, 1047–1065
  see also CBO (Cost-Based Optimizer)
  adaptive search strategy, 1053
  choosing access path, 1052
  choosing join method, 1052
  choosing join order, 1053
  choosing optimization mode, 1047
  deferring publishing of statistics, 1056–1057
  effect when statistics not collected, 1063
  extended optimizer statistics, 1058–1060
  how CBO optimizes queries, 1051–1053
  multicolumn statistics, 1058
  OEM collecting optimizer statistics, 1064–1065
  providing statistics to CBO, 1053–1056
  providing statistics to optimizer, 1047–1049
  setting optimizer level, 1050–1051
  setting optimizer mode, 1049–1050
  specifying optimization type, 462
  SQL transformation, 1051
ALTER_USER procedure, 530
PUBLIC role, 613
PUBLIC user group, 569, 576, 615
PUBLIC role, 613
PUBLIC role, 613
PUBLIC role, 613
RDBMS (relational database management system), 19, 452
RDBMS compatibility level, ASM disk groups, 910
rdbms ipc message idle event, 1181
reactive performance tuning, 1042
read consistency, 199, 345, 356, 359
READ ONLY clause, ALTER TABLE, 273
READ ONLY clause, CREATE VIEW, 312
read permission, UNIX files, 59
READ WRITE clause, ALTER TABLE, 274
read/write operations, UNIX, 83
read-committed isolation level, 344, 345, 346
reader class, Oracle Secure Backup, 788
README files, Oracle Database 11g, 392
read-only mode, 273–274, 502
read-only tablespaces, 172, 229
read-uncommitted isolation level, 344
READY status, Oracle listener, 522
Real Application Clusters see RACs
Real Application Testing, 148, 1209
analyzing after-upgrade SQL workload, 1219
analyzing prechange SQL workload, 1218
capturing production SQL workload, 1217
capturing production workload, 1210–1211
creating SQL Tuning Set, 1217
Database Replay tool, 1209–1216
loading SQL Tuning Set, 1217
making system change, 1211
preprocessing workload, 1211
replaying captured workload, 1211
setting up replay clients, 1212
SQL Performance Analyzer, 1216–1220
transporting SQL Tuning Set, 1218
real dictionary tables, 1063
REBALANCE command, ASM disk groups, 916
rebalance master (RBAL) process, 185, 907
rebalancing disk groups, ASM, 916
REBUILD command
ALTER INDEX statement, 303, 305
POWER clause, 916
rebuilding indexes/tables regularly, 1089
rebuilding indexes online, 934, 935
RECNUM column specification, SQL*Loader, 636
Recommendation Options/Types, SQL Access Advisor, 322
recommendations, ADDM, 878, 880–881
ADDM reports, 888, 891, 892
recommendations, Segment Advisor, 932
Recompile Invalid Objects window, DBUA, 431
RECORD parameter, TKPROF utility, 1102
RECORD_FORMAT_INFO clause, 647
records, PL/SQL, 1244
RECOVER command, RMAN, 812, 813, 820
RECOVER command, SQL*Plus, 134
RECOVER BLOCK command, BMR, 864
RECOVER COPY command, 733, 778
RECOVER DATABASE command, 815, 816, 861
RECOVER DATAFILE command, RMAN, 819
RECOVER TABLESPACE command, RMAN, 818
recoverer (RECO), 185, 479
recovery
see also backups; flash recovery area
cache recovery, 804
cancel-based recovery, 824
change-based SCN recovery, 820, 824
checkpoint data in control files, 175
cloning databases, 833–840
closed recovery, 807
complete recovery, 807
crash and instance recovery, 804–805
Data Recovery Advisor, 211, 829–833
database failures, 801–804
ers, 866–870
Flashback Data Archive, 870–874
flashback recovery techniques, 202
Flashback techniques and, 847–861
granular recovery techniques, 840–847
incomplete recovery, 807
instance recovery, 468
log sequence-based recovery, 821
media recovery, 806–808
media recovery scenarios, 814–829
media vs. nonmedia recoveries, 808
open recovery, 807
Oracle recovery process, 804–809
parameters, 468–470
redo log files, 175
reducing vulnerability to recoveries, 809
repairing data corruption, 864–865
restore points, 861–864
restoring vs. recovering datafiles, 806
SHOW RECYCLEBIN command, 116
Simplified Recovery Through Resetlogs feature, 823
SQL*Plus, 134
time-based recovery, 820, 824
traditional recovery techniques, 848
transaction recovery, 804
trial recovery, 865–866
recovery catalog, RMAN, 744
backing up recovery catalog, 769
base recovery catalog, 772
benefits of RMAN, 743
cataloging backups, 770
connecting to, 746
connecting to RMAN, 767–768
creating recovery catalog, 768
creating recovery catalog schema, 767
dropping recovery catalog, 772
getting rid of invalid entries, 760
granting roles, 767
importing recovery catalog, 771
maintaining, 769–772
moving recovery catalog, 772
<table>
<thead>
<tr>
<th>performing PITR</th>
<th>766</th>
</tr>
</thead>
<tbody>
<tr>
<td>recovering recovery catalog</td>
<td>770</td>
</tr>
<tr>
<td>recovery catalog schema</td>
<td>743</td>
</tr>
<tr>
<td>registering database</td>
<td>768–769</td>
</tr>
<tr>
<td>resynchronizing recovery catalog</td>
<td>769</td>
</tr>
<tr>
<td>upgrading recovery catalog</td>
<td>771</td>
</tr>
<tr>
<td>working with</td>
<td>766–769</td>
</tr>
<tr>
<td>Recovery Configuration window</td>
<td>432, 487</td>
</tr>
<tr>
<td>recovery errors</td>
<td>866–870</td>
</tr>
<tr>
<td>recovery files, OFA guidelines</td>
<td>399</td>
</tr>
<tr>
<td>Recovery Manager</td>
<td>see RMAN</td>
</tr>
<tr>
<td>recovery scenarios</td>
<td>814–829</td>
</tr>
<tr>
<td>control file recovery</td>
<td>824–828</td>
</tr>
<tr>
<td>datafile recovery</td>
<td>818–820</td>
</tr>
<tr>
<td>recovering datafiles without backup</td>
<td>828–829</td>
</tr>
<tr>
<td>incomplete recovery</td>
<td>820–824</td>
</tr>
<tr>
<td>performing hot restore with RMAN</td>
<td>816</td>
</tr>
<tr>
<td>tablespace recovery</td>
<td>817–818</td>
</tr>
<tr>
<td>whole database recovery</td>
<td>814–817</td>
</tr>
<tr>
<td>RECOVERY WINDOW option, RMAN</td>
<td>762</td>
</tr>
<tr>
<td>recovery writer (RVWR) process</td>
<td>185</td>
</tr>
<tr>
<td>RECOVERY_CATALOG_OWNER role</td>
<td>773</td>
</tr>
<tr>
<td>recovery-related parameters</td>
<td>468–470</td>
</tr>
<tr>
<td>recursive CPU usage</td>
<td>1157</td>
</tr>
<tr>
<td>recursive relationships</td>
<td>28</td>
</tr>
<tr>
<td>Recycle Bin, 850–851</td>
<td></td>
</tr>
<tr>
<td>DBA_RECYCLEBIN view</td>
<td>850</td>
</tr>
<tr>
<td>DROP TABLE PURGE command</td>
<td>116</td>
</tr>
<tr>
<td>Flashback Drop feature</td>
<td>376, 849, 850–851</td>
</tr>
<tr>
<td>free space</td>
<td>244</td>
</tr>
<tr>
<td>Oracle removing items from</td>
<td>850</td>
</tr>
<tr>
<td>permanently removing objects from</td>
<td>853</td>
</tr>
<tr>
<td>PURGE RECYCLEBIN command</td>
<td>853</td>
</tr>
<tr>
<td>recovering dropped table/user objects</td>
<td>548</td>
</tr>
<tr>
<td>removing items from</td>
<td>852</td>
</tr>
<tr>
<td>removing tables without using</td>
<td>852</td>
</tr>
<tr>
<td>removing tablespace without using</td>
<td>853</td>
</tr>
<tr>
<td>security</td>
<td>852</td>
</tr>
<tr>
<td>SHOW RECYCLEBIN command</td>
<td>116, 850</td>
</tr>
<tr>
<td>USER_RECYCLEBIN view</td>
<td>850</td>
</tr>
<tr>
<td>recycle buffer pool</td>
<td>189, 1146</td>
</tr>
<tr>
<td>recycle pool</td>
<td>457, 458</td>
</tr>
<tr>
<td>RECYCLEBIN parameter, Flashback Drop</td>
<td>849</td>
</tr>
<tr>
<td>redirection, UNIX</td>
<td>56–57</td>
</tr>
<tr>
<td>redo data</td>
<td>227, 460</td>
</tr>
<tr>
<td>redo entries, SQL*Loader</td>
<td>640</td>
</tr>
<tr>
<td>redo log buffer</td>
<td>176, 187, 192–193</td>
</tr>
<tr>
<td>committing transactions</td>
<td>182, 198</td>
</tr>
<tr>
<td>how Oracle processes transactions</td>
<td>197</td>
</tr>
<tr>
<td>transferring contents to disk</td>
<td>182</td>
</tr>
<tr>
<td>redo log files</td>
<td>173, 175–176</td>
</tr>
<tr>
<td>see also online redo log files</td>
<td></td>
</tr>
<tr>
<td>ADD LOGFILE GROUP syntax</td>
<td>983</td>
</tr>
<tr>
<td>after-image records</td>
<td>176</td>
</tr>
<tr>
<td>applying redo logs during recovery</td>
<td>867</td>
</tr>
<tr>
<td>arch directory containing</td>
<td>397</td>
</tr>
<tr>
<td>archivelog mode</td>
<td>176, 726, 775</td>
</tr>
<tr>
<td>archiver process</td>
<td>1187</td>
</tr>
<tr>
<td>archiving</td>
<td>135, 176, 184, 459</td>
</tr>
<tr>
<td>backup and recovery architecture</td>
<td>201</td>
</tr>
<tr>
<td>backup guidelines</td>
<td>729</td>
</tr>
<tr>
<td>benefits of temporary tables</td>
<td>277</td>
</tr>
<tr>
<td>committing transactions</td>
<td>198, 338, 339</td>
</tr>
<tr>
<td>creating database</td>
<td>481</td>
</tr>
<tr>
<td>data consistency</td>
<td>176</td>
</tr>
<tr>
<td>database creation log</td>
<td>484</td>
</tr>
<tr>
<td>database files</td>
<td>398</td>
</tr>
<tr>
<td>defining filename format for archived</td>
<td>460</td>
</tr>
<tr>
<td>flash recovery area</td>
<td>738</td>
</tr>
<tr>
<td>increasing size of</td>
<td>445</td>
</tr>
<tr>
<td>instance performance</td>
<td>1205</td>
</tr>
<tr>
<td>LogMiner utility analyzing</td>
<td>845–847</td>
</tr>
<tr>
<td>making whole closed backups</td>
<td>791</td>
</tr>
<tr>
<td>monitoring</td>
<td>984</td>
</tr>
<tr>
<td>multiplexing</td>
<td>176, 182, 184, 444, 455, 729</td>
</tr>
<tr>
<td>names and locations of</td>
<td>174</td>
</tr>
<tr>
<td>naming conventions</td>
<td>398</td>
</tr>
<tr>
<td>noarchivelog mode</td>
<td>176, 726</td>
</tr>
<tr>
<td>optimizing size over time</td>
<td>445</td>
</tr>
<tr>
<td>Oracle Managed Files (OMF)</td>
<td>250, 252, 455, 924</td>
</tr>
<tr>
<td>Oracle recommendations</td>
<td>445</td>
</tr>
<tr>
<td>organization of</td>
<td>982</td>
</tr>
<tr>
<td>Pre-Upgrade Information Tool</td>
<td>428</td>
</tr>
<tr>
<td>renaming redo logs</td>
<td>983</td>
</tr>
<tr>
<td>RESETLOGS option</td>
<td>822</td>
</tr>
<tr>
<td>rolling back transactions</td>
<td>198</td>
</tr>
<tr>
<td>sizing for database creation</td>
<td>444</td>
</tr>
<tr>
<td>whole open backups</td>
<td>792</td>
</tr>
<tr>
<td>write ahead protocol</td>
<td>199</td>
</tr>
<tr>
<td>redo log groups</td>
<td>176, 184, 982, 983, 984</td>
</tr>
<tr>
<td>redo log space requests</td>
<td>1181</td>
</tr>
<tr>
<td>redo logs see redo log files</td>
<td></td>
</tr>
<tr>
<td>redo records</td>
<td>176, 182, 339</td>
</tr>
<tr>
<td>redundancy</td>
<td>729, 901, 914</td>
</tr>
<tr>
<td>REDUNDANCY attribute, ASM</td>
<td>910, 915</td>
</tr>
<tr>
<td>redundancy levels, ASM</td>
<td>909</td>
</tr>
<tr>
<td>REDUNDANCY option, RMAN</td>
<td>762</td>
</tr>
<tr>
<td>redundancy set</td>
<td>730–731, 809</td>
</tr>
<tr>
<td>REENABLE clause, SQL*Loader</td>
<td>642</td>
</tr>
<tr>
<td>REF CURSOR type, PL/SQL</td>
<td>664, 666, 1247</td>
</tr>
<tr>
<td>reference partitioning</td>
<td>284–286</td>
</tr>
<tr>
<td>REFERENCES clause, CREATE TABLE</td>
<td>285</td>
</tr>
<tr>
<td>referential integrity</td>
<td>36</td>
</tr>
<tr>
<td>referential integrity constraints</td>
<td>225, 308, 716</td>
</tr>
<tr>
<td>Reflections X-Client</td>
<td>46</td>
</tr>
<tr>
<td>refresh options, materialized views</td>
<td>316, 319</td>
</tr>
<tr>
<td>REGEXP_XYZ functions, SQL</td>
<td>1238</td>
</tr>
<tr>
<td>REGISTER DATABASE command, RMAN</td>
<td>769</td>
</tr>
<tr>
<td>registerDriver method, JDBC drivers</td>
<td>538</td>
</tr>
<tr>
<td>registration, dynamic service</td>
<td>183</td>
</tr>
<tr>
<td>regular expressions, SQL</td>
<td>1237–1239</td>
</tr>
<tr>
<td>regular expressions, UNIX</td>
<td>65</td>
</tr>
<tr>
<td>Related Links section, Database Control</td>
<td>150</td>
</tr>
<tr>
<td>relational algebra</td>
<td>21–22</td>
</tr>
</tbody>
</table>
Relational database life cycle, 23–37
Logical database design, 24–34
Physical database design, 34–37
Requirements gathering, 23–24
Relational database management system (RDBMS), 19, 452
Relational database model, 20–22, 38
Object-relational database model, 39–40
Transforming ER diagrams into relational tables, 35
Relational databases, 19–37
Relational tables, 265
Relationships, ER modeling, 25
Building entity-relationship diagram, 28
Cardinality of (1:1, 1:M, M:M), 26
Recursive relationships, 28
Relative distinguished names, 536
Relative path, UNIX, 47, 62
Release Notes and Addendums, Oracle Database 11g, 392
Releases
Ensuring database compatibility, 452
Upgrade paths to Oracle Database 11g, 426
Variable naming release number, 128
Releatly backup software, Oracle Secure Backup, 785
Relies ON clause, result cache, 1124
Reload command, lsnrctl utility, 523
Reloads, determining number of, 1137
RelY constraints, 310
Remap connection procedure, 1214
Remap data parameter, Data Pump, 693, 710
Remap datafile parameter, Data Pump, 709
Remap schema parameter, Data Pump, 679, 709
Remap table parameter, Data Pump, 709
Remap tablespace parameter, Data Pump, 710
Remap XYZ parameters, Data Pump, 679
Remark command, SQL*Plus, 128, 132
Remote access to UNIX server, 78
Remote client authentication, 615
Remote external jobs, 1002
Remote login (rlogin), 78, 79
Remote servers, copying files to/from, 254
Remote_login_passwordfile parameter, 472, 599
Remote_os_authent parameter, 612, 615
Removing files, UNIX, 59
Rename command, ALTER TABLE, 272
Rename partition command, 291
Repair XYZ commands, Data Recovery Advisor, 832, 833
Repeat interval, jobs, Oracle Scheduler, 999–1000
Repeatable-read isolation level, 344
Repeating groups, 1NF, 30
Repeating-baseline templates, AWR, 966
Repeat_interval attribute, 999, 1014
Repfooter/Repheader commands, SQL*Plus, 123
Replace clause, SQL*Loader, 629
Replace function, SQL, 1228
Replace option, SQL*Plus, 116, 120, 124
Replace script command, RMAN, 752
Replace_user_sql_profiles parameter, 1117
Replay driver, Database Replay, 1212–1216
Replay_report function, 1215
Report schema command, RMAN, 759, 769
Report XYZ commands, RMAN, 759
Report_analysis_task function, 1219
Report_auto_tuning_task function, 1119
Report_diagnostic_task function, 1037
Reporting commands, RMAN, 758–760
Reports, AWR, 966–971
Report_tuning_task procedure, 1114
Repository
Management Repository, Grid Control, 154
RMAN, 743, 744
Requirements gathering, 23–24
Res column, top command, 84
Resetlogs option, 821, 822, 823
Flashback database limitations, 861
User-managed control file recovery, 827
Resident connection pooling, DRCP, 180
Resize clause, tablespaces, 223, 238
Resize clause, tablespaces, 223, 238
Resize command, 219, 364
Resolve privilege, 616
Resource allocation, 554, 555
Resource consumer groups
Database Resource Manager, 554, 555, 556, 557–559
Assigning users to, 562–565
Automatic assignment to session, 564
Automatic group switching method, 555, 560
Create_consumer_group procedure, 557
default Oracle database groups, 558
default Oracle resource plans, 558
Enforcing per-session CPU and I/O limits, 564–565
groups granted to users or roles, 566
OEM administering, 567
Sessions currently assigned to, 566
Verifying user membership, 563
Oracle Scheduler, 1011, 1016–1017
Set_consumer_group_mapping procedure, 564
Set_consumer_mapping_pri procedure, 564
Set_initial_consumer_group procedure, 563
V$Rsrc_consumer_group view, 566
Resource management
Database Resource Manager, 554–567
dynamic resource management, 941–943
Resource Manager, Oracle, 147, 996
Resource Monitors page, OEM, 566
resource parameters, 549, 553
resource plan directives, Database Resource Manager, 555, 556, 560–562, 942
Resource Plan Wizard, OEM, 556
resource plans
  Database Resource Manager, 554, 555, 556, 559–560
  CREATE_PLAN procedure, 560
  CREATE_PLAN_DIRECTIVE procedure, 561, 565
determining status of, 562
OEM administering, 567
showing plans, 566
resource allocation for automated tasks, 1022
Scheduler, 997, 1013–1017
RESOURCE role, 574
RESOURCE_CONSUMER_GROUP attribute, 1012
RESOURCE_LIMIT parameter, 552
RESOURCE_MANAGER_PLAN parameter, 565
RESOURCE_PLAN attribute, 1014
resources
  ALTER RESOURCE COST statement, 549
  controlling user’s use of, 548, 549
  Database Resource Manager, 208
  DBA resources, 13–14
freeing up resources from dead processes, 183
Grid Control, 159
hard parsing, 1138
inefficient SQL using most resources, 1109–1110
query optimization, 1043
severe contention for, 1188
SQL Trace tool showing usage of, 1100
system usage problems, 1188
response action, alerts, 954
response directory, 421
response files, 421–422
RESTORE command, RMAN, 811, 812, 820
RESTORE CONTROLFILE command, RMAN, 825
RESTORE DATABASE command, RMAN, 814, 815
RESTORE DATAFILE command, RMAN, 819
restore optimization, RMAN, 810
RESTORE POINT clause, RMAN, 780
restore points, 861–864
RESTORE TABLESPACE command, RMAN, 817
restore utilities, UNIX, 76–77
RESTORE_DEFAULTS procedure, 533
restoring database
  performing hot restore with RMAN, 816
  restoring vs. recovering datafiles, 806
  restoring pre-upgrade database, DBUA, 433
RESTRICT command, SQL*Plus, 105
restrict option (-R), SQL*Plus, 115
RESTRICT option, STARTUP command, 908
restricted database mode, 501
  quiescing database, 505
RESTRICTED SESSION privilege, 570
result cache, 192, 1120–1126
Client Query Result Cache, 1125–1126
Function Result Cache, 1124–1125
managing, 1120, 1123–1124
RESULT_CACHE_MODE parameter, 1121–1122
result cache background (RCBG) process, 186
Result Cache Memory, 1120
result code, finding, 69
RESULT_CACHE hint, 1121, 1122, 1125
RESULT_CACHE_MAX_RESULT parameter, 464, 1121
RESULT_CACHE_MAX_SIZE parameter, 464, 1120
RESULT_CACHE_MODE parameter, 192, 464, 1120, 1121–1122, 1125
RESULT_CACHE_REMOTE_EXPIRATION parameter, 1121
Results for Task page, SQL Access Advisor, 323
ResultSet object, JDBC, 539
RESUMABLE parameter, SQL*Loader, 635
Resumable Space Allocation, 382–386
  enabling/disabling, 470
  expanding tablespaces, 224
  user-quota-exceeded errors, 226
RESUMABLE_NAME parameter, SQL*Loader, 636
RESUMABLE_TIMEOUT parameter, 384, 470, 636
RESYNC CATALOG command, RMAN, 769, 770
RETENTION GUARANTEE clause, 363, 367, 460
Flashback Query, 374
RETENTION NOGUARANTEE clause, 363
RETENTION parameter, 964, 965
retention period, Oracle Secure Backup, 789
RETENTION POLICY parameter, RMAN, 762
return codes, SQL*Loader utility, 639
REUSE parameter, 828
REUSE_DATAFILES parameter, Data Pump, 707
REUSE_DUMPFILE parameter, Data Pump, 691
reverse engineering, databases, 38
reverse key indexes, 301, 1072
Review page, SQL Access Advisor, 323
REVOKE CATALOG command, RMAN, 774
REVOKE CREATE SESSION statement, 548
REVOKE statement, 568, 569
revoking object privileges, 573
rewrite integrity, materialized views, 316
REWRITE_OR_ERROR hint, 315
Risk Matrix, 618
rlogin command, UNIX/Linux, 78, 79
rm command, UNIX/Linux, 59, 62
### RMAN (Recovery Manager)
- **Advantages of**, 810
- **Architecture**, 743–744
- **Benefits of**, 742–743
- **Block media recovery (BMR)**, 808, 864
- **Catalog reports**, 759
- **Change-tracking**, 185
- **Channels**, 753, 764
- **Cloning databases**, 834–838
- **Connecting to recovery catalog**, 746
- **Connecting to RMAN**, 745–746, 767–768
- **Control file recovery**, 825–826
- **Converting datafiles to match endian format**, 721
- **Data block corruption**, 167
- **Datafile recovery**, 819
- **Ending RMAN session**, 745
- **Identifying necessary files for recovery**, 812
- **Image copies**, 752–753
- **Incomplete recovery**, 820–823
- **Large pool**, 193
- **Making RMAN output visible**, 747
- **Media Management Layer (MML)**, 744, 745
- **Migrating databases to ASM**, 919–920
- **Monitoring/verifying RMAN jobs**, 782–784, 813
- **Oracle media recovery process**, 807
- **Performing hot restore with**, 816
- **Proxy copies**, 753
- **Recovering datafiles without backup**, 828–829
- **Recovery catalog**, 744, 766–772
- **Recovery with RMAN**, 809–814
- **Redirecting output to log file**, 747
- **Registering database**, 768–769
- **Restore optimization**, 810
- **Scripting with RMAN**, 747–752
- **Storing metadata**, 744
- **Tablespace point-in-time recovery (TSPITR)**, 840–841
- **Tablespace recovery**, 817–818
- **Terminology**, 752–753
- **User-managed recovery procedures**, 813–814
- **Virtual private catalogs**, 772–774
- **Whole database recovery**, 814–816
- **Backup sets**, 752
- **Backup tags**, 753
- **Block-change tracking**, 779
- **Compressed backups**, 780
- **Compressing**, 780
- **Checking if backup possible**, 811
- **Control file**, 765–766, 776, 784–785
- **Cross-checking backups made with RMAN**, 783
- **Cumulative backup**, 756, 757
- **Datafiles**, 777
- **Detecting physical/logical corruption**, 783
- **Differential backup**, 756, 757
- **Encrypting RMAN backups**, 781
- **Examples of**, 774–777
- **Fast incremental backups**, 779
- **File locations**, 755
- **Files**, 178
- **Image copy backups**, 756
- **Incremental backups**, 756–757, 778, 812
- **Limiting maximum size of**, 781
- **Logical check of backup files**, 756
- **Making copies of**, 754
- **Monitoring and verifying RMAN jobs**, 782–784
- **Open backups**, 792
- **Optimization parameters**, 765
- **Oracle Secure Backup intermediary**, 789
- **Performing backup and recovery tasks**, 813
- **Physical database backups**, 725
- **Restarting RMAN backup**, 777
- **Specifying limits for backup duration**, 777–778
- **Tablespaces**, 777
- **User-managed backups as alternative**, 790
- **RMAN client**, 743
- **RMAN commands**, 755–761
- **ALLOCATE CHANNEL**, 757
- **BACKUP**, 769, 755–757
- **BACKUP ARCHIVELOG ALL**, 775
- **BACKUP AS COMPRESSED**, 780
- **BACKUP AS COPY**, 756, 757
- **BACKUP CONTROLFILE**, 785
- **BACKUP CURRENT CONTROLFILE**, 776
- **BACKUP DATABASE**, 774, 775
- **BACKUP DATAFILE**, 777
- **BACKUP INCREMENTAL LEVEL**, 756–757, 778
- **BACKUP TABLESPACE USERS**, 777
- **BACKUP VALIDATE**, 783, 784
- **BLOCKRECOVER**, 864
- **CATALOG**, 759, 760, 770
- **CHECKSYNTAX parameter**, 749
- **CONFIGURE**, 753, 761–766
- **CONNECT CATALOG**, 767
- **COPY**, 758, 769
- **CREATE CATALOG**, 768
INDEX

CREATE GLOBAL SCRIPT, 750
CREATE SCRIPT, 748, 750
CROSSCHECK, 758, 761, 783
DELETE, 758, 759
DELETE SCRIPT, 750
DROP CATALOG, 768, 772
DUPLICATE, 810, 834–837
EXECUTE SCRIPT, 749
exit, 745
FLASHBACK DATABASE, 854
IMPORT CATALOG, 771, 772
job commands, 757–758
LIST, 760–761, 810
LIST INCARNATION, 823
LIST SCRIPT NAMES, 750, 761
operating system file executing, 749
PRINT SCRIPT, 750, 751
RECOVER COPY, 778
RECOVER DATABASE, 815
REGISTER DATABASE, 769
REPLACE SCRIPT, 752
REPORT, 758–760
REPORT SCHEMA, 759, 769
RESTORE, 811
RESTORE DATABASE, 814, 815
RESYNC CATALOG, 769, 770
RUN, 749, 757
SET NEWNAME, 815
SHOW ALL, 761
SWITCH, 757
SWITCH DATABASE, 816
UPGRADE CATALOG, 771
USING clause, 747, 750, 751
VALIDATE, 761, 783–784
VALIDATE BACKUP, 810–811
VALIDATE BACKUPSET, 761, 811
RMAN configuration parameters, 761–766
archivelog deletion policy, 766
backup optimization, 765
backup retention policy, 762–763
channel configuration, 764
compression, 764
CONFIGURE command, 762
control file backups, 765–766
default device type, 763
default values, 761
viewing parameters changed from, 762
degree of parallelism, 765
encryption, 763, 764
optimization parameters, 765
RECOVERY POLICY configuration, 762
RMAN executable, 743
RMAN repository, 743, 744
rmdir command, UNIX/Linux, 62
ROLE_ROLE_PRIVS view, 577
roles, 574–577
as alternative to granting privileges, 612
CONNECT role, 574
Create Role Properties page, 150
creating, 575
creating Database Control roles, 150
Data Pump Export/Import utility, 703
DBA role, 574
DBA views managing, 577
default role, 574
DELETE_CATALOG_ROLE, 569
disabling, 576, 618
dropping, 577
enabling, 577
EXECUTE_CATALOG_ROLE, 569
EXP_FULL_DATABASE role, 574
granting privileges through roles, 618
granting privileges to roles, 575
granting role to another user, 576
granting role WITH ADMIN OPTION, 576
granting roles, recovery catalog, 767
IMP_FULL_DATABASE role, 574
IMPORT_FULL_DATABASE role, 705
predefined roles, 574
PUBLIC role, 613
PUBLIC user group and roles, 576
RESOURCE role, 574
role authorization, 575–576
secure application roles, 618
SELECT_CATALOG_ROLE, 569
ROLE_SYS_PRIVS view, 577
ROLE_TAB_PRIVS view, 577
ROLLBACK command, 198, 200
exiting SQL*Plus session, 102
rollback method, conn, JDBC, 540
rollback phase, automatic checkpoint
tuning, 932
rollback segments
database creation log, 484
SET TRANSACTION USER ... statement, 365
undo management, 921
using CTAS with large tables, 132
using traditional rollback segments, 356
roll-forward phase, automatic checkpoint
tuning, 932
rolling back transactions, 196, 197, 198, 804, 806
rolling forward see cache recovery
ROLLUP operator, GROUP BY clause, SQL, 1235
root directory, UNIX, 47, 57, 63, 399
root user, Oracle Database 11g, 414
root.sh script
backing up, post-installation, 424
installing Grid Control, 154, 155
installing Oracle software, 420, 422
ROUND function, SQL, 1229
ROUND_ROBIN method, resource consumer
groups, 557
row data section, data blocks, 167
row exclusive locks, 349
%ROWCOUNT attribute, PL/SQL, 1246
ROWID, B-tree index, 298
ROWID method, online data redefinition, 936
INDEX

ROWID table access, query optimization, 1052
row-level access, VPD, 578
row-level Flashback techniques, 848
row-level locks, 347, 348, 349
row-level security, 583, 584, 585
rows
identifying changed row versions, 375
INSERT/DELETE/UPDATE statements, 263
row expiry time/SCN, 371
storing table rows, 267
ROWS parameter, SQL*Loader, 634, 641
%ROWTYPE attribute, PL/SQL, 1242, 1244
RPAD function, SQL, 1228
rule-based optimization, 1047
RULES keyword, 668–670
RUN command, RMAN, 749, 757
RUN command, SQL*Plus, 125, 126
run queue length, 1153
RUN_CHAIN procedure, 1010
RUN_CHECK procedure, 1033
runInstaller script, 415, 416, 418
RUN_JOB procedure, 1000
runnable process, UNIX, 80
runtime area, PGA memory, 194
runtime option, installing Oracle Client, 518
RUN_TO_COMPLETION method, 557
rwx group, file permissions, UNIX, 60, 61

S
-S option (silent), SQL*Plus, 115
salt, encryption algorithms, 608
SAM tool, 408
SAME (stripe-and-mirror-everything), 399, 1160
sample data, performance statistics, 948
SAMPLE parameter, Data Pump, 694
sampling data dynamically, 463
SANs (Storage Area Networks), 93
sar command, UNIX, 83, 1153, 1159, 1177
SAVE command, SQL*Plus, 106, 124
saved metrics, 952
SAVEPOINT statement, 263, 339
savepoints, rolling back transactions, 198
scalability, 1141–1142
OEM, 139
Oracle Net Services features, 512
Schedule page, SQL Access Advisor, 323
Scheduler, 208, 994–1019
architecture, 997
attribute management, 1017
automated maintenance tasks, 1020
automatic optimizer statistics collection, 898–899
calendar expression, 999
chain jobs, 996
chains, 995, 1008–1010
components, 994–995, 996–997
managing, 998–1000, 1011–1017
CREATE JOB privilege, 997
database jobs, 995
database management, 147, 211
DBMS_JOB package compared, 996
default operation windows, 899
detached jobs, 996
events, 995, 1010–1011
EXECUTE privileges, 997
external jobs, 996
gathering statistics, 1047
GATHER_STATS_JOB, 898–899
job classes, 996, 1011–1013
job table, 997
jobs, 994
administering, 1000
creating, 998–999
default scheduler jobs, 1019
managing, 998–1000
managing external jobs, 1002–1006
monitoring scheduler jobs, 1017–1019
specifying repeat interval, 999–1000
lightweight jobs, 996, 1000–1002
linking to Resource Manager, 996
MANAGE SCHEDULER privilege, 997, 1013, 1015
managing Scheduler attributes, 1017
managing/monitoring database, 213
object naming, 997
privileges, 997–998
programs, 995, 1006–1007
purging job logs, 1019
resource consumer groups, 1011
SCHEDULER_ADMIN role, 997
schedules, 995, 1007–1008
slaves, 997
types of Scheduler jobs, 995–996
window groups, 997, 1017
windows, 996, 1013–1017
Scheduler Agent
creating/enabling remote external jobs, 1005–1006
installing/configuring, 1004–1005
managing external jobs, 1002
setting registration password for, 1003
Scheduler wait class, 1163
SCHEDULER_ADMIN role, 997
schedules, Oracle Scheduler, 995, 1007–1008
creating event-based schedules, 1011
windows compared, 1013
scheduling
large jobs, Oracle, 554
performing backups with Oracle Secure
Backup, 789
serializable schedules, 342
specifying limits for backup duration, 777
UNIX jobs, 77
schema mode, Data Pump Export, 688, 705
schema owner, 264
granting object privileges, 572
INDEX

1321

Schema page, Database Control, 147
schema-independent users, 603
schemas
database schemas, 20
logical database structures, 165
Oracle Database 11g sample schemas, 1221–1222
SET SCHEMA statement, 327
shared schemas, LDAP, 603
SCHEMAS parameter, Data Pump, 688, 705, 708
schemas, Oracle, 264–265, 327
SCN (system change number), 199
committing transactions, 198
consistent database backups, 727
control files, 174
converting between time stamps and
SCNs, 847
ENABLE_AT_SYSTEM_CHANGE_NUMBER
procedure, 369
Flashback Database, 859, 860
Flashback Versions Query, 372
incomplete recovery using RMAN, 820, 822
TO SCN clause, FLASHBACK TABLE, 378
undo records, 356
user-managed incomplete recovery, 824
VERSIONS_STARTSCN pseudo-column, 371
SCNHINT parameter, 380, 869
SCN_TO_TIMESTAMP function, 847
scope, SPFILE dynamic parameter changes, 496
SCOPE clause, ALTER SYSTEM, 496
scp command, UNIX/Linux, 79
scripting with RMAN, 747–752
scripts
See also SQL scripts
coraenv script, 424
dbshtut.sh script, 423
dbstart.sh script, 423
oraenv script, 424
post upgrade actions script, 439
upgrading shutdown/start up scripts, 423
upgrade actions script, 438
upgrade script, 438
SDK Instant Client packages, 520
se (storage element), Oracle Secure Backup, 789
SEC_CASE_SENSITIVE_LOGON parameter,
465, 491, 597, 615
SEC_MAX_FAILED_LOGIN_ATTEMPTS
parameter, 465, 615
Second Normal Form (2NF), 31–33
secondary indexes, 297, 1070, 1071
second-generation database management
systems see RDBMS
sec_protocal_error_xyz_action parameters, 615
SEC_RELEVANT_COLS parameter, 585
SECTION SIZE parameter, RMAN, 781, 784
secure application roles, 618
secure configuration, automatic, 611
SecureFiles, 472
security
See also authentication; passwords
access control lists, 616, 617
Advanced Security option, 618
alerts, 617
altering profiles, 619
application security, 618
aspects of database security, 543
auditing database usage, 586–596
authenticating users, 596–602
automatic secure configuration, 611
centralized user authorization, 602
checking for latest security patches, 617
column-level security, 312
controlling database access, 567–586
Critical Patch Updates, 617
data encryption, 603–608
database auditing, 612
database management, 147
DBA role, 3, 4
dBA views managing users/roles/
privileges, 577
default auditing, 611
determining user’s currently executing
SQL, 619
directory naming method, 534
disabling roles, 618
enhancing database security, 611–622
terprise user security, 603–611
effine-grained data access, 578–586
granting privileges, 612
granting privileges through roles, 618
implementing physical database design, 37
killing user’s session, 620
listing user information, 619
logging in as different user, 620
managing users, 619
MetaLink, 617
multiple DBAs, 613
OEM, 138
OID (Oracle Internet Directory), 611
operating system authentication, 612
Oracle Advanced Security, 535
Oracle Backup Web Interface tool, 788
password-related parameters, user profiles,
549, 550
password-specific security settings, 611
Peter Finnegan’s Oracle security web site, 614
privileges, 567–574
protecting data dictionary, 613
PURGE option, DROP TABLE command, 852
RAID systems, 88
recovery catalog, 770
restricting SQL*Plus usage, 618
Risk Matrix, 618
roles, 574–577
schemas, 165
securing listener, 614
securing network, 614
setting security-related initialization parameters, 615
setting UNIX permissions, 613
stored procedures, 577
tablespace encryption, 608–610
underlying objective, 543
user accounts, 611
value-based security, 312
views, 312, 577
security policies, 583–585
Security Settings window, DBCA, 488
security, SQL*Plus see SQL*Plus security
security-related parameters, 472
Segment Advisor, 212
advice levels, 930
automatic performance tuning features, 1132
Automatic Segment Advisor, 212, 931
choosing candidate objects for shrinking, 930
Database Control Segment Advisor page, 931
description, 977
managing/monitoring database, 214
modes, 930
segment shrinking using, 930, 935
using, 980
Segment Advisor Recommendations page, 932, 980
SEGMENT ATTRIBUTES option, Data Pump, 711
segment-level statistics, 1173
segment shrinking, 927–930
segment space management, 217–218, 219, 221
segments, 166, 169
analyzing segment growth, 268
DBA_SEGMENTS view, 244
extent allocation/deallocation, 220
segment growth, 219
tablespace space alerts, 226
tablespaces, 166, 170
types of, 220
unable to extend segment error, 385
Segments by Physical Reads section, AWR, 971
sel clause, COPY command, SQL*Plus, 133
Select a Product to Install window, Grid Control, 155
SELECT ANY DICTIONARY privilege, 569
Select Configuration Options window, 419
Select Installation Type window, 417
SELECT statement
AS OF clause, 367–368
CTAS command, 273
DML statements, 263
locking, 349
VERSIONS BETWEEN clause, 370
SELECT statement, PL/SQL, 1242
SELECT statement, SQL, 1223–1224, 1226
SELECT_CATALOG_ROLE, 569, 570
SELECT_CURSOR_CACHE procedure, 1217
Selecting Database Instance window, DBUA, 431
selection operations, 21, 1046
selectivity, 1065
self join, SQL, 1233
self-managing database, 877
semantic data models, 25
Semantic Web, The, 25
semaphores, 404
semi-structured data models, 40
SEMMNS kernel parameter, 413
SEQUEL (structured English query language), 22
SEQUENCE function, SQL*Loader, 637
SEQUENCE option, SET UNTIL command, 821
sequence privileges, 571
sequences, 327–328, 329
SQL*Loader utility, 643
SEQUENTIAL_ORDER, rules, 670
serializable isolation level, 344, 346
serializable schedules, data concurrency, 342
server configurations, 180
server error triggers, 591
server file, 493
Server Generated Alerts, 386
Server Manageability Packs, 459
Server page, Database Control, 146–147
server parameter file see SPFILE
server processes, 179–180, 197
see also Oracle processes
server processes, RMAN, 744
server software, installing, 416–420
server-executed commands, SQL, 103
server-generated alerts, 211, 952–953
SERVEROUTPUT variable, SQL*Plus, 108, 109–110
servers, 45, 47
server-side access controls, securing network, 614
service level agreements see SLAs
service metrics, 951
Service Name Configuration page, 528
service names, database, 514
service registration, dynamic, 183
SERVICE_NAME parameter, 452, 529
SERVICE_NAMES parameter, 514, 521
services command, lsnrctl utility, 522
Services Summary, checking listener status, 522
SERV_MOD_ACT_TRACE_ENABLE procedure, 1106
session metrics, 951
session multiplexing, 180
session-related parameters, 456
session sample data, 948
session switching, 561
session variables, SQL*Plus, 126
SESSION_CACHED_CURSORS parameter, 1140, 1141
INDEX

session-control statements, SQL, 262
SESSION_PRIVS/ROLES views, 577
sessions
  Active Session History (ASH), 210, 971–975
  ACTIVE_SESSION_POOL parameter, 560
  ACTIVE_SESS_POOL_MTH parameter, 560
  altering properties of user sessions, 262
  audit by session, 587
  creating plan directives, 560
  creating session temporary table, 278
  determining session-level CPU usage, 1155
  killing user sessions, 620
  LICENSE_MAX_SESSIONS parameter, 461
  QUEUEING_MTH parameter, 560
  specifying maximum number of, 461
  sys_context discovering session
    information, 579
  terminating database sessions, 75
  Top Sessions page, 1200
  UNIX session, 47
  using Database Control to manage session
    locks, 354–355
  V$ACTIVE_SESSION_HISTORY view, 1169
  viewing Data Pump sessions, 714
SESESSIONS parameter, 455
sessions, SQL*Plus see SQL*Plus sessions
SESSIONS_PER_USER parameter, 549
SESSION_TRACE_ENABLE procedure, 1175
SESSION_USER attribute, USERENV
  namespace, 579, 580
SET command, SQL*Plus, 106–107
  ALTER SESSION command, 262
  controlling security via SET ROLE
    command, 105
  getting list of environment variables, 107
  SET ERRORLOGGING command, 111
  SET SERVEROUTPUT command, 109–110
  SET SQLPROMPT command, 118
SET command, RMAN, 815
set operations, 21
set operators, 263, 1227
set password clause, lsnrctl utility, 524
SET ROLE command, 105, 618
SET SCHEMA statement, 327
SET TRANSACTION statement, 263
  Automatic Undo Management (AUM), 357
  USER ROLLBACK SEGMENT clause, 365
SET UNTIL command, 820, 821, 822
SET UNUSED command, 271
SET_ATTRIBUTE procedure, 1005, 1008, 1018
SET_ATTRIBUTE_NULL procedure, 1017
SET_ATTRIBUTES procedure, 1016
setAutoCommit method, conn, JDBC, 540
SET_CONSUMER_GROUP_MAPPING
  procedure, 564
SET_CONSUMER_MAPPING_PRI
  procedure, 564
SET_DEFAULT_TASK procedure, 890
SET_DEFAULT_TASK_PARAMETER procedure, 884, 890
setenv command, UNIX/Linux, 54
SET_EV procedure, 1175
SET_IDENTIFIER procedure, 1106
SET_INITIAL_CONSUMER_GROUP
  procedure, 563
set-oriented relational model, 20
SET_SCHEDULER_ATTRIBUTE procedure, 1017
SET_SESSION_LONGOPS procedure, 943
SET_SESSION_TIMEOUT procedure, 385, 386
SET_SQL_TRACE procedure, 1101
SET_SQL_TRACE_IN_SESSION procedure, 1102
SET_TABLE_PREFS function, 1057
SET_TASK_PARAMETER procedure, 324, 978
SET_THRESHOLD procedure, 955
settings, SQL*Plus, 115
SET_TUNING_TASK_PARAMETERS procedure, 1037, 1117
SET_TUNING_TASK_PARAMETERS view, 1117
SETUID files, 613
Setup page
  Database Control, 148–149
  Grid Control, 159
Setup Privileges window, Grid Control, 156
setupinfo file, connecting to Grid Control, 157
SEVIS (Student and Exchange Visitor
  Information System), 1193
SGA (system global area), 187–193
  ASM Cache component, 906
  automatic memory management, 195–196,
    894, 896
  components, 187
  database buffer cache, 187, 188–189, 190
  Java pool, 187, 193
  large pool, 187, 193
  managing/monitoring database, 214
  memory allocation, 194
  memory management, 894
  memory-configuration parameters, 456
  MEMORY_TARGET parameter, 457
  multiple database block sizes, 189–190
  PRIVATE_SGA parameter, 549
  redo log buffer, 187, 192–193
  Result Cache Memory, 1120
  shared pool, 187, 191–192
  SHOW SGA command, 117
  sizing buffer cache, 1144
  Streams pool, 187, 193
  tuning shared pool, 1133
SGA_TARGET parameter, 195, 196
  automatic memory management, 895,
    896, 897
db file scattered read wait event, 1177
managing result cache, 1120
setting size of, 1142
sh (Bourne shell), 45
  see also shells, UNIX
SH (sales history) schema, 1222
shadow process, Data Pump, 686
Shallahamer, Craig A., 1148
share lock mode, 199
shared context sensitive security policy, 584
shared database objects, 987–991
  comparing data, 987–989
  converging data, 990–991
shared libraries, 1190
shared memory, 894, 1190
shared pool, 187, 191–192
  components, 1133
    delays due to shared pool problems, 1191
  dictionary cache, 1134–1135
    flushing, 1135
  how Oracle processes transactions, 197
    LARGE_POOL_SIZE parameter freeing memory in, 459
  library cache, 191, 1133–1134
  Oracle’s guidelines, 456
  pinning objects in, 1142–1143
  reducing parse-time CPU usage, 1157
    sizing, 1142
    tuning, 1133–1135
shared pool latch, 1180
shared schemas, LDAP, 603
shared server architecture, 512
shared server configuration, 180
shared static security policy, 584
SHARED_POOL_SIZE parameter, 195
shell scripts, UNIX, 45, 68–74
  evaluating expressions, 69
  executing with arguments, 70
  flow control structures, 71–74
  making variables available to, 54
shells, UNIX, 45–46
  changing shell limits, preinstallation, 406
  changing shell prompt, 51
  customizing environment, 55
  running programs with nohup option, 75
  shell prompts, 47
  shell variables, 53, 54, 55, 69
ship.db.cpio.gz file, 415
SHMMAX kernel parameter, 413
SHOW ALL command, RMAN, 761
SHOW command, SQL*Plus, 116–118
SHOW ERRORS command, SQL*Plus, 111
SHOW HM_RUN command, 1034
SHOW INCIDENT command, 1026
SHOW PARAMETER command, 474
SHOW PARAMETER UNDO command, 364
SHOW RECYCLEBIN command, 850
SHRINK SPACE clause
  ALTER TABLE command, 929
  ALTER TABLESPACE command, 231–232
shutting down database from SQL*Plus, 502–505
SHUTDOWN command, ASM instances, 908, 910, 911
SHUTDOWN command, SQL*Plus, 502–505
SHUTDOWN IMMEDIATE command, 492, 503
SHUTDOWN NORMAL command, 502
shutting down scripts, updating post-installation, 423
SHUTDOWN TRANSACTIONAL command, 503
Simplified Recovery Through Resetlogs feature, 823
Simulated backups and restores, RMAN, 743
Single Sign-On feature, 603
smallfile tablespaces, 172, 236, 238
SMBS (SQL Management Base), 1085
SMON timer idle event, 1181
snapshot backup techniques, 730
snapshot variables, AWR, 882
smallfile tablespaces, 172, 236, 238
SORT parameter, TKPROF utility, 1102
sort command, UNIX/Linux, 68
sort segment, temporary tablespaces, 230
SORT_AREA_SIZE parameter, 194
SORTED_INDEXES parameter, SQL*Loader, 641, 642
sorting data
  by multiple columns, 1227
  guidelines for creating indexes, 297
  one-pass sort, 1149
SMB (SQL Management Base), 1085
SORTED_INDEXES parameter, SQL*Loader, 641, 642
SORT parameter, TKPROF utility, 1102
sort command, UNIX/Linux, 68
sort segment, temporary tablespaces, 230
SORT_AREA_SIZE parameter, 194
SORTED_INDEXES parameter, SQL*Loader, 641, 642
sorting data
  by multiple columns, 1227
  guidelines for creating indexes, 297
  one-pass sort, 1149
optimal sort, 1149
ORDER BY clause, 263, 1226
sort-merge join, 1052
source command, UNIX/Linux, 55
space
DBMS_SPACE package, 298–299
preinstallation checks, 401
user’s space quota errors, 383
space alerts, 226–227
space management
ADDM recommendations, 881
automatic checkpoint tuning, 932–933
automatic Segment Advisor job, 931
Automatic Segment Space Management, 928
automatic space management, 921–933
Automatic Undo Management (AUM), 921
Data Pump Export estimating, 696, 697
Data Pump technology, 679
DBA_FREE_SPACE view, 243
DBMS_SPACE package, 255–256, 268
deallocating unused space, 268, 928
estimating space requirements, 268
file management with OMF, 922–927
finding unused space in segments, 928
fragmentation, 928
free space, 255–256
managing flash recovery area, 740
manual segment shrinking, 928–929
monitoring resumable operations, 386
online segment shrinking, 927–928
preinstallation tasks, 400
reclaiming unused space, 1090
recursive CPU usage, 1158
removing objects from Recycle Bin, 850
Resumable Space Allocation, 382–386
segment shrinking using Segment
Advisor, 930
segment space management, 217–218
temporary tablespace groups, 233
Space Summary, Performance page, Database
Control, 145
SPACE_BUDGET_PERCENT parameter,
SMB, 1085
SPACE_USAGE procedure, 255, 268
specifiers of time interval, Scheduler jobs, 999
Specify File Locations window, 155
Specify Home Details window, 417
sperrorlog table, SQL*Plus, 111
SPFILE (server parameter file), 177, 446,
493–497
see also initialization files; PFILE
automatic service registration, 521
backup guidelines, 729
cloning databases with RMAN, 836, 837
DB_WRITER_PROCESSES parameter, 182
making changes permanent, 447
OMF control files, 250
post-upgrade actions, 441
reading SPFILE, 473
setting archivelog-related parameters, 491
setting initialization parameters,
post-installation, 424
spindles, disk I/O, 1159
SPLIT PARTITION command, 291
SPM (SQL Plan Management), 1080–1087
parallel execution, 1085
SQL Management Base (SMB), 1085
SQL plan baselines, 1080–1085
SPOOL command, SQL*Plus, 120, 121
restriction levels, 106
spool files, manual upgrade process, 434, 441
spool files, SQL*Plus, 120
sps column, iostat command, 82
SQL (structured query language)
see also PL/SQL
analyzing locks, 353–354
cursors, 1245–1247
determining user’s currently executing
SQL, 619
efficient SQL, 1065–1075
avoiding improper use of views, 1075
avoiding unnecessary full table scans, 1075
bind variables, 1075
bitmap join indexes (BJI), 1069
indexing strategy, 1070–1073
inline stored functions, 1074–1075
monitoring index usage, 1073
removing unnecessary indexes, 1073
selecting best join method, 1068
selecting best join order, 1070
using hints to influence execution plans,
1067–1068
using similar SQL statements, 1074
WHERE clauses, 1065–1067
ending SQL and PL/SQL commands, 103
grouping operations, 1234–1236
hierarchical retrieval of data, 1232
inefficient SQL, 1108–1110
introduction, 22
MODEL clause, 667–670
operators, 1227
optimal mode operation, 194
Oracle SQL Developer, 136
performing online data reorganization,
934–935
regular expressions, 1237–1239
selecting data from multiple tables,
1232–1239
server-executed commands, 103
single-pass operation, 194
transforming data, 658–667
using SQL to generate SQL, 135–136
wildcard characters, 1224
writing subqueries, 1236–1237
XML and SQL, 261
SQL Access Advisor, 212, 320–324, 977
automatic performance tuning features, 1132
creating materialized views, 317
evaluation mode, 321
index recommendations, 303
invoking through Database Control, 320–323
invoking through DBMS_ADVISOR, 323
managing/monitoring database, 214
QUICK_TUNE procedure invoking, 324
tuning SQL statements, 1127
SQL Access Mode, 1198
SQL buffers, 125, 128
SQL Developer, 401
SQL files, Data Pump utilities, 681
SQL functions, 1228–1232
aggregate functions, 1229, 1232
analytical functions, 1231–1232
conditional functions, 1230
date functions, 1229
general functions, 1230
histograms, 1232
hypothetical ranks and distributions, 1232
inline stored functions for efficient SQL, 1074–1075
inverse percentiles, 1231
moving-window aggregates, 1231
number functions, 1229
period-over-period comparisons, 1231
ranking functions, 1231
ratio-to-report comparisons, 1231
single-row functions, 1228
statistical functions, 1231
SQL functions, list of
AVG, 1229
CASE, 1230
COALESCE, 1230
CONCAT, 1228
COUNT, 1229
DECODE, 1230
INSTR, 1228
LENGTH, 1228
LOWER, 1228
LPAD, 1228
MAX, 1229
MIN, 1229
NVL, 1230
RANK, 1231
REPLACE, 1228
ROUND, 1229
RPAD, 1228
SUBSTR, 1228
SUM, 1229
SYSDATE, 1229
TO_CHAR, 1230
TO_DATE, 1229
TO_NUMBER, 1230
TO_TIMESTAMP, 1229
TRIM, 1228
TRUNC, 1229
SQL hash value, 343
SQL Management Base (SMB), 1085
SQL MERGE command, 269
SQL Net Services see Oracle Net Services
SQL Ordered by Elapsed Time section, AWR, 970
SQL Performance Analyzer, 213, 1216–1220
change management, 7
database management, 148
database upgrade management, 1081
SQL performance tuning tools, 1090–1105
 Autotrace utility, 1095–1099
 EXPLAIN PLAN tool, 1090–1095
 SQL Trace utility, 1099–1102
 TKPROF utility, 1102–1105
 SQL plan baselines, 1081–1085
 SQL Plan Management see SPM
SQL plans
OPTIMIZER_XYZ_BASELINES parameters, 462, 464
SQL processing, improving, 1075–1080
 materialized views, 1077
 partitioned tables, 1076
 stored outlines, 1077–1080
 table compression, 1076
SQL Profile feature, 1068
SQL profiles, 1112, 1115, 1116, 1117
SQL queries see query optimization
SQL Query Result Cache, 1124
SQL Repair Advisor, 1023, 1035–1038
SQL Response Time chart, Database Control, 1196
SQL scripts
@@commandfile notation, 129
catdwgrd.sql, 434, 442
catumgrd.sql, 434, 438, 439
catuppset.sql, 438, 439
cautution when using /, 126, 128
create directory containing, 397
eexecuting command scripts consecutively, 129
eexecuting in SQL*Plus, 124–126
utilr.sql, 434
utlu111i/utlu111s.s scripts, 429, 434
utluuppset.sql, 434
viewing script before executing, 128
SQL statements, 261–264, 1223–1226
 Automatic Tuning Optimizer (ATO), 1113
 AWR performance statistics for, 1184
 comparing performance of, 1127
 creating stored outlines for, 1079
 DDL statements, 264
deadlocks, 340
DELETE statement, 1225–1226
DML statements, 263
embedded SQL statements, 262
executing, JDBC, 539–540
identical statements, 1134
identifying inefficiency in, 1127
identifying problem statements, 1127
identifying SQL with highest waits, 1171
INSERT statement, 1225
instance performance, 1194
object privileges, 570
performance tuning, 1042
processing steps, 1133
processing through JDBC, 538–540
query processing stages, 1043–1046
SELECT statement, 1223–1224
session-control statements, 262
stages of SQL processing, 343
statement-level rollback, 340
system-control statements, 262
terminating in SQL*Plus, 101
terminating, 130
TKPROF utility information on, 1103
transition control statements, 263
tunings, 1126–1127
types of, 262
undoing data changes, 374
UPDATE statement, 1226
using similar SQL statements, 1074
wait events, 1162
SQL Test Case Builder, 211, 1023, 1038
SQL Trace utility, 1099–1102
interpreting trace files with TKPROF, 1102–1103
monitoring index usage, 304
providing parse information, 1135–1136
tracing SQL statements, 1107
tuning SQL, 1105
turning on/off, 467
SQL transformation, CBO optimizing queries, 1051
SQL tuning
configuring automatic tuning, 1117–1119
GUI-based tools, 1120
identifying inefficient SQL statements, 1127
interpreting automatic tuning reports, 1119
managing automatic tuning, 1118–1119
managing tuning categories, 1115
using dictionary views for, 1108, 1110
views managing automatic tuning, 1114
SQL Tuning Advisor, 212, 976, 1111–1115
automatic performance tuning features, 1132
Automatic SQL Tuning Advisor, 209, 212, 1115–1120
Database Control, 1200
DBMS_SQLTUNE package, 1113–1115
evolving SQL plan baselines, 1083
managing SQL profiles, 1115
managing/monitoring database, 214
OEM running, 1115
performing automatic SQL tuning, 1113–1114
superior SQL plan baselines, 1085
SQL Tuning Sets (STS), 1115, 1217, 1218
SQL*Loader control file, 628–636
APPEND clause, 629
BAD parameter, 635
BEGINDATA clause, 628, 630
bind array, 634
BINDSIZE parameter, 634
command-line parameters, 633–636
CONCATENATE clause, 630
CONTINUEIF clause, 630
CONTROL parameter, 633
DATA parameter, 634
data transformation parameters, 633
data types, 632
datafile specification, 630
delimiters, 632
DIRECT parameter, 634
DISCARD parameter, 635
DISCARDMAX parameter, 635
ENCLOSED BY clause, 632
ERRORS parameter, 634
field list, 629
fixed record format, 631
INFILE parameter, 628, 630
input file field location, 632
INSERT clause, 629
INTO TABLE clause, 629
LOAD DATA keywords, 629
LOAD parameter, 634
LOG parameter, 635
logical records, 630
OPTIONS clause, 633
PARALLEL parameter, 635
physical records, 630
POSITION clause, 632
record format specification, 631
REPLACE clause, 629
RESUMABLE parameters, 635, 636
ROWS parameter, 634
SILENT parameter, 635
SKIP parameter, 636
stream record format, 631
table column name specification, 631
TERMINATED BY clause, 632
USERID parameter, 633
variable record format, 631
SQL*Loader utility, 207, 625, 627–645
COLUMNARRAYROWS parameter, 641
CONSTRAINT parameter, 636
control file, 628–636
c conventional data loading, 639
data-loading techniques, 642–645
DATA_CACHE parameter, 641
datafiles, 628
DEFAULTIF parameter, 642
DIRECT clause, 641
direct-path loading, 639–642
disabling constraints, 308
dropping indexes before bulk data loads, 644
dropping external tables compared, 646
EXPRESSION parameter, 636
EXTERNAL_TABLE parameter, 653
generating data during data load, 636
generating external table creation statements, 653–656
invoking, 637–638
loading data from table into ASCII file, 643
loading into multiple tables, 644
loading large data fields into a table, 643
loading sequence number into a table, 643
loading username into a table, 643
loading XML data into Oracle XML database, 645
log files, 638–639
MULTITHREADING parameter, 641
NOLOGGING option, 644
NULLIF parameter, 642
optimizing use of, 642
PARFILE parameter, 637
password security, 638
RECNUM column specification, 636
redo entries, 640
REENABLE clause, 642
resumable database operations, 383
return codes, 639
ROWS parameter, 641
SEQUENCE function, 637
SKIP_INDEX_MAINTENANCE clause, 641, 642
SKIP_UNUSABLE_INDEXES clause, 641, 642
SORTED_INDEXES clause, 641, 642
steps when using, 628
STREAMSIZE parameter, 641
sysdate variable, 637
trapping error codes, 645
types of data loading using, 627
UNRECOVERABLE parameter, 640, 641, 642
user pseudo-variable, 643
viewing available parameters, 629
WHEN clause, 642
SQL*Net message from client idle event, 1181
SQL*Plus, 207
& prefixing variable names, 126
actions following commands, 123
adding comments to scripts, 132
archiving redo log files, 135
calculating statistics, 123
cautions when using / command, 126, 128
continuation characters, 102
copying tables, 132–133
creating command files, 124–129
creating reports, 122, 123–124
creating web pages using, 134
creating Windows batch script, 126
creating/deleting session variables, 126
displaying environment variable values, 116
editing within SQL*Plus, 129–134
error logging, 111–112
establishing Oracle connectivity, 517
executing contents of SQL*Plus buffer, 125
executing packages/procedures, 121
executing previous command entered, 128
executing SQL command scripts consecutively, 129
executing SQL scripts in, 124–126
formatting output, 122
Instant Client packages, 520
listing SQL commands, 128
listing table columns and specifications, 119
making DML changes permanent, 133
Oracle SQL*Plus interface, 97
predefined variables, 118, 127
preserving environment settings, 115
printing report footer text, 123
printing report header text, 123
privileged connections, 99
prompts, 99
reasons for using, 97
recovering database/files/tablespaces, 134
removing current settings, 115
restricting usage of, 618
rollback command, 102
saving output to operating system, 120
saving SQL buffer contents to file, 124
saving user input in variable, 121
sending messages to screen, 121
setting default editor’s name, 124
setting environment, 106–107
showing database instance name in prompt, 118
showing help topics, 106
showing properties of columns, 123
shutting down database from, 502–505
specifying where formatting changes occur, 122
starting database from, 497–499
substitution variables, 126
terminating SQL statements, 101, 132
turning off messages after code execution, 108
using comments in, 128
using operating system commands from, 119
using SQL to generate SQL, 135–136
viewing details about archive logs, 135
viewing output screen by screen, 121
viewing previous command, 128
viewing SQL script before executing, 128
SQL*Plus command files, 124–129
SQL*Plus command-line options, 113–115
SQL*Plus commands
administrative/management commands, 115–118
database administration commands, 134–135
disabling commands, 105
doing commands, 118–122
ending SQL*Plus commands, 103
formatting commands, 118, 122–124
list of available commands, 106
local commands, 103
restriction levels for, 106
<table>
<thead>
<tr>
<th>Running commands sequentially, 124–129</th>
<th>Execution order of glogin.sql and login.sql file, 111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show errors command, 111</td>
<td>Predefined variables, 127</td>
</tr>
<tr>
<td>Terminating SQL*Plus commands, 102</td>
<td>Preserving environment settings, 115</td>
</tr>
<tr>
<td>Types of commands, 103</td>
<td>Setting SQL*Plus environment, 106–107</td>
</tr>
<tr>
<td>Working commands, 118–122</td>
<td>SHOW ALL command, 116</td>
</tr>
</tbody>
</table>

**SQL*Plus commands, list of**

| ACCEPT, 121                              | Specification of global preferences, 110         |
| APPEND, 131                              | Specifying global preferences, 110               |
| ARCHIVE LOG, 135                         | Specifying individual preferences, 110–111       |
| BREAK, 122                               | SQLTERMINATOR, 132                               |
| BTTITLE, 123                             | SQL*Plus Instant Client, 98                       |
| CHANGE, 129                              | SQL*Plus security, 103–106                       |
| CLEAR, 115                               | SQL*Plus sessions                                 |
| COLUMN, 123                              | Connecting through Windows GUI, 101              |
| COMPUTE, 123                             | Connecting using CONNECT command, 100             |
| CONNECT, 100                             | Customizing session, 113                         |
| COPY, 132–133                            | Exiting session, 102                             |
| DEFINE, 126                              | NOLOG option, 101                                 |
| DEL, 131                                 | Overriding environment variables                |
| DESCRIBE, 119                            | Preferred session settings, 108                  |
| ed, 130                                  | Privileged connections using AS clause, 99       |
| EDIT, 106                                | Rollback command, 102                            |
| EXECUTE, 121                             | Setting Oracle environment for, 98               |
| EXIT, 102                                | Starting session from command line, 98–100       |
| GET, 106, 128                            | Starting session, 98–101                         |
| GRANT, 136                               | Starting session from command line, 97–101       |
| HELP INDEX, 106                          | SQLException method, Java error handling, 540     |
| HOST, 105, 106, 119                      | SQLFILE parameter, Data Pump, 681, 706–707       |
| INPUT, 130                               | SQL_HANDLE attribute, 1083                       |
| LIST, 128                                | SQLJ, 1253                                       |
| MARKUP, 134                              | SQLLDR command, SQL*Loader, 637–638              |
| PASSWORD, 547                            | sqlnet.ora file                                   |
| PAUSE, 121                               | backup guidelines, 729                           |
| PROMPT, 121                              | external naming method, 534                      |
| QUIT, 102                                | local naming method, connections, 526            |
| RECOVER, 134                             | OID making database connections, 535             |
| REMARK, 128, 132                         | Securing network, 614                            |
| REPFOOTER/REPHEADER, 123                 | SQLPLUS command                                   |
| RESTRICT, 105                            | Receiving FAN events, 101                         |
| RUN, 125, 126                            | sqlplus command                                   |
| SAVE, 106, 124                           | Easy connect naming method, 529                  |
| SET, 106–107                             | Starting SQL*Plus session from command line, 98  |
| SET ERRORLOGGING, 111                    | sqlplus.sql file, 116                            |
| SET SERVEROUTPUT, 109–110                | SQLPLUS_RELEASE variable, 128                    |
| SHOW, 116                                | SQLPROMPT command, 118                           |
| SHUTDOWN, 502–505                        | SQLPROMPT variable, 108                          |
| SPOOL, 106, 120, 121                     | SQLTERMINATOR variable, 132                      |
| SQLPROMPT, 118                           | SQL_TRACE parameter, 467, 1101                    |
| SQLTERMINATOR, 132                       | SQTUNE_CATEGORY parameter, 1115                   |
| START, 106                               | SQL/XML operators, 1249                          |
| STARTUP, 497–499                         | SREADTIM statistic, 1061                         |
| STORE, 106, 115                          | ssh command, UNIX/Linux, 79                      |
| TTITLE, 123                              | Staging directory, Oracle installation, 416      |
| UNDEFINE, 126                            | Staging element, Oracle Streams, 671             |
| SQL*Plus environment variables, 107–108  | STALE_TOLERATED mode, 316, 465                   |
| AUTOCOMMIT, 133                          | Standard auditing see auditing                    |
| BLOCKTERMINATOR, 132                     | Standard Edition, 417                            |
| changing, 109                            |                                                    |
| creating/deleting session variables, 126 |                                                    |
| displaying all values, 116               |                                                    |
standard error/input/output, UNIX, 56
standard.rsp response file template, 422
standby databases
  see also Oracle Data Guard
  avoiding data center disasters, 802
  high-availability systems, 798
  Oracle Data Guard and, 799–800
  physical and logical, 799
start command, lsnrctl utility, 523
START command, SQL*Plus, 106
START keyword, 1061
START WITH clause, SQL, 1232
START_CAPTURE procedure, 1211
START_DATE attribute, 999, 1014
starting database from SQL*Plus, 497–499
START_JOB parameter, Data Pump, 703, 704, 713
START-POOL procedure, 532
STAR_TRANSFORMATION_ENABLED parameter, 1078
START_REDEF_TABLE procedure, 938, 939
START_REPLAY procedure, 1214
START_TIME attribute, CREATE_WINDOW, 1015
STARTUP command, ASM instances, 907, 908
STARTUP command, SQL*Plus, 497–499
STARTUP MOUNT command, 492, 498
startup scripts, post-installation update, 423
startup time, improving for instances, 805
STARTUP UPGRADE command, 427, 437
stateful/stateless alerts, 952
Statement object, JDBC, 539
statement timed out error, 384
statement-level audit, 587
statement-level read consistency, 345, 346, 356
statements see SQL statements
statement_types parameter, ADD_POLICY, 594
static data dictionary views, 204
static parameters, 448, 496
static security policy, 584
statistical functions, SQL, 1231
statistics
  see also Automatic Optimizer Statistics Collection
  Active Session History (ASH), 971–975
  automated tasks feature, 211
  Automatic Tuning Optimizer (ATO), 1112
  Automatic Workload Repository (AWR), 210
  AUX_STATS$ table, 1060
  calculating, SQL*Plus, 123
  collecting fixed object statistics, 1062
  collecting operating system statistics, 1086, 1086–1062
  collecting optimizer statistics, 299
  collecting real dictionary table statistics, 1063
  collecting statistics for column groups, 1058, 1059
  collecting statistics on dictionary tables, 1062–1063
  COMPUTE STATISTICS option, 299
creating column groups, 1059
database usage metrics, 151
Database Usage Statistics property sheet, 152
database wait statistics, 1162–1163
DBA_TAB_COL_STATISTICS view, 1049
DBA_TAB_STATISTICS table, 1049
DBMS_STATS package collecting, 1053–1056
deferring publishing of statistics, 1056–1057
determining publishing status of statistics, 1056
dynamically sampling data, 463
effect when statistics not collected, 1063
examining database feature-usage statistics, 152
expression statistics, 1059
extended optimizer statistics, 1058–1060
frequency of statistics collection, 1063
GATHER_STATS_JOB, 1047–1049
INDEXSTATS view, 334
making pending statistics public, 1057
manageability monitor (MMON) process, 185
managing/monitoring database, 213
manual optimizer statistics collection, 900
multicolumn statistics, 1058
operating system statistics, 1061
OPTIMIZER_USE_PENDING_STATISTICS parameter, 463
Oracle recommendation, 1065
problems due to bad statistics, 1191
providing statistics to CBO, 1053–1056
refreshing statistics frequently, 1086
sampling data, 1053
segment-level statistics, 1173
setting publishing status of statistics, 1057
specifying level of statistics collected, 461
specifying use of pending statistics, 463
statistics retention period, AWR, 960
system usage statistics, 1181
time-model statistics, 879–880
TIMED_STATISTICS parameter, 468
turning on statistics collection, 1167
using histograms, 1086–1087
using OEM to collect optimizer statistics, 1064–1065
VSCPOOL_CC_STATS view, 533
VSCPOOL_STAT view, 533
VSFILESTAT view, 246
V$SESSSTAT view, 959
V$SYSSTAT view, 959
when manual collection is required, 1054
statistics collection job, 1132
statistics management, 147
STATISTICS_LEVEL parameter, 461
ADDM, 881, 883, 888
automatic optimizer statistics collection, 898, 899
creating/deleting snapshot baselines, 963
server-generated alerts feature, 953
setting trace initialization parameters, 1100
Statspack utility, 959
status command, lsnrctl utility, 521
STATUS command, Data Pump, 703, 704, 713
STATUS function, DBMS_RESULT_CACHE, 1123
status information, Grid Control, 159
STATUS parameter, Data Pump, 699, 708
status.sql script, 125
stop command, lsnrctl utility, 523
STOP keyword, operating system statistics, 1061
STOP_JOB command, Data Pump, 686, 702, 703, 704, 713
STOP_ON_WINDOW_CLOSE attribute, 1016
STOP_POOL procedure, 532
storage, 729
ASM (Automatic Storage Management), 900–920
Storage Area Networks (SANs), 93
storage element (se), Oracle Secure Backup, 789
STORAGE option, Data Pump Import, 711
Storage Options window, DBCA, 487
storage parameters, 220–222
storage, database
Automatic Storage Management (ASM), 209
database management, 146
disk storage requirements, 392
flash recovery area, 734
implementing physical database design, 37
storage, UNIX, 93–95
STORAGE_CHANGE parameter, 978
STORE command, SQL*Plus, 106, 115, 116
STORE IN clause, CREATE TABLE, 283
stored execution plan, 1112
stored functions, inline, 1074–1075
stored outlines, 1077–1080
stored procedures
database security, 577
definer’s rights, creating with, 573
displaying output on screen, 109
invoker’s rights, creating with, 573
Java stored procedures, 1252
stored scripts, RMAN, 747
stream record format, SQL*Loader, 631
streaming, table functions, 662
streams pool, 187, 193, 1148
STREAMSIZE parameter, SQL*Loader, 641
STREAMS_POOL_SIZE parameter, 193, 195, 673
stretch cluster, ASM, 309
stripe size, RAID systems, 88
stripe sizes, logical volumes, 1159
stripe-and-mirror-everything (SAME), 399, 1160
striping
ASM, 901, 914
disk striping, 88, 467
RAID options, 89–90
STRIPING attribute, ASM, 910
strong authentication
ldap_directory_sysauth parameter, 615
STS (SQL Tuning Sets), 1115, 1217, 1218
subclasses, object-oriented database model, 39
SUBMIT_PENDING_AREA procedure, 562
SUBPARTITION BY HASH clause, 288
SUBPARTITION BY LIST clause, 288
SUBPARTITION BY RANGE clause, 289, 290
SUB_PLAN parameter, resource plans, 560
subqueries, SQL, 263, 1067, 1236–1237
substitution variables, SQL*Plus, 126
SUBSTR function, SQL, 663, 1228
success code, SQL*Loader, 639
SUM function, SQL, 1229
summaries, materialized views, 314
Summary window, Oracle Universal Installer, 419
super administrator account see SYSMAN
supplemental logging
Flashback Transaction Backout feature, 868
LogMiner utility, 842–843
Support Workbench, 211, 1022, 1028–1032
invoking SQL Repair Advisor from, 1035
suppressed mode, Oracle Universal Installer, 421, 422
suspended operations, 385, 386
suspending databases, 505, 945
swap ins/outs, memory management, 81
swap space
preinstallation checklist, 413
preinstallation checks, 401, 403
virtual memory, 1158
swapping
affecting performance, 1205
system usage problems, 1188
SWITCH command, RMAN, 757
using RMAN for TSPITR, 841
SWITCH DATABASE command, RMAN, 816
SWITCH_FOR_CALL parameter, 561, 565
SWITCH_GROUP parameter, 561
SWITCH_IO_MEGABYTES parameter, 561
SWITCH_IO_REQS parameter, 561
SWITCH_XYZ parameters, plan directives, 942
symbolic links, UNIX, 58
automatic database startup, 501
symbolic names, UNIX, 47, 49
symbolic notation, modifying permissions, 60, 61
synchronization
ASM and Cluster Synchronization Service, 902–904
fast mirror resync feature, ASM, 908–909
SYNCHRONIZATION parameter, 1214
SYNC_INTERIM_TABLE procedure, 940
synonyms, 265, 324–327
DBA_SYNONYMS view, 326, 329
SYS parameter, TKPROF utility, 1102
SYS schema, 569
SYS user
   AUDIT_SYS_OPERATIONS parameter, 588
   creating database, 481
   data dictionary tables, 203
   default password, 475
SYS_AUDS table, 588, 595, 596
sys.fga_auds table, 595
SYSASM privilege, 101, 570
   OSASM group, 407
   revoking, 902
SYS_AUTO_SQL_TUNING_TASK procedure, 1118
SYSAUX clause, 239
Sysaux tablespace, 172, 215, 238–240
   AWR storage space requirement, 964
   creating database, 481
   creating, OMF, 926
   creating/locating OMF files, 252
   database creation log, 484
   manual database upgrade process, 438
   Pre-Upgrade Information Tool, 428
   removing tablespaces, 225, 240
   renaming tablespaces, 228, 240
   sizing for database creation, 444
   various user quotas in, 546
sys_context function, 579
sysctl.conf file, 405
SYSDATE function, SQL, 449, 1229
sysdate variable, SQL*Loader, 637
SYSDBA privilege, 570, 613
   ASM instances, 904
   connecting using CONNECT command, 100
   creating databases, 475
   password file, 599
   starting SQL*Plus session from command line, 99
SYSDBA role, 451, 458, 451
SYS_DEFAULT_CONNECTION_POOL, 532
SYS_GROUP resource consumer group, 558, 563
SYSMAN (super administrator account), 148, 157, 158
SYSOPER privilege, 570
   ASM instances, 904
   connecting using CONNECT command, 100
   password file, 599
   starting SQL*Plus session from command line, 99
SYSOPER role, 451, 458
SYSTEM account, 475, 481
System Activity Reporter see sar command, UNIX
system administration, Oracle DBA, 12–13
system administration, UNIX see UNIX system administration, 76
system administrator
   post-installation tasks, 423–424
      creating additional operating system accounts, 424
      updating shutdown/startup scripts, 423
   preinstallation tasks, 401–410
      applying OS packages, 403
      changing login scripts, 406
      changing shell limits, 406
      checking kernel version, 402
      checking memory and physical space, 403
      checking operating system version, 402
      checking OS packages, 403
      creating database directories, 410
      creating directories, 409–410
      creating flash recovery area, 410
      creating groups, 406–408
      creating mount points, 404
      creating Oracle base directory, 409
      creating Oracle home directory, 410
      creating Oracle Inventory directory, 409
      creating Oracle software owner user, 408–409
      creating ORAINVENTORY group, 407
      creating OSASM/OSDBA/OSOPER groups, 407
      reconfiguring kernel, 404–406
      setting file permissions, 409
      verifying operating system software, 402–403
      verifying unprivileged user exists, 408
   system change number see SCN
   system configuration files, UNIX, 63
   system failures, 802
   system global area see SGA
   System I/O wait class, 1164
   system management, DBA role, 5–7
   system metrics, 951
   system monitor process see SMON
   system monitoring
      DBA role, 5, 15
      GlancePlus package, 84
   system partitioning, 287
   system performance
      application knowledge for diagnosis, 1182
      CPU usage, 1203
      database load affecting, 1205
      eliminating wait event contention, 1208
      evaluating system performance, 1152–1159
      CPU performance, 1153–1158
      disk I/O, 1158–1159
      operating system physical memory, 1158
      examining system performance, 1181–1182
      I/O activity affecting, 1204
      measuring I/O performance, 1159–1161
      measuring system performance, 1203
      memory affecting, 1205
      network-related problems, 1203
      Oracle wait events, 1175–1181
      reducing I/O contention, 1160
      SAME guidelines for optimal disk usage, 1160
      wait events affecting, 1206
   system privileges, 567–570
   System Privileges page, Database Control, 150
System tablespace, 172, 215
  creating database, 481
  creating, OMF, 926
  creating/locating OMF files, 252
  database creation log, 484
  default tablespace, need for, 544
  default temporary tablespaces, 232
  removing tablespaces, 225
  renaming tablespaces, 228
  sizing for database creation, 444
  storing undo data, 357
  Sysaux tablespace and, 238
  using multiple block size feature, 223
  system usage statistic, 1181
  system-control statements, SQL, 262
  system-level triggers, 591–593
  systems management, OEM, 139
  systemstate dump, database hangs, 1193
  SYS_TIKCS system usage statistic, 1182

T
  table compression, 274–276, 1076
  table functions, 662–667
    ETL components, 626
  table locks, explicit, 350–351
  table lookups, minimizing, 1067
  table mode, Data Pump Export, 688
  table operations, 267
  TABLE parameter, TKPROF utility, 1102
  table privileges, 371
  table scans, 467
  table types
    table functions, 664
    user-defined data types, 264
  table versioning, workspaces, 387
  TABLE_EXISTS_ACTION parameter, Data
    Pump, 708
table-level Flashback techniques, 848
  table-level locks, 349
tables
    ALTER TABLE command, 217
    avoiding unnecessary full table scans, 1075
    caching small tables in memory, 1090
    copying tables, SQL*Plus, 132–133
    creating tables first, 215
    data dictionary tables, 203
    designing different types of, 35
dictionary tables, 1062
    dropping tables, 224
      recovering dropped table, 548
      restoring dropped tables, 851–852
dynamic performance tables, 203
    encrypting table columns, 607–608
    fixed dictionary tables, 1062
    Flashback Table, 366, 376–378
    full table scans, 1052
    heap-organized tables, 1072
    indexing strategy, 1070–1073
    keys, 20
  listing columns and specifications of, 119
  locking issues, 1189
  multitable inserts, 660–662
  naming conventions, 36
  online table redefinition, 935, 936–941
  ordering of columns, 20
  partitioned tables improving SQL
    processing, 1076
  permanently removing, 852–853
  real dictionary tables, 1063
  rebuilding tables regularly, 1089
  selecting data from multiple tables, SQL,
    1232–1234
  separating table and index data, 170
  sizing for database creation, 444
  SQL moving tables online, 935
  table access by ROWID, 1052
  table structures, 36
  transforming ER diagrams into relational
    tables, 35
  TABLES parameter, Data Pump, 688, 708
  tables, Oracle, 265–276
    adding columns to, 271
    clustered tables, 266
    clusters, 295–296
    creating, 268–269, 273
    data dictionary views for managing, 292–295
    database integrity constraints, 306–310
    DBA_ALL_TABLES view, 330
    DBA_EXTERNAL_TABLES view, 330
    DBA_PART_TABLES view, 331
    DBA_TAB_COLUMNS view, 332
    DBA_TABLES view, 292, 330
    DBA_TAB_MODIFICATIONS view, 331
    DBA_TAB_PARTITIONS view, 330
default values for columns, 270
dropping columns from, 271
dropping tables, 276
dual table, 265
estimating size before creating, 266, 268
eexternal tables, 280
extracting DDL statements for, 294
full table scans, 297
getting details about tables, 292
guidelines for creating indexes, 297
heap-organized tables, 265
index-organized tables, 265, 278–280
inserting data from another table, 269
moving tables between tablespaces, 273
null value, 269
object tables, 265
organizing tables, 265
partitioned tables, 265, 280–292
read-only mode, 273–274
relational tables, 265
removing all data from, 272
renaming columns, 272
renaming tables, 272
setting columns as unused, 271
storing rows, 267

table compression, 274–276

temporary tables, 277–278

views and, 312

virtual columns, 270–271

virtual tables see views

tablespace alerts, 956–958

tablespace backups, 728, 794

tablespace compression, 274

tablespace encryption, 608–610

TABLESPACE GROUP clause, 233

tablespace metrics, 951

tablespace mode, 688

adding tablespaces, OMF, 253, 927

assigning tablespace quotas to new users, 545

backing up with RMAN, 777

backup guidelines, 730

bigfile tablespaces (BFTs), 172, 236–238

block size, 167

changing default tablespace type, 238

CREATE TABLESPACE statement, 218

creating database, 481, 482

creating permanent tablespaces, 219

creating tables or indexes, 215

creating tablespaces, 218

clauses/options/parameters, 219–223

determining extent sizing and allocation, 220–222

NOLOGGING clause, 227

with nonstandard block sizes, 223

data block sizes and, 171–172

data dictionary views managing, 243–246

datafiles, 170, 173, 219–220

DBCA creating additional tablespaces, 489

default permanent tablespaces, 235–236

default tablespaces, 170

description, 166, 170, 215

encrypted tablespaces, 240–243

enlarging, 174

expanding tablespaces, 223–224

extent sizing, 216

free space, 255–256

locally managed tablespaces, 172

making tablespace offline, 228

managing availability of, 228, 229

mandatory tablespaces, 444

migrating tablespaces, 217

moving tables between, 273

multiple database block sizes and buffer cache, 190

naming conventions, 398

OFFLINE clauses, 228, 229

operating system files, 215

Oracle Managed Files (OMF), 247–253

permanent tablespaces, 172, 215

Pre-Upgrade Information Tool, 428

read-only tablespaces, 172, 229

recovering tablespaces, 817–818

recovering, SQL*Plus, 134

removing tablespaces, 224–225

renaming tablespaces, 228

RESIZE clause, 219

revisiting tablespace quotas to users, 546

segment space management, 221, 217–218, 219

separating table and index data, 170, 171

setting alert thresholds, 957

sizing for database creation, 444

smallfile tablespaces, 172, 236

storage allocation to database objects, 222

storage parameters, 220–222

Sysaux tablespace, 172, 215, 238–240

System tablespace, 172, 215

tablespace quotas, 226

tablespace space alerts, 226–227

temporary tablespace groups, 233–235

temporary tablespaces, 172, 215, 229–235

transportable tablespaces, 171, 716–723

undo tablespaces, 172, 215, 356

UNDO_TABLESPACE parameter, 460

UNLIMITED TABLESPACE privilege, 268

unlimited tablespace usage rights, 546

upgrading with DBUA, 430

user tablespaces, 225

users creating tablespaces, 546

using multiple block size feature, 223

viewing tablespace quotas, 546

TABLESPACES parameter, Data Pump, 688, 708

TAG parameter, RMAN, 756

tags, RMAN backup tags, 753

tail command, UNIX/Linux, 76

tape backups, 726, 729

RMAN (Recovery Manager), 742

tape library components, Oracle Secure Backup, 789

tar command, UNIX/Linux, 76

target database, RMAN, 743, 840

Target Privileges, Database Control roles, 150

Targets page, Grid Control, 159

TCP/IP protocol, 517, 529

TCP/IP Protocol page, 528

TCP_INVITED_NODES parameter, 615

TDE (Transparent Data Encryption), 240, 241, 608

tee command, RMAN output, 747

telnet, 46, 78

temp files, OMF file-naming conventions, 249, 923

TEMPFILE clause


tablespace, 219, 230

Oracle Managed Files (OMF), 247
TEMPLATE.TNAME.XYZ attributes, ASM, 910
templates
  baseline templates, AWR, 965–971
  using templates with aliases, ASM, 917
TEMPORARY clause, CREATE
  TABLESPACE, 230
temporary directory, 413, 415
temporary files, UNIX, 63
temporary segments, 169, 184
temporary space, 401, 413
temporary tables, 277–278
temporary tablespaces, 172, 215, 229–235
  altering, 231
  AUTOALLOCATE clause, 233
  creating database, 482
  creating, 230–231
  creating/locating OMF files, 252, 927
database creation log, 484
  DBA_TEMP_FILES view, 246
default temporary tablespace, 232, 234
  DEFAULT TEMPORARY TABLESPACE
  clause, 232
dropping, 230
  encrypted tablespaces, 241
  extent sizing, 230
  failure to assign default space, 232
  managing users, 544
  Oracle size recommendation, 231
  shrinking, 231–232
  sizing for database creation, 444
  sort segment, 230
  V$TEMPFILE view, 246
TERMINAL attribute, USERENV namespace, 579, 580
terminal emulators, UNIX, 46
TERMINATED BY clause, SQL*Loader, 632
terminating database sessions, 75
terminating processes with kill, UNIX, 75
TERMOUT variable, SQL*Plus, 108
test command, UNIX/Linux, 69, 71
test databases, 9
text editors, UNIX, 63–65
text extraction utilities, UNIX, 65
text pages, Oracle process, 1190
text scripts, RMAN, 747
text, SQL*Plus, 130, 131
theta-join operation, 21
THINK_TIME_XYZ parameters, 1214
Third Normal Form (3NF), 33
threads
  identifying user threads in Windows, 621
  log%_%s.%_r format, archived redo logs, 823
  PARALLEL parameter, Data Pump, 700
thresholds
  adaptive thresholds, 954
  alert thresholds, 226–227, 953
  critical alert thresholds, 226, 952, 954, 956
  Database Control setting alert thresholds, 954
  Edit Thresholds page, 954, 955
  fixed values, 954
  percentage of maximum, 954
  proactive tablespace alerts, 956
  setting for metrics, 953
  significance level, 954
  threshold-based alerts, 952
  warning alert thresholds, 226, 952, 954, 956
time
  CONNECT_TIME parameter, 549
  ENABLE_AT_TIME procedure, 368
  execution time limit resource allocation method, 555
  idle time resource allocation method, 555
  IDLE_TIME parameter, 549
  logical time stamp, 199
  PASSWORD_XYZ_TIME parameters, 550
  Pre-Upgrade Information Tool, 428
  RESUMABLE_TIMEOUT parameter, 470
time-based recovery, RMAN, 821
  TIME column, top command, 84
time-model statistics
  ADDM, 879–880
  data collected by AWR, 960
  DB time metric, 878
  Time Model Statistics section, AWR reports, 969
  Time Periods Comparison feature, 1206
  TIME variable, SQL*Plus, 108
time zone data type, 430
time-based recovery, 820, 824
  TIMED_STATISTICS parameter, 468, 1100
  TIMEHINT parameter, 379, 869
  TIMEOUT clause, ALTER SESSION, 384
  TIMESTAMP data type, 1223
time-stamping methods, data concurrency, 347
time stamps
  Flashback Database example, 860
  TIMESTAMP_TO_SCN function, 847
timing
  CLEAR TIMING command, SQL*Plus, 115
  TIMING variable, SQL*Plus, 108
titles
  placing title on page, SQL*Plus, 123
  TKPROF utility, SQL, 1099, 1101, 1102–1105
  end-to-end tracing, 1107
  examining parse information, 1136
  reducing parse time CPU usage, 1157
/tmp directory, UNIX, 63, 403, 404
TMPDIR variable, 413
TNS_ADMIN variable, 412, 526
tnsnames map file, 533
tnsnames.ora file
  backup guidelines, 729
  cloning databases with RMAN, 836
  connecting to database, 100
  connecting to Oracle, 206
  connecting to SQL*Plus through Windows
    GUI, 101
  connecting using CONNECT command, 100
Easy connect naming method, 530
external naming method, 533
installing Oracle Client, 518
local naming method, connections, 525, 526
location of, 412
modifying manually, 526–528
modifying with NCA, 528–529
remotely connecting to Oracle database, 98
removing EXTPROC functionality, 614
specifying DRCP connection, 532
typical file, 526
TO SCN clause, FLASHBACK TABLE, 378
TOAD software, SQL tuning, 1120
TO_CHAR function, SQL, 1230
TO_DATE function, SQL, 1229
TO_NUMBER function, SQL, 1230
tools, Oracle, 213–214
Top 5 Timed Events section, AWR reports, 969
Top Activity page, Database Control, 1200
Top Background Events section, ASH
reports, 973
Top Blocking Sessions section, ASH reports, 974
top command, UNIX/Linux, 84, 1186
Top Service/Module section, ASH reports, 973
Top Sessions page, Database Control, 1200
Top Sessions section, ASH reports, 974
Top SQL Command Types section, ASH reports, 974
Top SQL statements, 1115
Top SQL Statements section, ASH reports, 974
Top SQL Using Literals section, ASH reports, 974
Top User Events section, ASH reports, 973
Top-N analysis, SQL subqueries, 1236
Total Recall, 1209
TO_TIMESTAMP function, SQL, 1229
touch command, UNIX/Linux, 63
touch count, buffer cache, 1146, 1148
TO_XYZ conversion functions, SQL, 1223
trace directory, ADR, 178, 1024
trace files, 178, 958
tracing
 collecting timed statistics during, 468
creating trace in user dump directory, 1136
DBMS_MONITOR package, 1175
DBMS_SYSTEM package, 1175
enabling, 1135
end-to-end tracing, 1105–1107
event 10046 tracing SQL code, 1174
oradebug utility performing trace, 1174
sec_protocol_error_trace_action
parameter, 615
SQL Trace utility, 1099–1102
SQL_TRACE parameter, 467
TKPROF utility, SQL, 1102–1105
tracking block changes, RMAN, 779
training, DBA, 10–11
Oracle by Example (OBE), 14
Oracle Database Two-Day DBA course, 14
Oracle MetaLink, 15
Oracle online learning program, 11
Oracle Web Conference (OWC), 15
resources, 13–14
Transaction Backout see Flashback
Transaction Backout
transaction control statements, SQL, 263
transaction identifier, 371
transaction management, 337
transaction migration, 337
Transaction Playback page, Grid Control, 160
transaction recovery, 804, 806
TRANSACTION_BACKOUT procedure, 379, 868, 869
transaction-level read consistency, 345, 346
transactions, 196–197, 337–340
abnormal program failure, 338
ACID properties, 340
automatic checkpoint tuning, 932
backing out transactions, 869–870
COMMIT statement, 338–339
COMMIT_LOGGING parameter, 339
committing transactions, 196, 197–198, 340
COMMIT_WAIT parameter, 339
compensation transactions, 868
creating transaction temporary table, 278
data concurrency, 341–342, 347–355
data consistency and, 196
database consistency, 342
DDL and DML statements, 338
fast commit mechanism, 199
Flashback error correction using undo data, 366–368
Flashback Transaction, 366, 379–380
Flashback Transaction Query, 366, 372–375
how Oracle processes transactions, 196
ISO transaction standard, 342
isolation levels, 342–346
limiting execution times, 942–943
locking and, 350
long-running transactions affecting performance, 1202
making permanent in database files, 181
monitoring backout of, 380
monitoring performance with Grid Control, 160
naming, 340
normal program conclusion, 338
processing, 196, 338
redo log files, 176
ROLLBACK statement, 338, 339–340
rolling back transactions, 196, 198
SAVEPOINT command, 339
SHUTDOWN command, 503
statement-level read consistency, 356
system change number (SCN), 200
undo data providing read consistency, 356–366
undo management, 200, 921
undo segment, 338
TRANSACTIONS parameter, 455
transactions, Oracle
  autonomous transactions, 380–382
  DBMS_TRANSACTION package, 380
  discrete transactions, 380
  managing long transactions, 386–388
  managing transactions, 380–382
  Resumable Space Allocation, 382–386
  Workspace Manager, 386–388
transaction-set consistent data, 199
transfer rate, RAID systems, 88
transform operator, SQL/XML, 1249
TRANSFORM parameter, Data Pump, 710–711
transformations, SQL, 1051
transforming data, 626, 656–670
  see also ETL (extraction, transformation, loading)
  deriving data from existing tables, 657
  MERGE statement, 658–660
  MODEL clause, 667–670
  SQL*Loader/external tables compared, 646
  table functions, 662–667
  using SQL to transform data, 658–667
transform-then-load method, 626
transform-while-loading method, 626
table functions, 662
transient files, flash recovery area, 735
transition point, partitioning, 282
transparency, synonyms, 325
Transparent Application Failover feature, 802
Transparent Data Encryption (TDE), 240, 241, 604–608
crypting table columns, 607–608
cryption algorithms, 608
generating master encryption key, 607
Oracle Wallet, 605–606
transparent encryption, 763
TRANSPARENT value, Data Pump, 695, 696
TRANSPORTABLE parameter, Data Pump, 694, 710
transportable tablespaces, 171, 716–723
  converting datafiles to match endian format, 721
  copying export/tablespace files to target, 719
  copying files to target system, 722
  Data Pump Import importing metadata, 723
determining endian format of platforms, 720
e nsuring tablespaces are self contained, 721
ETL components, 626
exporting dictionary metadata for, 718
exporting metadata using Data Pump, 721
generating transportable tablespace set, 717–719
importing metadata from dump file, 719
making tablespace read-only, 721
performing tablespace import, 719–720
referential integrity constraints, 716
selecting tablespace to transport, 716
self-contained criteria, 716
TRANSPORT_FULL_CHECK parameter, 691
transporting tablespace across platforms, 720–723
transporting tablespaces between databases, 716–720
TRANSPORT_SET_CHECK procedure, 717
uses for, 716
TRANSPORTABLE_TABLESPACES parameter, 705
TRANSPORT_DATAFILES parameter, 708
TRANSPORT_FULL_CHECK parameter, 691, 708, 717, 721
TRANSPORT_SET_CHECK procedure, 717, 721
TRANSPORT_TABLESPACE parameter, 718
TRANSPORT_TABLESPACES parameter, 688, 708
trees, B-tree index, 298
trial recovery, 864–866
triggers, 328–329, 590–593
deriving data from existing tables, 657
ensuring data validity, 37
SQL*Loader direct-path loading, 641
TRIM function, SQL, 1228
troubleshooting
  alert log file, 178
  recovery errors, 866–870
TRUNC function, SQL, 1229
TRUNCATE command, SQL, 272, 1226
TRUSTED mode
  QUERY_REWRITE_INTEGRITY parameter, 316, 465
try ... catch blocks, Java, 540
TSPITR (tablespace point-in-time recovery), 808, 840–841
TTITLE command, SQL*Plus, 123, 116, 124
TTS_FULL_CHECK parameter, 718
TUNE_MVIEW procedure, 317
tuning
  see also performance tuning
  automatic checkpoint tuning, 932–933
  Automatic SQL Tuning Advisor, 209, 212, 1115–1120
  automatic undo retention tuning, 209
  buffer cache, 1144–1148
  hard parsing and soft parsing, 1135–1143
  instance tuning, 1129–1130, 1194–1209
  see also instance performance
Java pool, 1148
large pool, 1148
Oracle memory, 1132–1152
Oracle Tuning Pack, 149, 949
PGA memory, 1148–1152
proactive tuning, 1042
self-tuning mechanism, AWR, 959
shared pool, 1133–1135
SQL Tuning Advisor, 212, 1111–1115
streams pool, 1148
tuning SQL statements, 1126–1127
tuning mode, Oracle optimizer, 1111
tuning pack, Server Manageability Packs, 459
tuples, 20, 21
TWO_TASK environment variable, 519
txnames parameter, 869
%TYPE attribute, PL/SQL, 1242
type inheritance, abstract data types, 1240
TYPICAL value, STATISTICS_LEVEL
parameter, 461

U
UDUMP (default trace directory), 168
umask command, 409
UMASK variable, UNIX, 61, 613
uname command, UNIX/Linux, 49
UNASSIGN_ACL procedure, 617
UNCOMPRESS clause, ALTER TABLE, 275
UNDEFINE command, SQL*Plus, 126
underscore (_) character, SQL, 1224
Undo Advisor, 977, 980–981
automatic performance tuning
features, 1132
sizing undo tablespaces, 359, 362, 365
undo data, 356–366
active/inactive, 359
Automatic Undo Management (AUM), 356–362
backup and recovery architecture, 201
committed/uncommitted, 359
committing transactions, 338
determining default tablespace for, 460
Flashback error correction using, 366–368
Flashback Table, 376
Flashback Transaction, 379
Flashback Transaction Query, 373
insufficient data to flash back, 377
OEM managing, 365–366
retaining in undo tablespace, 359
RETENTION GUARANTEE clause, 367
storing, 460
storing in System tablespace, 357
undo records, 356
UNDO_RETENTION parameter, 373
Undo Generation and Tablespace Usage
graph, 366
UNDO keyword, CREATE TABLESPACE, 358
undo management, 200, 560
Automatic Undo Management (AUM), 921
undo pool method, Database Resource
Manager, 555
undo records see undo data
undo retention, 360
default, 361
guaranteed undo retention, 362–365
RETENTION GUARANTEE clause, 363
undo retention tuning, 209
undo segments, 169
AUM, 200, 361, 357, 922
AWR determining number of, 359
committing transactions, 198
data consistency, 199
fast ramping up of, 359
managing undo information in, 359
rolling back transactions, 198
transactions, 338
undo space
managing automatically, 357
managing undo space information, 364–365
undo tablespaces, 172, 215
adding space to, 358
auto-extensible undo tablespaces, 358
reason against auto-extensible, 362
automatic undo retention, 361
before-image records, 176, 197, 356
changing and editing, 365
CREATE UNDO TABLESPACE
statement, 218
creating, 358
creating database, 482
creating, OMF, 927
creating/locating OMF files, 252
database creation log, 484
default choice of, 358
dropping, 364
encrypted tablespaces, 241
inserting new row, 197
multiple undo tablespaces, 357, 358
RETENTION GUARANTEE clause, 374
rolling back transactions, 198
setting up Oracle Streams, 672
sizing, 358, 360, 362, 364
for database creation, 444
OEM Undo Advisor assisting, 365
system activity and usage statistics, 365
using Flashback features, 367
undocumented initialization parameters, 473
UNDO_MANAGEMENT parameter, 200,
357, 460
UNDO_POOL parameter, 365, 943
undo-related parameters, 460–461
UNDO_RETENTION parameter, 200,
359–362, 460
automatic undo retention tuning, 209
Flashback Query, 367
Flashback Table, 377
Flashback Transaction Query, 373
Flashback Versions Query, 370
guaranteed undo retention, 362, 367
setting up Oracle Streams, 673
snapshot-too-old error, 364
UNDO_TABLESPACE parameter, 357–359, 460
UNDROP clause, ALTER DISKGROUP, 916
UNIFORM extents option, 216, 221
using bigfile tablespaces, 237
UNIFORM SIZE clause, creating tablespaces,
221, 230
uninstalling Oracle, 425–426
UNION operation, 21, 1071
UNION operator, SQL, 1228
uniq command, UNIX/Linux, 68
UNIQUE constraint, 300, 307
unique identifiers see primary keys
unique indexes, 297, 299
Universal Installer see Oracle Universal Installer
UNIX, 43
- accessing UNIX system, 46–48
- archiving, 76
- backup and restore utilities, 76–77
- changing shell prompt, 51
- customizing environment, 55
- directory structure, 47
- disk availability, 87
- disk partitioning, 87
- disk performance, 87
- disk striping, 88
- displaying environment variables, 54
- Emacs text editor, 65
- file systems, creating, 87
- files, 62
- flow control structures, 71–74
- input/output redirection, 56–57
- kernel, 45
- Linux compared, 43, 45
- logical volumes, 88
- monitoring disk usage, 86
- operating system level permissions, 613
- redirection operators, 56
- remote access to UNIX server, 78
- scheduling jobs, 77
- session, 47
- shell scripting, 68–74
- starting SQL*Plus session from, 99
- vi text editor, 63–65
- Vim text editor, 65
UNIX boxes, 47
UNIX commands, 48–57
- basic commands, 48
- batch mode, 54
- bin directory, 48
- built-in commands, 48
- controlling output of commands, 52
- editing previous commands, 50
- interactive mode, 54
- passing output between commands, 52
- retrieving previous commands, 50
UNIX commands, list of
- at, 78
- batch, 78
- case, 74
- cat, 52, 58
- cd, 48, 62
- chgrp, 62
- chmod, 60, 61, 70
- chsh, 46
- cp, 59
- cpio, 76, 77
- crontab, 77–78
what is, 51
where is, 49
which, 49
who, 50
who am I, 50
UNIX directories see directories, UNIX
UNIX disk storage see disk storage, UNIX
UNIX files see files, UNIX
UNIX knowledge, DBA, 12
UNIX processes, 74–76
runnable process, 80
thread analogy, 179
UNIX shell scripts see shell scripts, UNIX
UNIX shells see shells, UNIX
UNIX system administration, 76
accessing remote computers, 78
backup and restore utilities, 76–77
crontab and automating scripts, 77–78
Oracle DBA and, 76
performance monitoring, 80–81
performance monitoring tools, 81–85
remote copying, 79
remote login (Rlogin), 78
scheduling database tasks, 77–78
Secure Shell (SSH), 79
sending and receiving files using FTP, 79
telnet, 78
UNIX variables, 53, 54
UNKEEP procedure, 1138
UNKNOWN status, Oracle listener, 522
UNLIMITED TABLESPACE privilege, 268, 546
UNLIMITED value, default profile, 550, 551
unloading data, populating external tables, 650
UNPACK_STGTAB_SQLSET procedure, 1218
unprivileged user, verifying existence of, 408
unrecoverable datafiles, 759
UNRECOVERABLE parameter
backup guidelines, 729
SQL*Loader utility, 640, 641, 642
until-do-done loop, UNIX, 73
unused block compression feature, RMAN, 742
UNUSED_SPACE procedure, 255
update anomaly, 29, 32, 33
UPDATE GLOBAL INDEXES option, 302
UPDATE statement, 263, 287, 1226, 1242
UPDATE-ELSE-INSERT operation, 658
.updates
Critical Patch Updates, 617
lost-update problem, 341
RULES UPDATE specification, 669
upgrade actions script, 438
UPGRADE CATALOG command, RMAN, 771
Upgrade Results window, DBUA, 433
upgrade script, 438
Upgrade Summary window, DBUA, 432
upgrade.log spool file, 435
upgrading databases
see also manual database upgrade process
preserving database performance, 1080
upgrading to Oracle Database 11g, 426–441
case sensitivity of passwords, 491
Database Upgrade Assistant (DBUA), 428,
430–433
ensuring database compatibility, 452
manual database upgrade process, 427,
434–441
methods and tools, 427–430
post-upgrade actions, 441
Post-Upgrade Status Tool, 429
preparing database for upgrading, 441
Pre-Upgrade Information Tool, 428–429
resetting passwords, post-upgrade, 441
retaining behavior of older software
release, 463
upgrade paths, 426–427
upserts, 626, 658, 659, 669
used_bytes attribute, CREATE_INDEX_COST, 299
user accounts, database security, 611
user authentication, 596–602
user class, Oracle Secure Backup, 788
user-created variables, UNIX, 53
user-defined object types, 264
user-defined variables, UNIX, 54
user errors, 802, 803
User I/O wait class, 1163, 1164
user management, 544, 619, 620
user processes, 179
user profile file, 110–111
user profiles, 548–553
altering, 552, 619
FAILED_LOGIN_ATTEMPTS parameter, 612
managing resources, 554
user pseudo-variable, SQL*Loader, 643
user tablespaces, 225
USER variable, SQL*Plus, 118, 119, 127
USER views, 204
useradd command, 408
USER_ADVISOR_XYZ views, 324
USER_DUMP_DEST parameter, 1101, 1102
USERENV namespace, 579, 580
USERID parameter, SQL*Loader, 633, 718
user-managed backups, 790–795
user-managed control file recovery, 826–828
user-managed copies, RMAN, 752
user-managed datafile recovery, 819–820
user-managed incomplete recovery, 824
user-managed recovery without backup, 829
user-managed tablespace recovery, 818
user-managed whole database recovery,
816–817
USER_RECYCLEBIN view, 850
users
altering properties of user’s session, 262
altering users, 547
assigning users to consumer groups, 562
centralized user authorization, 602
changing another user’s password
temporarily, 620
changing user’s password, 547
configuring, Oracle Secure Backup, 788
controlling use of resources, 548
    COMPOSITE_LIMIT parameter, 549
creating Oracle software owner user, 408–409
creating users, 544–547
    assigning tablespace quotas, 545
    privileges, 545
    temporary tablespace groups, 234
current system users, 50
DBA views managing, 577
DBCA changing passwords for default users, 490–491
denying access to database, 548
dropping users, 547–548
enterprise user security, 603–611
granting privileges to users with UTL_FILE package, 257
granting role to another user, 576
identifying high CPU users, 1154
LICENSE_MAX_USERS parameter, 461
listing user information, 619
managing users, 544, 619, 620
password management function, 552
privileged users, 599
profiles see user profiles
program global area (PGA), 193
resource consumer groups, 557
revoking tablespace quotas, 546
saving user input in variable, 121
schema-independent users, 603
SHOW USER command, 117
specifying maximum number of, 461
unlimited tablespace usage rights, 546
users creating tablespaces, 546
users with most waits, 1170
verifying existence of nobody, 408
whoami command, 50
USERS tablespaces, 484
USER_TABLESPACES view, 238
USER_TICKS system usage statistic, 1181
USE_STORED_OUTLINES parameter, 1079
USING clause, RMAN commands, 747, 750, 751
USING clause, role authorization, 575
utilities, Oracle, 207–208
utl_dir directory, 257
UTL_FILE package, 256–259
    setting permissions, 613
    specifying directory for processing I/O, 454
UTL_FILE_DIR parameter, 257, 454, 456
utllockt.sql script, 353, 355
utlpwdmg.sql script, 552
UTL_RECOMP package, 439
utlrp.sql script, 432, 434, 439, 440
utlu111l.sql script, 429, 434, 435
utlu111s.sql script, 429, 434, 440
utlupset.sql script, 434
utlxplan.sql script, 1091, 1092, 1095
UTL_XYZ packages, 615
V
V$ tables
    using V$ tables for wait information, 1165–1166
V$ views
see also dynamic performance views
VSACTIVE_SESSION_HISTORY, 971, 972, 1169, 1170, 1171, 1186
VSADVISOR_PROGRESS, 978
VSALERT_TYPES, 958
VSARCHIVE, 1187
VSARCHIVED_LOG, 795, 813, 844
VSASM_DISK, 909
VSBACKUP, 795
VSBACKUP_CORRUPTION, 782, 864
VSBACKUP_DATAFILE, 779
VSBACKUP_FILES, 760, 780
VSBACKUP_SET, 744
VSBGPROCESS, 971
VSBLOCK_CHANGE_TRACKING, 779
VSBUFFER_POOL_STATISTICS view, 1145
VSCONTROLFILE, 175
VSCOPY_CORRUPTION, 782, 864
VSCPOOL_CC_STATS, 533
VSCPOOL_STAT, 533
VSDATABASE, 507, 854
VSDATABASE_BLOCK_CORRUPTION, 865
VSDATAFILE, 246, 795, 812, 900, 993
VSDBCACHE_ADVICE, 189, 1145
VSDIAG_INFO, 1023, 1026
VSENCRYPTED_TABLESPACES, 243
VSENQUEUE_STAT, 1179
VSEVENT_NAME, 1175
VSFILESTAT, 246, 1177
VSFFLASHBACK_DATABASE_XYZ views, 858
VSFLASH_RECOVERY_AREA_USAGE, 740
VSHM_CHECK, 1032
VSHM_XYZ views, 1033
VSENSTANCE, 507
VSENSTANCE_RECOVERY, 805, 1205
VSLATCH, 1168
VSLIBRARYCACHE, 1137, 1138
VSLIBRARY_CACHE_MEMORY, 1138
VSLock_XYZ views, 354
VSLOG, 795, 985
VSLOGFILE, 900, 984
VSLOG_HISTORY, 795, 823
VSPGMNR_CONTENTS, 842, 845, 847
VSMAP_XYZ views, 994
VSMEMORY_XYZ views, 896
VSMETRICNAME, 958
VSOBJECT_USAGE, 305, 1073
VSOSTAT, 1181
VSPARAMETER, 473, 494
VSPGASTAT, 1149, 1150
VSPGA_TARGET_ADVICE, 1149
VSPROCESS, 1152
VSPROCESS_MEMORY, 1151
VSPWFILE_USERS, 600
<table>
<thead>
<tr>
<th>SQL Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V$RECOVERY_FILE_DEST</td>
<td>740</td>
</tr>
<tr>
<td>V$RECOVERY_LOG</td>
<td>813</td>
</tr>
<tr>
<td>V$RESTORE_POINT</td>
<td>863</td>
</tr>
<tr>
<td>V$RESULT_CACHE_XYZ views</td>
<td>1123</td>
</tr>
<tr>
<td>V$RECOVERY_LOG</td>
<td>813</td>
</tr>
<tr>
<td>V$RESULT_CACHE_XYZ views</td>
<td>1123</td>
</tr>
<tr>
<td>V$ROLLSTAT</td>
<td>365</td>
</tr>
<tr>
<td>V$RSRC_CONSUMER_GROUP</td>
<td>566</td>
</tr>
<tr>
<td>V$SEGMENT_NAME</td>
<td>1173</td>
</tr>
<tr>
<td>V$SEGMENT_STATISTICS</td>
<td>1173, 1206</td>
</tr>
<tr>
<td>V$SESSION_LONGOPS</td>
<td>714, 783, 943</td>
</tr>
<tr>
<td>V$SESSION_WAIT</td>
<td>386, 960, 1163, 1165, 1169, 1171, 1176, 1195, 1203</td>
</tr>
<tr>
<td>V$SESSION_WAIT_CLASS</td>
<td>1172</td>
</tr>
<tr>
<td>V$SESSION_WAIT_HISTORY</td>
<td>1168</td>
</tr>
<tr>
<td>V$SESSION_EVENT</td>
<td>1162, 1165, 1203</td>
</tr>
<tr>
<td>V$SESSION_FIX_CONTROL</td>
<td>1036</td>
</tr>
<tr>
<td>V$SESSIONLONGOPS</td>
<td>714, 783, 943</td>
</tr>
<tr>
<td>V$SESSION_WAIT</td>
<td>386, 960, 1163, 1165, 1169, 1171, 1176, 1195, 1203</td>
</tr>
<tr>
<td>V$SYS_TIME_MODEL</td>
<td>880, 1206</td>
</tr>
<tr>
<td>V$TABLESPACE</td>
<td>238, 246, 812, 993</td>
</tr>
<tr>
<td>V$TRANSACTION</td>
<td>340, 346</td>
</tr>
<tr>
<td>V$UNDOSTAT</td>
<td>360, 365</td>
</tr>
<tr>
<td>V$WAITCLASSMETRIC</td>
<td>1173</td>
</tr>
<tr>
<td>V$WAITSTAT</td>
<td>1167</td>
</tr>
</tbody>
</table>

Validating objects online, SQL, 934
Validating operator, SQL/XML, 1249
Validation commands, RMAN, 761
Validation methods, data concurrency, 347
VALID_TABLE_LIST parameter, 978
Value-based security, 312
Values
  default values for columns, 270
  null value, 269
VALUES LESS THAN clause, CREATE TABLE, 281
VARCHAR2 data type, 1222
VARIABLE clause, 647
Variable record format, SQL*Loader, 631
Variables, SQL*Plus see SQL*Plus environment
Variables, UNIX, 53, 54
VARRAY type, 1240
VERIFY variable, SQL*Plus, 108
verify_function_11g function, 552
Versions, 394
  checking kernel version, 402
  checking operating system version, 402
  fixing bugs, 1131
Flashback Versions Query, 366, 369–372
Identifying changed row versions, 375
Locating product files, 397
Multiple names for same version, 395
Preinstallation checks, 401
Pre-Upgrade Information Tool, 428
Retaining behavior of previous versions, 463
Retrieving version of UNIX command, 49
Table versioning and workspaces, 387
Timing conversion to new version, 1131
Variable naming Oracle database version, 128
VERSIONS BETWEEN clause, 370, 372
VERSIONS BETWEEN TIMESTAMP clause, 874
Vi text editor, UNIX, 63–65
View privileges, 571
View resolution, 314
Views, 312–314
  see also data dictionary views
  avoiding improper use of, 1075
database security, 577
Getting part/full text of views, 332
Managing database objects, 329
Materialized views, 314–320, 1077
Vim text editor, UNIX, 65
Virtual columns, 270–271
Partitioning, 286
VIRTUAL keyword, 270
Virtual memory, 1158
Virtual private catalogs, RMAN, 772–774
Virtual private database see VPD
Virtual tables see views
Vmstat utility, UNIX, 82, 1153, 1181, 1186
VNC (Virtual Network Computing), 46
Volume identification sequence, 789
VPD (virtual private database), 578
column-level VPD, 585–586
wait classes, 1163, 1171
- analyzing instance performance, 1164
  breakdown of waits by, 1172
determining total waits and percentage waits by, 1164
- metric values of, 1173
time spent in each type, 1172
Wait Event History, Database Control, 1200
wait event views, 1163
wait events, 1163
- analysis-based performance approach, 1183
  ASH reports, 975
  buffer busy wait events, 1175–1177
  checkpoint completed wait event, 1177
collecting wait event information, 1174–1175
  complete listing of, 1175
database wait statistics, 1162
db file scattered read wait event, 1177
db file sequential read wait event, 1178
  DBMS_MONITOR package, 1175
  DBMS_SYSTEM package, 1175
direct path read/write wait events, 1178
eliminating contention, 1208
enqueue waits event, 1179
event 10046 tracing SQL code, 1174
free buffer waits event, 1178
idle wait events, 1181
ignoring the unimportant, 1183
instance-wide wait event status, 1167
latch free wait events, 1179–1180
log buffer space wait event, 1180
log file switch wait event, 1180
log file sync wait event, 1181
most important recent wait events, 1170
Oracle wait events, 1175–1181
oradebug utility performing trace, 1174
system performance, 1206
V$SESSION_WAIT view, 1195
V$SESSION_WAIT_HISTORY view, 1168
wait information
- analyzing with Active Session History, 1169
  key dynamic performance tables
    showing, 1165
obtaining, 1167–1168
  using V$ tables for, 1165–1166
WAIT option
  committing transactions, 339
  MODE parameter, LOCK TABLE, 351
wait statistics
- data collected by AWR, 960
dynamic performance views containing, 1163
identifying SQL with highest waits, 1171
measuring instance performance, 1162–1163
not relying on, 1182
objects with highest waits, 1170
segment-level statistics, 1173
users with most waits, 1170
managing, 1015–1016
overlapping windows, 1015
precedence for, 1017
predefined maintenance windows, 1020
prioritizing jobs, 1016–1017
WITH ADMIN OPTION clause
  granting roles, 570, 576
  granting system privileges, 569
WITH READ ONLY clause, 312
WMSVERSIONED_TABLES table, 387
WMSVERSION_TABLE table, 387
WORD_WRAPPED option, FORMAT clause, 109
WORKAREA_SIZE_POLICY parameter, 194
worker process, Data Pump, 686
Workload Source page, SQL Access Advisor, 322
workloads
  analyzing after-upgrade SQL workload, 1219
  analyzing prechange SQL workload, 1218
  Automatic Workload Repository, 210, 959–971
  capturing production workload, 1210, 1211, 1217
  collecting operating system statistics, 1061
  Database Replay, 1214, 1215–1216
  DBMS_WORKLOAD_CAPTURE package, 1210
  preprocessing workload, 1211
  replaying captured workload, 1211
Workspace Manager, 386–388
workspaces, 386, 387, 388
WRAPPED option, FORMAT clause, 109
write-ahead protocol, 182, 199, 340
write permission, UNIX files, 59
write-allowed period, Oracle Secure Backup, 789
writer processes, specifying number of, 455

X
X Window emulation, 46, 414
X$ tables, 204
XDBURIType data type, 1249
xhost program, 412, 416
xids parameter, TRANSACTION_BACKOUT, 869
XML (Extensible Markup Language)
  audit parameters, 450
  semistructured database model, 41
  SQL*Loader utility, 645
  SQL/XML operators, 1249
  viewing XML stored in Oracle table, 1250
XML and SQL, 261
XML DB, Oracle, 1248–1252
XML documents
  creating relational view from, 1252
  inserting into Oracle table, 1250
XML schema
  setting up XML schema, 1251
  user-defined data types, 264
XML value, AUDIT_TRAIL parameter, 587
XMLType data type, 1249