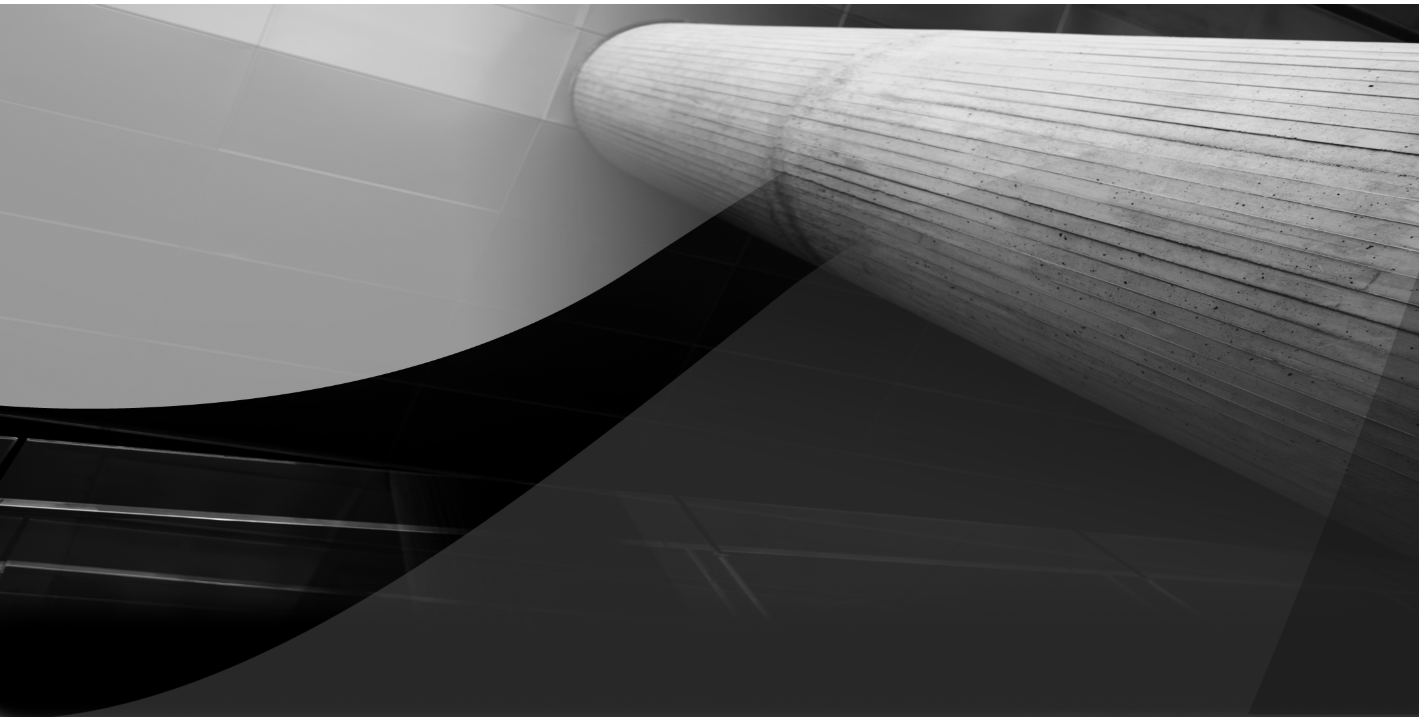

CHAPTER 8

Query Tuning: Developer and Beginner DBA

TIPS & COVERED

This chapter focuses on specific queries that you may encounter and some general information for tuning those specific queries, but it has also been updated to include some basic information on Oracle's 11*g* Automatic SQL Tuning and some queries to access Oracle's 11*g* Automatic Workload Repository (AWR). Examples of query tuning are spread throughout this book as well as instructions on making them more effective in terms of your system's architecture. This chapter centers on some of the most common queries that can be tuned on *most* systems. A query can display several variations in behavior, depending on system architecture, the data distribution in the tables, what tool or application is accessing the database, the specific version of Oracle Database, and a variety of other exceptions to the rules. Your results will vary; use your own testing to come up with the most favorable performance. The goal in this chapter is to show you many of the issues to watch for and how to fix them.

This chapter uses strictly cost-based examples for timings (except where noted). No other queries were performed at the time of the tests performed for this chapter. Many hints are also used throughout this chapter. For a detailed look at hints and the syntax and structure of hints, please refer to Chapter 7. Multiple table and complex queries are the focus of the next chapter and are not covered here.

Please note that this is not an all-inclusive chapter. Many other queries are covered throughout the book, which need to be investigated when trying to increase performance for a given query. Some of the most dramatic include using the parallel features of Oracle Database (Chapter 11), using partitioned tables and indexes (Chapter 2), and using PL/SQL to improve performance (Chapter 10). Note the benefits of using EXPLAIN and TRACE for queries (Chapter 6). Oracle Database 11*g* provides the Automatic Workload Repository (AWR) and Automatic Database Diagnostic Monitor (ADDM). The Enterprise Manager views of these new features are shown in Chapter 5. Tips covered in this chapter include the following:

- What queries do I tune? Querying the V$SQLAREA and V$SQL views
- Some useful new 11*g* views for locating resource-intensive sessions and queries
- When should I use an index?
- What if I forget the index?
- Creating and checking an index
- What if I create a bad index?
- Exercising caution when dropping an index
- Using invisible indexes
- Function based indexes and virtual columns
- Increasing performance by indexing the SELECT and WHERE columns
- Using the Fast Full Scan feature to guarantee success
- Making queries "magically" faster
- Caching a table into memory
- Using the new 11*g* Result Cache

- Choosing between multiple indexes on a table (use the most selective)
- Indexes that can get suppressed
- Tuning OR Clauses
- Using the EXISTS clause and the nested subquery
- That table is a view!
- SQL and the Grand Unified Theory
- Automatic SQL Tuning and the SQL Tuning Advisor
- Using the SQL Performance Analyzer (SPA)

What Queries Do I Tune? Querying V$SQLAREA and V$SQL Views

V$SQLAREA and V$SQL are great views that you can query to find the worst-performing SQL statements that need to be optimized. The value in the DISK_READS column signifies the volume of disk reads that are being performed on the system. This, combined with the executions (DISK_READS/EXECUTIONS), return the SQL statements that have the most disk hits per statement execution. Any statement that makes the top of this list is most likely a problem query that needs to be tuned. The AWR Report or Statspack Report also lists the resource-intensive queries; see Chapter 14 for detailed information.

Selecting from the V$SQLAREA View to Find the Worst Queries

The following query can be used to find the worst queries in your database. This query alone is worth the price of this book if you've not heard of V$SQLAREA yet.

To find the worst queries:

```
select      b.username username, a.disk_reads reads,
              a.executions exec, a.disk_reads /decode
             (a.executions, 0, 1,a.executions) rds_exec_ratio,
              a.sql_text Statement
from        V$sqlarea a, dba_users b
where       a.parsing_user_id = b.user_id
and         a.disk_reads > 100000
order       by a.disk_reads desc;

USERNAME     READS   EXEC  RDS_EXEC_RATIO STATEMENT
--------   ------- ----- --------------- ---------------------
---------------------------------------------------
ADHOC1    7281934     1         7281934  select custno, ordno
from cust, orders
```

```
ADHOC5    4230044     4         1057511  select ordno
from orders where trunc(ordno) = 721305

ADHOC1     801716     2          400858  select custno,
ordno from cust where substr(custno,1,6) = '314159'
```

The DISK_READS column in the preceding statement can be replaced with the BUFFER_GETS column to provide information on SQL statements requiring the largest amount of memory.

Now consider the output in a second example where there is a count of a billion-row table (EMP3) and a count of what was originally a 130M row table (EMP2), where all of the rows in EMP2, except the first 15 rows inserted, were deleted. Note that Oracle counts all the way up to the high water mark (HWM) of EMP2 (it read over 800,000, 8K blocks even though *all* of the data was only in 1 block). This listing would have told you something is wrong with the query on EMP2 that needs to be addressed, given that it only has 15 rows in it (analyzing the table will not improve this).

```
USERNAME    READS  EXEC  RDS_EXEC_RATIO STATEMENT
--------  ------- ----- --------------- -------------------------
SCOTT     5875532     1         5875532 select count(*) from emp3
SCOTT      800065     1          800065 select count(*) from emp2
```

For this issue, if the EMP2 table was *completely* empty, you could simply truncate the table to fix it. Since the table still has 15 rows, you have a few options; *which* option you choose depends on your unique situation. I can

- EXPORT/TRUNCATE/IMPORT; CREATE TABLE emp2b AS SELECT * FROM emp2 (CTAS) and then DROP and RENAME (I have to worry about indexes/related objects, etc.)
- Do an "ALTER TABLE *emp2* MOVE TABLESPACE *new1*" and rebuild the indexes.
- If it has a primary key, use DBMS_REDEFINITION.CAN_REDEF_TABLE to verify that the table can be redefined online.

Please check the Oracle documentation for syntax/advantages/disadvantages and stipulations (not all are listed here) for each of these options, so you can apply the best option to your situation (each of these options have major downsides, including users not being able to access the table and related objects getting dropped depending on which you use, so *be careful*). Once I reorganize the table, the next `count(*)` only reads 1 block instead of 800,065 blocks (it was well worth fixing the problem). Note in the query, I change "emp2" to emP2" so I can find that cursor in the cache.

```
alter table emp2 move;   -- You can specify a tablespace

select count(*)
from   emP2;

select     b.username username, a.disk_reads reads,
           a.executions exec, a.disk_reads /decode
          (a.executions, 0, 1,a.executions) rds_exec_ratio,
           a.sql_text Statement
```

```
from       V$sqlarea a, dba_users b
where      a.parsing_user_id = b.user_id
and        a.sql_text like '%emP2%'
order      by a.disk_reads desc;

USERNAME    READS  EXEC  RDS_EXEC_RATIO  STATEMENT
--------  ------- ----- --------------- ---------------------
SCOTT           1     1               1 select count(*) from emP2
```

You can also shrink space in a table, index-organized table, index, partition, subpartition, materialized view, or materialized view log. You do this using ALTER TABLE, ALTER INDEX, ALTER MATERIALIZED VIEW, or ALTER MATERIALIZED VIEW LOG statement with the SHRINK SPACE clause. See the *Oracle Administrators Guide* for additional information. Lastly, if you want to use the "ALTER TABLE *table* MOVE TABLESPACE *tablespace_name*" command, consider using the same size tablespace (or smaller if appropriate) to move things "back and forth" so as not to waste space.

TIP
Query V$SQLAREA to find your problem queries that need to be tuned.

Selecting from the V$SQL View to Find the Worst Queries

Querying V$SQL allows you to see the shared SQL area statements individually versus grouped together (as V$SQLAREA does). Here is a faster query to get the top statements from V$SQL (this query can also access V$SQLAREA by only changing the view name):

```
select *
from   (select address,
         rank() over ( order by buffer_gets desc ) as rank_bufgets,
         to_char(100 * ratio_to_report(buffer_gets) over (), '999.99') pct_bufgets
from   v$sql )
where  rank_bufgets < 11;

ADDRESS  RANK_BUFGETS PCT_BUF
-------- ------------ -------
131B7914            1   66.36
131ADA6C            2   24.57
131BC16C            3    1.97
13359B54            4     .98
1329ED20            5     .71
132C7374            5     .71
12E966B4            7     .52
131A3CDC            8     .48
131947C4            9     .48
1335BE14           10     .48
1335CE44           10     .48
```

You can alternatively select SQL_TEXT instead of ADDRESS if you want to see the SQL:

```
COL SQL_TEXT FOR A50
select *
from  (select sql_text,
        rank() over ( order by buffer_gets desc ) as rank_bufgets,
        to_char(100 * ratio_to_report(buffer_gets) over (), '999.99')
pct_bufgets
from   v$sql )
where  rank_bufgets < 11;
```

TIP
You can also query V$SQL to find your problem queries that need to be tuned.

Oracle 11g Views for Locating Resource-Intensive Sessions and Queries

Oracle 11*g* provides many new views, giving you access to a wealth of information from the OS (operating system) and the Automatic Workload Repository (AWR). The AWR provides metric-based information, which is useful for monitoring and diagnosing performance issues. Metrics are a set of statistics for certain system attributes as defined by Oracle. Essentially, they are context-defined statistics that are collated into historical information within the AWR.

Accessing the AWR and ADDM information via Enterprise Manager is covered in Chapter 5 as well as in the Oracle documentation. In this section, I am only looking at pulling some specific information out of these views using SQL to locate queries that may need tuning.

Selecting from V$SESSMETRIC to Find Current Resource-Intensive Sessions

This query shows the sessions that are heaviest in physical reads, CPU usage, or logical reads over a defined interval (15 seconds, by default). You may want to adjust the thresholds as appropriate for your environment.

To find resource-intensive sessions:

```
Select TO_CHAR(m.end_time,'DD-MON-YYYY HH24:MI:SS') e_dttm,   -- Interval End Time
        m.intsize_csec/100 ints,    -- Interval size in sec
        s.username usr,
        m.session_id sid,
        m.session_serial_num ssn,
        ROUND(m.cpu) cpu100,        -- CPU usage 100th sec
        m.physical_reads prds,      -- Number of physical reads
        m.logical_reads lrds,       -- Number of logical reads
        m.pga_memory pga,           -- PGA size at end of interval
        m.hard_parses hp,
        m.soft_parses sp,
        m.physical_read_pct prp,
        m.logical_read_pct lrp,
        s.sql_id
from   v$sessmetric m, v$session s
where (m.physical_reads > 100
or     m.cpu > 100
or     m.logical_reads > 100)
and    m.session_id = s.sid
and    m.session_serial_num = s.serial#
order by m.physical_reads DESC, m.cpu DESC, m.logical_reads DESC;

E_DTTM               INTS USR SID SSN  CPU100  PRDS LRDS    PGA HP SP PRP
LRP         SQL_ID
-------------------- ---- --- --- ---- ------ ----- ---- ------ -- -- ---
---------- --------------
20-NOV-2010 00:11:07   15 RIC 146 1501   1758 41348    1 781908  0  0 100
.512820513 03ay719wdnqz1
```

Viewing Available AWR Snapshots

The next few queries access AWR snapshot information.

Query the DBA_HIST_SNAPSHOT view to find more information about specific AWR snapshots:

```
select snap_id,
       TO_CHAR(begin_interval_time,'DD-MON-YYYY HH24:MI:SS') b_dttm,
        TO_CHAR(end_interval_time,'DD-MON-YYYY HH24:MI:SS') e_dttm
from   dba_hist_snapshot
where  begin_interval_time > TRUNC(SYSDATE);


SNAP_ID  B_DTTM               E_DTTM
-------- -------------------- --------------------
     503 25-MAY-2011 00:00:35 25-MAY-2011 01:00:48
     504 25-MAY-2011 01:00:48 25-MAY-2011 02:00:00
     505 25-MAY-2011 02:00:00 25-MAY-2011 03:00:13
     506 25-MAY-2011 03:18:38 25-MAY-2011 04:00:54
     507 25-MAY-2011 04:00:54 25-MAY-2011 05:00:07
```

Selecting from the DBA_HIST_SQLSTAT View to Find the Worst Queries

SQL statements that have exceeded predefined thresholds are kept in the AWR for a predefined time (seven days, by default). You can query the DBA_HIST_SQLSTAT view to find the worst queries. The following is the equivalent statement to the V$SQLAREA query earlier in this chapter.

To query DBA_HIST_SQLSTAT view to find the worst queries:

```
select snap_id, disk_reads_delta reads_delta,
        executions_delta exec_delta, disk_reads_delta /decode
      (executions_delta, 0, 1,executions_delta) rds_exec_ratio,
       sql_id
from   dba_hist_sqlstat
where  disk_reads_delta > 100000
order  by disk_reads_delta desc;

SNAP_ID   READS_DELTA   EXEC_DELTA  RDS_EXEC_RATIO      SQL_ID
------- ------------- ------------ ---------------  -------------
38            9743276          0          9743276  03ay719wdnqz1
39            9566692          0          9566692  03ay719wdnqz1
37            7725091          1          7725091  03ay719wdnqz1
```

Note that in the output, the same SQL_ID appears in three different AWR snapshots. (In this case, it was executed during the first one and is still running). You could also choose to filter on other criteria, including cumulative or delta values for DISK_READS, BUFFER_GETS, ROWS_PROCESSED, CPU_TIME, ELAPSED_TIME, IOWAIT, CLWAIT (cluster wait), and so on. Run a DESC command of the view DBA_HIST_SQLSTAT to get a full list of its columns. This listing shows different SQL_IDs at the top of the list.

```
   SNAP_ID READS_DELTA EXEC_DELTA RDS_EXEC_RATIO SQL_ID
---------- ----------- ---------- -------------- -------------
       513     5875532          1        5875532 f6c6qfq28rtkv
       513      800065          1         800065 df28xa1n6rcur
```

Selecting Query Text from the DBA_HIST_SQLTEXT View

The query text for the offending queries shown in the previous two examples can be obtained from the DBA_HIST_SQLTEXT view with the following query:

To query DBA_HIST_SQLTEXT:

```
select command_type,sql_text
from   dba_hist_sqltext
where  sql_id='03ay719wdnqz1';

COMMAND_TYPE  SQL_TEXT
------------  ----------------------------
           3  select count(1) from t2, t2

select command_type,sql_text
from   dba_hist_sqltext
where  sql_id='f6c6qfq28rtkv';

COMMAND_TYPE  SQL_TEXT
------------  ---------------------------
           3  select count(*) from emp3
```

Selecting Query EXPLAIN PLAN from the DBA_HIST_SQL_PLAN View

The EXPLAIN PLAN for the offending SQL is also captured. You may view information about the execution plan through the DBA_HIST_SQL_PLAN view. If you want to display the EXPLAIN PLAN, the simplest way is to use the DBMS_XPLAN package with a statement such as this one:

```
select *
from   table(DBMS_XPLAN.DISPLAY_AWR('03ay719wdnqz1'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
SQL_ID 03ay719wdnqz1
--------------------
select count(1) from t2, t2

Plan hash value: 1163428054
--------------------------------------------------------------------------
| Id | Operation            | Name  | Rows | Cost (%CPU)| Time      |
--------------------------------------------------------------------------
| 0  | SELECT STATEMENT     |       |      | 10G(100)    |           |
| 1  | SORT AGGREGATE       |       |    1 |             |           |
| 2  | MERGE JOIN CARTESIAN |       | 6810G| 10G (2)     |999:59:59 |
| 3  | INDEX FAST FULL SCAN | T2_I1 | 2609K| 3996 (2)    | 00:00:48 |
| 4  | BUFFER SORT          |       | 2609K| 10G (2)     |999:59:59 |
| 5  | INDEX FAST FULL SCAN | T2_I1 | 2609K| 3994 (2)    | 00:00:48 |
--------------------------------------------------------------------------
```

As you can see, this particular query is a Cartesian join, which is normally not a valid table join (certainly not a good idea as it joins *every* row of one table with *every* row of another table) and can lead to the massive resource consumption. This query was used to show how to take advantage of some of the new 11*g* functionality for identifying and collecting information about poorly performing SQL. Here is the output for the query that was used earlier that queries the EMP3 table, which is over 1 billion rows (still fast at 5 minutes, even though it's 1B rows):

```
select *
from   table(DBMS_XPLAN.DISPLAY_AWR('f6c6qfq28rtkv'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------
SQL_ID f6c6qfq28rtkv
--------------------
select count(*) from emp3

Plan hash value: 1396384608
```

```
-----------------------------------------------------------------
| Id  | Operation         | Name | Rows   | Cost (%CPU)| Time     |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT |      |        |  1605K(100)|          |
|   1 |  SORT AGGREGATE  |      |     1 |            |          |
|   2 | TABLE ACCESS FULL| EMP3 |  1006M|  1605K  (1)| 05:21:10 |
----------------------------------------------------------------
```

When Should I Use an Index?

In Oracle version 5, many DBAs called the indexing rule the 80/20 Rule; you needed to use an index if less than 20 percent of the rows were being returned by a query. In version 7, this number was reduced to about 7 percent on average, and in versions 8*i* and 9*i*, the number was closer to 4 percent. In versions 10*g* and 11*g*, Oracle is better at retrieving the entire table, so the value continues to be in the 5 percent or less range, although it depends not only on the number of rows but also on how the blocks are distributed as well (see Chapter 2 for additional information). Figure 8-1 shows when an index should generally be used (in V5 and V6 for rule-based optimization and in V7, V8*i,* V9*i,* V10*g,* and V11*g* for cost-based optimization). However, based on the distribution of data, parallel queries or partitioning can be used and other factors need to be considered. In Chapter 9, you will see how to make this graph for your own queries. If the table has fewer than 1000 records (small tables), then the graph is also different. For small tables, Oracle's cost-based optimizer generally uses the index when only less than 1 percent of the table is queried. This graph shows you the progress in versions of Oracle. The lower the percentage of rows returned, the more likely you would use an index. This graph shows the speed of a full table scan becoming faster. Because of the many variables starting with Oracle 9*i,* the percentage could continue to decrease as the trend shows happening from V5 to V8*i,*

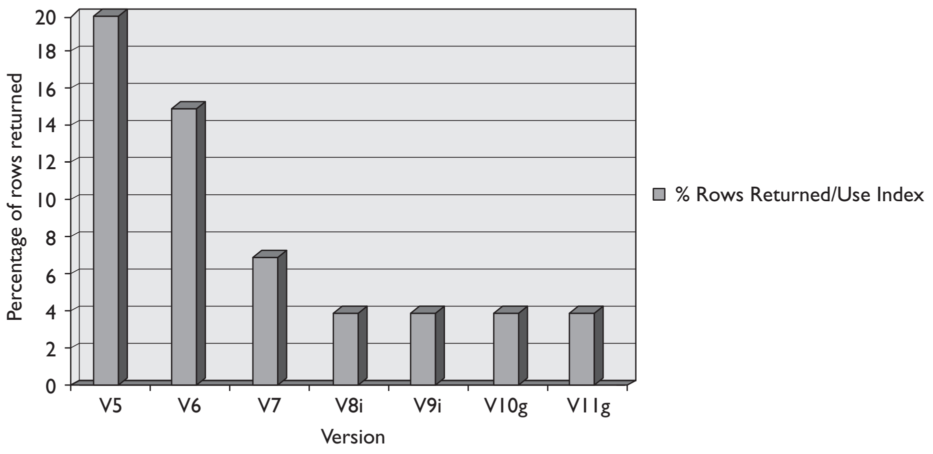

FIGURE 8-1. *When to* generally *use an index based on the percentage of rows returned by a query*

or it could increase slightly, depending on how you architect the database. In Oracle 9*i*, Oracle 10*g*, and Oracle11*g*, you create where the graph goes and Exadata and Exalogic enhancements can further alter this graph (where percentage could decrease to less than 1 percent); your choice may depend on how the data and indexes are architected, how the data is distributed within the blocks, and how it is accessed.

TIP
When a small number of rows ("small" is version and hardware dependent) are returned to meet a condition in a query, you generally want to use an index on that condition (column), given that the small number of rows also returns a small number of individual blocks (usually the case).

What If I Forget the Index?

Although it seems obvious that columns, which are generally restrictive, require indexes, this requirement is not always such common knowledge among users or managers. I once went to a consulting job where their database was suffering from incredibly poor performance. When I asked for a list of tables and indexes, they replied, "We have a list of tables, but we haven't figured out what indexes are yet and if we should use them or not—do you think you can help our performance?" My first thought was, "Wow, can I ever—my dream tuning job." My second thought was that I had been training experts too long and had forgotten that not everyone is as far along in their performance education. While basic index principles and structure are covered in Chapter 2, this section will focus on query-related issues surrounding indexes.

Even if you have built indexes correctly for most columns needing them, you may miss a crucial column here and there. If you forget to put an index on a restrictive column, then the speed of those queries will not be optimized. Consider the following example where the percent of rows returned by any given CUST_ID is less than 1 percent (there are about 25M rows on the SALES2 table and about 25K of them are CUST_ID=22340. Under these circumstances, an index on the CUST _ID column should normally be implemented. The next query does *not* have an index on CUST_ID:

```
select count(*)
from   sales2
where  cust_id = 22340;

  COUNT(*)
----------
    25750

Elapsed: 00:00:08.47 (8.47 seconds)

Execution Plan
----------------------------------------------------------
Plan hash value: 2862189843
```

```
----------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Cost (%CPU)| Time     |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |     1 | 32639   (1)| 00:06:32 |
|   1 |  SORT AGGREGATE    |        |     1 |            |          |
|   2 |   TABLE ACCESS FULL| SALES2 |    24M| 32639   (1)| 00:06:32 |
----------------------------------------------------------------------
119260 consistent gets (memory reads)
119258 physical reads (disk reads)
1,000 times more blocks read than using an index (we'll see this in a moment)
```

Not only is the query extremely slow, but it also uses a tremendous amount of memory and CPU to perform the query. This results in an impatient user and a frustrating wait for other users due to the lack of system resources. (Sound familiar?)

Creating an Index

To accelerate the query in the last example, I build an index on the CUST_ID column. The storage clause must be based on the size of the table and the column. The table is over 25 million rows (the space for the index is about 461M). Specifying automatic segment-space management for the underlying tablespace allows Oracle to manage segment space automatically for best performance. I could also perform an ALTER SESSION SET SORT_AREA_SIZE = 500000000 (if I had the necessary OS memory) and the index creation would be much faster.

```
create index sales2_idx1 on sales2(cust_id)
tablespace rich
storage (initial 400M next 10M pctincrease 0);

Index Created.
```

Invisible Index

Oracle 11*g* has a new feature called invisible indexes. An invisible index is invisible to the optimizer by default. Using this feature, you can test a new index without affecting the execution plans of the existing SQL statements or you can test the effect of dropping an index without actually dropping it (the index continues to be maintained even though it is not seen by the optimizer; this ensures if you make it visible again, it's up to date). Note that Chapter 2 has additional information and queries related to invisible indexes.

You can create an invisible index or you can alter an existing index to make it invisible. To enable the optimizer to use *all* invisible indexes (not a good idea usually), a new initialization parameter called OPTIMIZER_USE_INVISIBLE_INDEXES can be set to TRUE. This parameter is set to FALSE by default. You can run this CREATE instead of the one in the previous section:

```
create index sales2_idx1 on sales2(cust_id)
tablespace rich
storage (initial 400M next 10M pctincrease 0) invisible;

Index Created
```

Checking the Index on a Table

Before creating indexes, check for current indexes that exist on that table to ensure there are no conflicts.

Once you have created the index, verify that it exists by querying the DBA_IND_COLUMNS view:

```
select  table_name, index_name, column_name, column_position
from    dba_ind_columns
where   table_name = 'SALES2'
and     table_owner = 'SH'
order   by index_name, column_position;

TABLE_NAME INDEX_NAME_ COLUMN_NAME COLUMN_POSITION
---------- ----------- ----------- ---------------
SALES2     SALES2_IDX1 CUST_ID                   1
```

The TABLE_NAME is the table that is being indexed; the INDEX_NAME is the name of the index; the COLUMN_NAME is the column being indexed; and the COLUMN_POSITION is the order of the columns in a multipart index. Because our index involves only one column, the COLUMN_POSITION is '1' (CUST_ID is the first and only column in the index). In the concatenated index section (later in this chapter), you will see how a multipart index appears.

Query USER_INDEXES to verify the visibility of the index:

```
select index_name, visibility
from  user_indexes
where index_name = 'SALES2_IDX1';

INDEX_NAME              VISIBILITY
----------------------- ----------
SALES2_IDX1             VISIBLE
```

Is the Column Properly Indexed?

Rerun the same query now that the CUST_ID column is properly indexed. The query is dramatically faster, and more important, it no longer "floods" the system with a tremendous amount of data to the SGA (it has a much lower number of block reads) and subsequently reduces the physical I/O as well. Originally, this query took around 120,000 physical reads. Now it only takes about 60 physical reads (1000× less) and over 800× faster. Even though the query itself runs in seconds, this time difference can be a big deal if the query runs many times.

```
select count(*)
from   sales2
where  cust_id = 22340;

  COUNT(*)
----------
    25750
```

```
Elapsed: 00:00:00.01 (0.01 seconds - )

Execution Plan
----------------------------------------------------------
Plan hash value: 3721387097
-------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |     1 |     4 |    10   (0)| 00:00:01
|   1 |  SORT AGGREGATE    |             |     1 |     4 |            |
|*  2 |   INDEX RANGE SCAN| SALES2_IDX |  3514 | 14056 |    10   (0)| 00:00:01
-------------------------------------------------------------------------------

127 consistent gets (memory reads)
60 physical reads (disk reads)
```

TIP
The first tip concerning slow queries is that you'll have a lot of them if you don't index restrictive columns (return a small percentage of the table). Building indexes on restrictive columns is the first step toward better system performance.

What If I Create a Bad Index?

In the query to the PRODUCT table, I have a COMPANY_NO column. Since this company's expansion has not occurred, all rows in the table have a COMPANY_NO = 1. What if I am a beginner and I have heard that indexes are good and have decided to index the COMPANY_NO column? Consider the following example which selects only certain columns from the PLAN_TABLE after executing the query.

The cost-based optimizer will analyze the index as bad and suppress it. The table *must* be reanalyzed after the index is created for the cost-based optimizer to make an informed choice. The index created on COMPANY_NO is correctly suppressed by Oracle internally (since it would access the entire table and index):

```
select  product_id, qty
from    product
where   company_no = 1;

Elapsed time: 405 seconds (all records are retrieved via a full table scan)

OPERATION           OPTIONS         OBJECT NAME
------------------  --------------  -----------
SELECT STATEMENT
TABLE ACCESS        FULL            PRODUCT

49,825 consistent gets (memory reads)
41,562 physical reads (disk reads)
```

You can force an originally suppressed index to be used (bad choice), as follows:

```
select  /*+ index(product company_idx1) */ product_id, qty
from    product
where   company_no = 1;

Elapsed time: 725 seconds  (all records retrieved using the index on company_no)

OPERATION           OPTIONS         OBJECT NAME
------------------  --------------  -----------
SELECT STATEMENT
TABLE ACCESS        BY ROWID        PRODUCT
INDEX               RANGE SCAN      COMPANY_IDX1

4,626,725 consistent gets (memory reads)
80,513 physical reads (disk reads)
```

Indexes can also be suppressed when they cause poorer performance by using the FULL hint:

```
select  /*+ FULL(PRODUCT) */ product_id, qty
from    product
where   company_no = 1;

Elapsed time: 405 seconds (all records are retrieved via a full table scan)

OPERATION           OPTIONS         OBJECT NAME
------------------  --------------  -----------
SELECT STATEMENT
TABLE ACCESS           FULL         PRODUCT

49,825 consistent gets (memory reads)
41,562 physical reads (disk reads)
```

Next, consider a similar example in an 11*g*R2 database on a faster server with a 25M row table where I am summing *all* rows together. Oracle *is* once again smart enough to do a full table scan since I am summing the *entire* table. A full table scan only scans the table, but if I force an index (as in the second example), it has to read many more blocks (almost 50 percent more), scanning both the table and the index (resulting in a query that is almost four times slower).

```
select sum(prod_id)
from   sales
where  cust_id=1;

SUM(PROD_ID)

------------
  1939646817

Elapsed: 00:00:08.58
```

```
Execution Plan
-----------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |         |     1 |     7 | 33009   (2)| 00:06:37 |
|   1 |  SORT AGGREGATE    |         |     1 |     7 |            |          |
|*  2 |   TABLE ACCESS FULL| SALES3 |    24M|   165M| 33009   (2)| 00:06:37 |
-----------------------------------------------------------------------------

Statistic
----------------------------------------------------------
     119665  consistent gets
     119660  physical reads
```

Now let's try scanning the index and then go to the table (bad idea):

```
select /*+ index (sales3 sales3_idx) */ sum(prod_id)
from   sales

where  cust_id=1


SUM(PROD_ID)
------------
  1939646817

Elapsed: 00:00:33.9

Execution Plan
------------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  |Bytes| Cost (%CPU)|Time       |
|   0 | SELECT STATEMENT             |           |     1 |  7  | 213K (1)   |00:42:37  |
|   1 |  SORT AGGREGATE              |           |     1 |  7  |            |           |
|   2 |  TABLE ACCESS BY INDEX ROWID | SALES3    |    24M|165M | 213K (1)   |00:42:37  |
|*  3 |   INDEX RANGE SCAN           | SALES3_IDX|    24M|     |47976 (1)   |00:09:36  |
------------------------------------------------------------------------------------------

Statistic
----------------------------------------------------------
     168022  consistent gets
     168022  physical reads
```

TIP

Bad indexes (indexing the wrong columns) can cause as much trouble as forgetting to use indexes on the correct columns. While Oracle's cost-based optimizer generally suppresses poor indexes, problems can still develop when a bad index is used at the same time as a good index.

Exercising Caution When Dropping an Index

Some people's first reaction when they find a query that is using a poor index is to drop the index. Suppressing the index should be your first reaction, however, and investigating the impact of the index on other queries should be the next action. Unless your query was the only one being performed against the given table, changing/dropping an index might be a detrimental solution. The invisible index feature in 11*g* can be used to determine the effect of dropping an index without actually dropping it. Issue the following command against the index that needs to be dropped.

```
alter index sales2_idx1 invisible;
```

An invisible index is an index that continues to be maintained but is ignored by the optimizer unless you explicitly set it back to being visible or turn *all* invisible indexes on by setting the OPTIMIZER_USE_INVISIBLE_INDEXES to TRUE (careful). This way you can test the effect of dropping a particular index. If you want to reverse it, all you need to do is

```
alter index sales2_idx visible;
```

The next section investigates indexing columns that are both in the SELECT and WHERE clauses of the query.

Indexing the Columns Used in the SELECT and WHERE

The preceding section described how dropping an index can hurt performance for a query. Consider the following query where the index was created to help. I built a million-row EMPLOYEES table from the famous SCOTT.EMP table. This query does not have indexed columns:

```
select ename
from   employees
where  deptno = 10;

Elapsed time: 55 seconds (a full table scan is performed)

OPERATION            OPTIONS          OBJECT NAME
------------------   --------------   -----------
SELECT STATEMENT
TABLE ACCESS         FULL             EMPLOYEES
```

First, I place an index on the DEPTNO column to try to improve performance:

```
Create index dept_idx1 on employees (deptno)
Tablespace test1
Storage (initial 20M next 5M pctincrease 0);

select  ename
from    employees
where   deptno = 10;

Elapsed time: 70 seconds (the index on deptno is used but made things worse)
```

```
OPERATION            OPTIONS          OBJECT NAME
------------------   --------------   -----------
SELECT STATEMENT
TABLE ACCESS         BY INDEX ROWID    EMPLOYEES

INDEX                RANGE SCAN        DEPT_IDX1
```

This situation is now worse. In this query, only the ENAME is selected. If this is a crucial query on the system, choose to index both the SELECT and the WHERE columns. By doing this, you create a concatenated index:

```
Drop index dept_idx1;

Create index emp_idx1 on employees (deptno, ename)
Tablespace test1
Storage (initial 20M next 5M pctincrease 0);
```

The query is now tremendously faster:

```
select  ename
from    employees
where   deptno = 10;

Elapsed time: Less than 1 second (the index on deptno AND ename is used)

OPERATION            OPTIONS     OBJECT NAME
------------------   ----------  -----------
SELECT STATEMENT
INDEX                RANGE SCAN  EMP_IDX1
```

The table itself did not have to be accessed, which increases the speed of the query. Indexing both the column in the SELECT clause and the column in the WHERE clause allows the query to only access the index.

Consider the following 25M-row SALES3 table (created from SALES2). I have a two-part single index on the CUST_ID and PROD_ID columns. Oracle only needs to access the index (no table access), since all needed information is contained in the index (60K reads instead of the 160K you saw earlier).

```
select sum(prod_id)
from   sales3
where  cust_id=1;

SUM(PROD_ID)
------------
  1939646817

Elapsed: 00:00:05.4
```

```
Execution Plan
| Id  | Operation           | Name           | Rows  | Bytes |Cost  (%CPU|Time     |
------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT     |                |     1 |     7 |16690  (2)|00:03:21 |
|   1 |SORT AGGREGATE       |                |     1 |     7 |          |         |
|*  2 |INDEX FAST FULL SCAN|SALES_IDX_MULTI|    24M|   165M|16690  (2)|00:03:21 |

Statistics:
----------------------------------
60574  consistent gets
60556  physical reads
```

TIP
For crucial queries on your system, consider concatenated indexes on the columns contained in both the SELECT and the WHERE clauses so only the index is accessed.

Using the Fast Full Scan

The preceding section demonstrated that if I index both the SELECT and the WHERE columns, the query is much faster. Oracle does not guarantee that only the index will be used under these circumstances. However, there is a hint that guarantees (under most circumstances) that only the index will be used. The INDEX_FFS hint is a fast full scan of the index. This hint accesses only the index and not the corresponding table. Consider a query from a table with 100M rows with the index on CUST_ID called SALES2_IDX.

First, you check the number of blocks read for a full table scan and then a full index scan:

```
select /*+ full(sales2) */ count(*)
from   sales2;

  COUNT(*)
----------
 100153887

Elapsed: 00:01:42.63

Execution Plan
----------------------------------------------------------
Plan hash value: 2862189843
---------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Cost  (%CPU)| Time     |
---------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |     1 | 32761   (1)| 01:06:32 |
|   1 |  SORT AGGREGATE    |        |     1 |            |          |
|   2 |   TABLE ACCESS FULL| SALES2 |    24M| 32761   (1)| 01:06:32 |
---------------------------------------------------------------------
```

```
Statistics
----------------------------------------------------------
     820038  consistent gets
     481141  physical reads
```

Now let's try to select using a full index scan instead:

```
select /*+ index_ffs (sales2 sales2_idx) */ count(*)
from sales2;

  COUNT(*)
----------
 100153887

Elapsed: 00:24:06.07

Execution Plan
----------------------------------------------------------
Plan hash value: 3956822556
----------------------------------------------------------------------------
| Id  | Operation             | Name        | Rows   | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |             |      1 | 81419   (2)| 00:16:18 |
|   1 |  SORT AGGREGATE       |             |      1 |            |          |
|   2 |   INDEX FAST FULL SCAN| SALES2_IDX  |    24M| 81419   (2)| 00:16:18 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
     298091  consistent gets
     210835  physical reads
```

The query with the INDEX_FFS hint now only accesses the index. Instead of scanning over 800K blocks (of which 400K were physical reads), you only scan around 300K blocks (of which 210K are physical reads). Also note, sometimes your queries scan the *entire* index (as this one did), which is often not as good as if you have a limiting condition, so be careful; using an index search is much better than a full index scan when possible. Oracle often scans the index versus scanning the table for a `count(*)`, by default, in 11g. Running either of these queries a second time (see next section) does not get rid of the physical scans since the query retrieves enough data to fill half of the number of blocks as in the total buffer cache (it is pushed out of the cache quickly since it is not a short table; see Chapter 14 for additional details).

TIP
The INDEX_FFS (available since Oracle 8) processes only the index and does not access the table. All columns that are used and retrieved by the query must be contained in the index.

Making the Query "Magically" Faster

Consider the following query from the last example in which the user adds a hint called "RICHS_SECRET_HINT." The user overheard a conversation about this hint at a recent user group and believes this hint (buried deep in the X$ tables) is the hidden secret to tuning. First, the query is run and no index can be used (a large EMPLOYEES table with over 14M rows):

```
select  ename, job
from    employees
where   deptno = 10
and     ename = 'ADAMS';

Elapsed time: 45.8 seconds (one record is retrieved in this query)

OPERATION           OPTIONS     OBJECT NAME
------------------  ----------  -----------
SELECT STATEMENT
TABLE ACCESS        FULL        EMPLOYEES
```

There is *no* index that can be used on this query. A full table scan is performed.
The user now adds Rich's secret hint to the query:

```
select  /*+ richs_secret_hint */ ename, job
from    employees
where   deptno = 10
and     ename = 'ADAMS';

Elapsed time: under 1 second (one record is retrieved in this query)

OPERATION           OPTIONS     OBJECT NAME
------------------  ----------  -----------
SELECT STATEMENT
TABLE ACCESS        FULL        EMPLOYEES
```

The hint worked and the query is "magically" faster, although a full table scan was still performed in the second query. Actually, the data is now stored in memory and querying the data from memory is now much faster than going to disk for the data—so much for the magic! By effectively using the 11*g* Result Cache, you can *magically* make things faster as well. See the "Using the New 11*g* Result Cache" section later in this chapter (Chapters 1 and 4 also include additional information).

TIP
When running a query multiple times in succession, it becomes faster because you have now cached the data in memory (although full table scans are aged out of memory quicker than indexed scans). At times, people are tricked into believing that they have made a query faster, when in actuality they are accessing data stored in memory. Flushing the buffer cache or restarting the test system can help you get accurate tuning results for comparisons.

Caching a Table in Memory

While it is disappointing that there is no "secret hint" for tuning (ORDERED and LEADING are the hints closest to magic), you can use the last section to learn from, and then you can use this knowledge to your advantage. In the last section, the query ran faster the second time because it was cached in memory. What if the tables used most often were cached in memory all the time? Well, the first problem is that if you cannot cache every table in memory, you must focus on the smaller and more often used tables to be cached. You can also use multiple buffer pools as discussed in Chapter 4. The following query is run against an unindexed customer table to return one of the rows:

```
select prod_id, cust_id
from   sales
where  cust_id=999999999
and    prod_id is not null;

   PROD_ID    CUST_ID
---------- ----------
        13  999999999

Elapsed: 00:00:00.84

Execution Plan
----------------------------------------------------------
Plan hash value: 781590677
---------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost(%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    50 |  1300 |  1241  (2)| 00:00:15 |
|*  1 |  TABLE ACCESS FULL| SALES |    50 |  1300 |  1241  (2)| 00:00:15 |
---------------------------------------------------------------------------
```

The database is then stopped and restarted so as to not influence the timing statistics (you can also perform an "ALTER SYSTEM FLUSH BUFFER_CACHE" but only do this on a test system). The table is altered to cache the records:

```
alter table sales cache;

Table altered.
```

Query the unindexed, but now cached, SALES table and it still takes 0.84 seconds. The table has been altered to be cached, but the data is not in memory yet. Every subsequent query will now be faster (after the first one). I query the unindexed (but now cached) SALES table to return one of the rows in 0.04 seconds, or 21 times faster (this increase in speed could add up fast if this query is run thousands of times):

```
select prod_id, cust_id
from   sales
where  cust_id=999999999
and    prod_id is not null;
```

```
   PROD_ID    CUST_ID
---------- ----------
        13  999999999

Elapsed: 00:00:00.04

Execution Plan
----------------------------------------------------------
Plan hash value: 781590677
---------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost(%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |    50 |  1300 |  1241  (2)| 00:00:15 |
|*  1 |  TABLE ACCESS FULL| SALES |    50 |  1300 |  1241  (2)| 00:00:15 |
---------------------------------------------------------------------------
```

The query is faster because the table is now cached in memory; in fact, all queries to this table are now fast regardless of the condition used. A cached table is "pinned" into memory and placed at the "most recently used" end of the cache; it is pushed out of memory only after other full table scans to tables that are not cached are pushed out. Running a query multiple times places the data in memory so subsequent queries are faster—only caching a table ensures that the data is not later pushed out of memory. Oracle 11*g* caches frequently used data, by default, as you access things over and over.

TIP
Caching an often-used but relatively small table into memory ensures that the data is not pushed out of memory by other data. Be careful, however—cached tables can alter the execution path normally chosen by the optimizer, leading to an unexpected execution order for the query (for instance, affecting the driving table in nested loop joins).

Using the New 11g Result Cache

In Oracle 11*g*, a new feature called the Result Cache lets you cache SQL results in an area of the SGA to improve performance.

The following RESULT_CACHE hint caches the results on execution:

```
select /*+ result_cache */ SUM(sal)
from   scott.emp
where  deptno=20;
```

When a query with RESULT_CACHE hint is run, Oracle will see if the results of the query have already been executed, computed, and cached, and, if so, retrieve the data from the cache instead of querying the data blocks and computing the results again. Take the following important points into consideration before using this feature:

- The Result Cache feature is useful only for SQL queries that are executed over and over again frequently.
- The underlying data doesn't change very often. When the data changes, the result set is removed from the cache.

If you are executing the same queries over and over, using the RESULT_CACHE hint often makes subsequent queries run faster. Chapters 1 and 4 contain additional information on this.

TIP
If you are executing the same queries over and over (especially grouping or calculation functions), using the RESULT_CACHE hint often makes subsequent queries run faster.

Choosing Among Multiple Indexes (Use the Most Selective)

Having multiple indexes on a table can cause problems when you execute a query where the choices include using more than one of the indexes. The optimizer almost always chooses correctly. Consider the following example where the percent of rows returned by any given PRODUCT_ID is less than 1 percent where the data is equally distributed between the blocks. Under these circumstances, place an index on the PRODUCT_ID column. The following query has a single index on PRODUCT_ID:

```
select product_id, qty
from   product
where  company_no = 1
and    product_id = 167;

Elapsed time: 1 second (one record is retrieved; the index on product_id is used)

OPERATION            OPTIONS     OBJECT NAME
------------------   ----------  -----------
SELECT STATEMENT
TABLE ACCESS         BY ROWID    PRODUCT
INDEX                RANGE SCAN  PROD_IDX1

107 consistent gets (memory reads)
1 physical reads (disk reads)
```

Now create an additional index on the COMPANY_NO column. In this example, all of the records have a COMPANY_NO = 1, an extremely poor index. Rerun the query with both indexes (one on PRODUCT_ID and one on COMPANY_NO) existing on the table:

```
select product_id, qty
from   product
```

```
where  company_no = 1
and    product_id = 167;

Elapsed time: 725 seconds (one record is returned; a full table scan is performed)

OPERATION           OPTIONS     OBJECT NAME
------------------  ----------  -----------
SELECT STATEMENT
TABLE ACCESS        FULL        PRODUCT

4,626,725 consistent gets (memory reads)
80,513 physical reads (disk reads)
```

Oracle has chosen not to use either of the two indexes (perhaps because of a multiblock initialization parameter or some other "exception to the rule"), and the query performed a full table scan. Depending on the statistical data stored and version of Oracle used, I have seen this same query use the right index, the wrong index, no index at all, or a merge of both indexes. The correct choice is to force the use of the correct index. The correct index is the most restrictive. Rewrite the query to force the use of the most restrictive index, as follows, or better yet, fix the real initialization parameter issue (the less hints that you use, the better—especially when you upgrade to the next version of Oracle).

To rewrite the query to force the use of the most restrictive index:

```
select /*+ index(product prod_idx1) */ product_id, qty
from   product
where  company_no = 1
and    product_id = 167;

Elapsed time: 1 second (one record is retrieved)

OPERATION           OPTIONS     OBJECT NAME
------------------  ----------  -----------
SELECT STATEMENT
TABLE ACCESS        BY ROWID    PRODUCT
INDEX               RANGE SCAN  PROD_IDX1

107 consistent gets (memory reads)
1 physical reads (disk reads)
```

TIP
When multiple indexes on a single table can be used for a query, use the most restrictive index when you need to override an optimizer choice. While Oracle's cost-based optimizer generally forces the use of the most restrictive index, variations will occur, depending on the version of Oracle used, the structure of the query, and the initialization parameters that you may use. Fix the larger issue if you see this as a trend.

TIP
Bitmap indexes usually behave differently because they are usually much smaller. See Chapter 2 for more information on the differences between bitmap indexes and other indexes.

The Index Merge

Oracle's index merge feature allows you to merge two separate indexes and use the result of the indexes instead of going to the table from one of the indexes. Consider the following example (in 11*g*, if you use a rule-based hint, which Oracle does not support, Oracle includes a note *in the EXPLAIN PLAN* that specifically suggests you use the cost-based optimizer). Also note that OPTIMIZER_MODE set to CHOOSE is not supported either, so use either ALL_ROWS or FIRST_ROWS instead.

The following statistics are based on 1,000,000 records. The table is 210M.

```
create  index year_idx on test2 (year);
create  index state_idx on test2 (state);

select  /*+ rule index(test2) */ state, year
from    test2
where   year = '1972'
and     state = 'MA';

SELECT STATEMENT Optimizer=HINT: RULE
   TABLE ACCESS (BY INDEX ROWID) OF 'TEST2'
     INDEX (RANGE SCAN) OF 'STATE_IDX' (NON-UNIQUE)

Note
---------------------------------------------------
   - rule based optimizer used (consider using cbo)

Elapsed time: 23.50 seconds

select  /*+ index_join(test2 year_idx state_idx) */
state,  year
from    test2
where   year = '1972'
and     state = 'MA';

SELECT STATEMENT
   VIEW OF 'index$_join$_001'
      HASH JOIN
         INDEX (RANGE SCAN) OF 'YEAR_IDX' (NON-UNIQUE)
         INDEX (RANGE SCAN) OF 'STATE_IDX' (NON-UNIQUE)

Elapsed time: 4.76 seconds
```

In the first query, I test the speed of using just one of the indexes and then going back to the table (under certain scenarios, Oracle tunes this with an AND-EQUAL operation to access data from the indexes). I then use the INDEX_JOIN hint to force the merge of two separate indexes and use the result of the indexes instead of going back to the table. When the indexes are both small compared to the size of the table, this can lead to better performance. On a faster system, the second query takes only 0.06 seconds, so your mileage will vary.

Now, let's consider a query to the 25M row SALES3 table on a faster server with separate indexes on the CUST_ID and PROD_ID columns. Using an index merge of the two indexes yields a *very* slow response time and many blocks read (over 200K physical reads):

```
select /*+ index_join (sales3 sales3_idx sales3_idx2) */ sum(prod_id)
from   sales3
where  cust_id=1;

SUM(PROD_ID)
------------
  1939646817

Elapsed: 00:01:37.5

Execution Plan
------------------------------------------------------------------------------------
| Id  | Operation               | Name             |Rows| Bytes |Cost(%CPU)|Time    |
------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT         |                  |  1 |     7 |  158K (1)|00:31:47|
|   1 | SORT AGGREGATE          |                  |  1 |     7 |          |        |
|*  2 |   VIEW                  | index$_join$_001| 24M|   165M|  158K (1)|00:31:47|
|*  3 |   HASH JOIN             |                  |    |       |          |        |
|*  4 |   INDEX RANGE SCAN      | SALES3_IDX       | 24M|  165M |48211 (2) |00:09:39|
|   5 |    INDEX FAST FULL SCAN | SALES3_IDX2      | 24M|  165M |63038 (1) |00:12:37|
------------------------------------------------------------------------------------

Statistic
-----------------------
8536  consistent gets
217514  physical reads
```

If I drop the two indexes on SALES3 and replace them with a two-part single index on the CUST_ID and PROD_ID columns, performance improves greatly—over ten times faster. Another benefit is the reduction of physical block reads from over 200K to only 60K.

```
select sum(prod_id)
from   sales3
where  cust_id=1;SUM(PROD_ID)
------------
  1939646817

Execution Plan
-----------------------------
| Id  | Operation           | Name           |Rows | Bytes |Cost(%CPU)|Time     |
--------------------------------------------------------------------------------
|   0 |SELECT STATEMENT    |                |   1 |     7 |16690 (2) |00:03:21 |
|   1 |SORT AGGREGATE      |                |   1 |     7 |          |         |
|*  2 |INDEX FAST FULL SCAN|SALES_IDX_MULTI|  24M|  165M |16690(2)  |00:03:21 |
--------------------------------------------------------------------------------

Statistic
-----------------------
60574  consistent gets
60556  physical reads
```

Indexes That *Can* Get Suppressed

Building the perfect system with all of the correctly indexed columns does not guarantee successful system performance. With the prevalence in business of bright-eyed ad-hoc query users comes a variety of tuning challenges. One of the most common is the suppression of perfectly good indexes. A modification of the column side of a WHERE clause often results in that index being suppressed (unless function-based indexes are utilized or the super-smart optimizer figures out a better path). Alternative methods for writing the same query do exist that do not modify the indexed column. A couple of those examples are listed next. Oracle *does* use the indexes in many cases, internally fixing the suppression (they continue to get better at this from version to version), especially when an index search or a full index scan can be run instead of a full table scan. If you use 3GL code or code within applications, the results vary, so I continue to show these areas that are a problem with certain tools or applications for you to consider when you run into that full table scan that you didn't expect.

A math function is performed on the column:

```
select  product_id, qty
from    product
where   product_id+12 = 166;

Elapsed time: 405 second

OPERATION         OPTIONS  OBJECT NAME
----------------  -------  -----------
SELECT STATEMENT
TABLE ACCESS      FULL     PRODUCT
```

The math function is performed on the other side of the clause (Oracle often fixes this internally):

```
select  product_id, qty
from    product
where   product_id = 154;

Elapsed time: 1 second

OPERATION         OPTIONS     OBJECT NAME
----------------  -------  --------------
SELECT STATEMENT
TABLE ACCESS      BY ROWID    PRODUCT
      INDEX       RANGE SCAN  PROD_IDX1
```

A function is performed on the column:

```
select  product_id, qty
from    product
where  substr(product_id,1,1) = 1;

Elapsed time: 405 second
```

```
OPERATION         OPTIONS  OBJECT NAME
----------------  -------  -----------
SELECT STATEMENT
TABLE ACCESS      FULL     PRODUCT
```

The function is rewritten so the column is not altered (a LIKE or function-based index would fix this):

```
select  product_id, qty
from    product
where   product_id like '1%';

Elapsed time: 1 second

OPERATION         OPTIONS     OBJECT NAME
----------------  ----------  -----------
SELECT STATEMENT
TABLE ACCESS      BY ROWID    PRODUCT
INDEX             RANGE SCAN  PROD_IDX1
```

As I stated previously, Oracle is often smart enough to figure out the issue and still use the index. The following query shows that the index is scanned with no table access despite the attempt to suppress the index (adding zero (0) or using an NVL gave the same result). In the following case, everything needed is in the index. Oracle figures out the substring function on the leading edge of the index but is still able to use *only* the index despite needing both columns from the index (versus using the index to access back to the table).

```
select sum(prod_id)
from   sales3
where  substr(cust_id,1)=1;

SUM(PROD_ID)
------------
  1939646817

Elapsed: 00:00:12.49

Execution Plan
------------------------------------------------------------------------------
| Id  | Operation             | Name           |Rows |Bytes|Cost(%CPU| Time   |
|   0 |SELECT STATEMENT       |                |   1 |    7|17651(8) |00:03:32|
|   1 | SORT AGGREGATE        |                |   1 |    7|         |        |
|*  2 |  INDEX FAST FULL SCAN|SALES_IDX_MULTI| 248K|1695K|17651 (8)|00:03:32|
------------------------------------------------------------------------------
```

TIP
At times, modifying the column side of the query can result in the index being suppressed unless a function-based index is used. Oracle may *also fix this issue during parsing.*

Function-Based Indexes

One of the largest problems with indexes, as seen in the previous section, is that indexes are often suppressed by developers and ad-hoc users. Developers using functions often suppress indexes. There is a way to combat this problem. Function-based indexes allow you to create an index based on a function or expression. The value of the function or expression is specified by the person creating the index and is stored in the index. Function-based indexes can involve multiple columns, arithmetic expressions, or maybe a PL/SQL function or C callout.

The following example shows how to create a function-based index:

```
CREATE INDEX emp_idx ON emp (UPPER(ename));
```

An index that uses the UPPER function has been created on the ENAME column. The following example queries the EMP table using the function-based index:

```
select ename, job, deptno
from   emp
where  upper(ename) = 'ELLISON';
```

The function-based index (EMP_IDX) can be used for this query. For large tables where the condition retrieves a small amount of records, the query yields substantial performance gains over a full table scan. See Chapter 2 for additional details and examples.

The following initialization parameters must be set (subject to change with each version) to use function-based indexes (the optimization mode must be cost-based as well). When a function-based index is not working, this is often the problem.

```
query_rewrite_enabled = true
query_rewrite_integrity = trusted (or enforced)
```

TIP
Function-based indexes can lead to dramatic performance gains when used to create indexes on functions often used on selective columns in the WHERE clause.

To check the details for function-based indexes on a table, you may use a query similar to this:

```
select table_name, index_name, column_expression
from   dba_ind_expressions
where  table_name = 'SALES2'
and    table_owner = 'SH'
order  by index_name, column_position;
```

Virtual Columns

Oracle 11*g* has introduced a new feature called the virtual column, a column that allows you to define a function on other column(s) in the same table. Here is an example of creating a table with a virtual column:

```
CREATE TABLE my_employees (
    empId         NUMBER,
    firstName     VARCHAR2(30),
    lastName      VARCHAR2(30),
    salary        NUMBER(9,2),
    bonus         NUMBER GENERATED ALWAYS AS (ROUND(salary*(5/100)/12))
                  VIRTUAL,
   CONSTRAINT  myemp_pk PRIMARY KEY (empId));
```

An important point to remember is that indexes defined against virtual columns are equivalent to function-based indexes.

The "Curious" OR

The cost-based optimizer often has problems when the OR clause is used. The best way to think of the OR clause is as multiple queries that are then merged. Consider the following example where there is a single primary key on COL1, COL2, and COL3. Prior to Oracle 9*i*, the Oracle Database performed this query in the following way:

```
select *
from   table_test
where  pk_col1 = 'A'
and    pk_col2 in ('B', 'C')
and    pk_col3 = 'D';

2      Table Access By Rowid TABLE_TEST
1      Index Range Scan TAB_PK
```

NOTE
PK_COL2 and PK_COL3 were not used for index access.

Since Oracle 9*i,* Oracle HAS improved how the optimizer handles this query (internally performing an OR-expansion). In Oracle 11*g*, the optimizer uses the full primary key and concatenates the results (as shown next), which is much faster than using only part of the primary key (as in the preceding access path). Even though the access path for the preceding query looks better because there are fewer lines, don't be tricked; fewer lines in the EXPLAIN PLAN doesn't mean a more efficient query.

```
5 Concatenation
2  Table Access By Rowid TAB
1  Index Unique Scan TAB_PK
4  Table Access By Rowid TAB
3  Index Unique Scan TAB_PK
```

To get this desired result prior to 9*i*, you would have needed to break up the query as shown here (I show this since often making a query longer can make it faster, as it's processed differently):

```
select *
from    table_test
where  (pk_col1 = 'A'
and     pk_col2 = 'B'
and     pk_col3 = 'D')
or     (pk_col1 = 'A'
and     pk_col2 = 'C'
and     pk_col3 = 'D');

5  Concatenation
2  Table Access By Rowid TAB
1  Index Unique Scan TAB_PK
4  Table Access By Rowid TAB
3  Index Unique Scan TAB_PK
```

TIP
Oracle has improved the way that it performs the OR clause. The NO_EXPAND hint can still be helpful, as it prevents the optimizer from using OR expansion, as described in Chapter 7.

Using the EXISTS Function and the Nested Subquery

Another helpful tip to remember is to use the EXISTS function instead of the IN function in most circumstances. The EXISTS function checks to find a single matching row to return the result in a subquery. Because the IN function retrieves and checks all rows, it is slower. Oracle has also improved the optimizer so it often performs this optimization for you as well. Consider the following example, where the IN function leads to very poor performance. This query is faster only if the ITEMS table is extremely small:

```
select   product_id, qty
from     product
where    product_id = 167
and      item_no in
(select  item_no
 from     items);

Elapsed time: 25 minutes (The items table is 10 million rows)


OPERATION            OPTIONS     OBJECT NAME
------------------   ----------  -----------
SELECT STATEMENT
NESTED LOOPS SEMI
  TABLE ACCESS       BY ROWID    PRODUCT
    INDEX            RANGE SCAN  PROD_IDX1
    SORT
      TABLE ACCESS   FULL        ITEMS
```

In this query, the entire ITEMS table is retrieved.

This query is faster when the condition PRODUCT_ID = 167 substantially limits the outside query:

```
select    product_id, qty
from      product a
where     product_id = 167
and       exists
(select  'x'
 from     items b
 where    b.item_no = a.item_no);

Elapsed time: 2 seconds (The items table query search is limited to 3 rows)

OPERATION           OPTIONS     OBJECT NAME
------------------  ----------  -----------
SELECT STATEMENT
NESTED LOOPS SEMI
  TABLE ACCESS      BY ROWID    PRODUCT
    INDEX           RANGE SCAN  PROD_IDX1
    INDEX           RANGE SCAN  ITEM_IDX1
```

In this query, only the records retrieved in the outer query (from the PRODUCT table) are checked against the ITEMS table. This query can be substantially faster than the first query if the ITEM_NO in the ITEMS table is indexed or if the ITEMS table is very large, yet the items are limited by the condition PRODUCT_ID = 167 in the outer query.

TIP
*Using the nested subquery with an EXISTS clause may make queries dramatically faster, depending on the data being retrieved from each part of the query. Oracle11*g *often makes this translation internally, saving you time and giving you performance gains!*

That Table Is Actually a View!

Views can hide the complexity of SQL but they can also add to the complexity of optimization. When looking at a SELECT statement, unless you have instituted some kind of naming convention for views, you cannot tell if an object is a table or a view from the SELECT statement alone. You must examine the object in the database to tell the difference. Views can join multiple tables. Be careful about joining views or using a view designed for one purpose for a different purpose, or you may pay a heavy performance price. Ensure that all tables involved in the view are actually required by your query. Also keep in mind that different types of triggers can also hide performance issues behind a simple query. Good developer documentation can save a lot of time in finding performance issues in complex code.

SQL and Grand Unified Theory

Many physicists have searched for a single theory that explains all aspects of how the universe works. Many theories postulated have worked well in certain circumstances and break down in others. This is fine for theoretical physics, but it can spell disaster in a database. When writing

SQL, one should not attempt to write the "Grand Unified SQL" statement that will do all tasks, depending on the arguments passed to it. This typically results in suboptimal performance for most tasks performed by the statement (or you feel the effect during the next upgrade). It is better to write separate, highly efficient statements for each task that needs to be performed.

Tuning Changes in Oracle Database 11g

The general SQL tuning principles remain the same in 11*g*, but some significant optimizer changes should be noted.

- The RULE (and CHOOSE) OPTIMIZER_MODE has been deprecated and desupported in 11*g*. (The only way to get rule-based behavior in 11*g* is by using the RULE hint in a query, which is *not* supported either). In general, using the RULE hint is not recommended, but for individual queries that need it, it is there. Consult with Oracle support before using the RULE hint in 11*g*.
- In 11*g*, the cost-based optimizer has two modes: NORMAL and TUNING.
  - In NORMAL mode, the cost-based optimizer considers a very small subset of possible execution plans to determine which one to choose. The number of plans considered is far smaller than in past versions of the database in order to keep the time to generate the execution plan within strict limits. SQL profiles (statistical information) can be used to influence which plans are considered.
  - The TUNING mode of the cost-based optimizer can be used to perform more detailed analysis of SQL statements and make recommendations for actions to be taken and for auxiliary statistics to be accepted into a SQL profile for later use when running under NORMAL mode. TUNING mode is also known as the *Automatic Tuning Optimizer mode,* and the optimizer can take several minutes for a single statement (good for testing). See the *Oracle Database Performance Tuning Guide Automatic SQL Tuning* (Chapter 17 in the 11.2 docs).

Oracle states that the NORMAL mode should provide an acceptable execution path for most SQL statements. SQL statements that do not perform well in NORMAL mode may be tuned in TUNING mode for later use in NORMAL mode. This should provide a better performance balance for queries that have defined SQL profiles, with the majority of the optimizer work for complex queries being performed in TUNING mode once, rather than repeatedly, each time the SQL statement is parsed.

Oracle 11g Automatic SQL Tuning

Oracle Database 10*g* introduced the SQL Tuning Advisor to help DBAs and developers improve the performance of SQL statements. The Automatic SQL Tuning Advisor includes statistics analysis, SQL profiling, access path analysis, and SQL structure analysis, and can be performed through the SQL Tuning Advisor. The SQL Tuning Advisor uses input from the ADDM, from resource-intensive SQL statements captured by the AWR, from the cursor cache, or from SQL Tuning Sets. Oracle 11*g* has extended the SQL Tuning Advisor by adding additional features such as SQL Replay, Automatic SQL Tuning, SQL Statistics Management, and SQL Plan Management.

Since this chapter is focused on query tuning, I'll describe how to pass specific SQL to the SQL Tuning Advisor in the form of a SQL Tuning Set, and then I'll cover 11*g*'s Automatic SQL Tuning Advisor and SQL Performance Analysis (SQL Replay).The Oracle recommended interface for the SQL Tuning Advisor is Oracle Enterprise Manager (see Chapter 5), but you can use the APIs via the command line in SQL*Plus. I cover the command-line session so you can better understand the analysis procedure for a single query. This section is only a small glance into the functionality of the SQL Tuning Advisor. You also have the capability to create SQL Tuning Sets and SQL profiles as well as the ability to transport SQL Tuning Sets from one database to another.

Ensuring the Tuning User Has Access to the API

Access to these privileges should be restricted to authorized users in a production environment. The privileges are granted by SYS. The "ADMINISTER SQL TUNING SET" privilege allows a user to access only his or her own tuning sets.

```
GRANT ADMINISTER SQL TUNING SET to &TUNING_USER;  -- or
GRANT ADMINISTER ANY SQL TUNING SET to &TUNING_USER;
GRANT ADVISOR TO &TUNING_USER
GRANT CREATE ANY SQL PROFILE TO &TUNING_USER;
GRANT ALTER ANY SQL PROFILE TO &TUNING_USER;
GRANT DROP ANY SQL PROFILE TO &TUNING_USER;
```

Creating the Tuning Task

If you want to tune a single SQL statement, for example,

```
select COUNT(*)
from   t2
where  UPPER(owner) = 'RIC';
```

you must first create a tuning task using the DBMS_SQLTUNE package:

```
DECLARE
 tuning_task_name VARCHAR2(30);
 tuning_sqltext  CLOB;
BEGIN
 tuning_sqltext :=   'SELECT COUNT(*) '  ||
                     'FROM t2 '  ||
                     'WHERE UPPER(owner) = :owner';
 tuning_task_name := DBMS_SQLTUNE.CREATE_TUNING_TASK(
             sql_text  => tuning_sqltext,
            bind_list  => sql_binds(anydata.ConvertVarchar2(100)),
            user_name  => 'RIC',
              scope  => 'COMPREHENSIVE',
         time_limit  => 60,
          task_name  => 'first_tuning_task13',
         description => 'Tune T2 count');
END;
/
```

Making Sure the Task Can Be Seen in the Advisor Log

To see the task, query the USER_ADVISOR log:

```
select task_name
from   user_advisor_log;

TASK_NAME
-------------------
first_tuning_task13
```

Executing the SQL Tuning Task

To execute the tuning task, you use the DBMS_SQLTUNE package, as shown here:

```
BEGIN
DBMS_SQLTUNE.EXECUTE_TUNING_TASK( task_name => 'first_tuning_task13' );
END;
/
```

Checking Status of the Tuning Task

To see the specific tuning task, query the USER_ADVISOR log:

```
select status
from   user_advisor_tasks
where  task_name = 'first_tuning_task13';

STATUS
---------
COMPLETED
```

Displaying the SQL Tuning Advisor Report

To see the SQL Tuning Advisor Report, you also use the DBMS_SQLTUNE package:

```
SET LONG 8000
SET LONGCHUNKSIZE 8000
SET LINESIZE 100
SET PAGESIZE 100

select dbms_sqltune.report_tuning_task('first_tuning_task13')
from   dual;
```

Reviewing the Report Output

The report output shown next is lengthy, but it essentially recommends creating a function-based index on the owner column of table T2. Had the SQL Tuning Advisor recommended the use of a SQL profile, this could have been accepted by using the DBMS_SQLTUNE.ACCEPT_SQL_PROFILE package.

```
DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
----------------------------------------------------------------------------------
GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------
Tuning Task Name                    : first_tuning_task13
Tuning Task Owner                   : RIC
Workload Type                       :  Single SQL Statement
Scope                               : COMPREHENSIVE
Time Limit(seconds)                 : 60
Completion Status                   : COMPLETED
Started at                                : 11/20/2010 20:49:56
Completed at                              : 11/20/2010 20:49:56
Number of Index Findings  : 1
Number of SQL Restructure Findings: 1

DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
----------------------------------------------------------------------------------

----------------------------------------------------------------------------------
Schema Name: RIC
SQL ID     : 8ubrqzjkkyj3g
SQL Text   : SELECT COUNT(*) FROM t2 WHERE UPPER(owner) = 'RIC'
-------------------------------------------------------------------------------
FINDINGS SECTION (2 findings)
-------------------------------------------------------------------------------
1- Index Finding (see explain plans section below)

DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
----------------------------------------------------------------------------------

--------------------------------------------------------
The execution plan of this statement can be improved by creating one or more
indices.

Recommendation (estimated benefit: 100%)
----------------------------------------
Consider running the Access Advisor to improve the physical schema design
or creating the recommended index.
create index RIC.IDX$$_00CF0001 on RIC.T2(UPPER('OWNER'));

Rationale

DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
--------------------------------------------------------------------------------
   Creating the recommended indexes significantly improves the execution plan
of this statement. However, it might be preferable to run "Access Advisor"
```

```
using a representative SQL workload as opposed to a single statement. This
will allow Oracle to get comprehensive index recommendations which takes into
account index maintenance overhead and additional space consumption.

2- Restructure SQL finding (see plan 1 in explain plans section)
----------------------------------------------------------------
The predicate UPPER("T2"."OWNER")='RIC' used at line ID 2 of the execution
plan contains an expression on indexed column "OWNER". This expression

DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
---------------------------------------------------------------------------
prevents the optimizer from selecting indices on table "RIC"."T2".

Recommendation
--------------
- Rewrite the predicate into an equivalent form to take advantage of
indices. Alternatively, create a function-based index on the expression.

Rationale
---------
The optimizer is unable to use an index if the predicate is an inequality
condition or if there is an expression or an implicit data type conversion

DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
----------------------------------------------------------------------------------
on the indexed column.

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------
1- Original
-----------
Plan hash value: 1374435053
-------------------------------------------------------------------------------
DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
----------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes | Cost (%CPU) |Time      |
-------------------------------------------------------------------------------
|  0  | SELECT STATEMENT     |       |  1    |  6    |  4049  (3)  |00:00:49  |
|  1  |  SORT AGGREGATE      |       |  1    |  6    |             |          |
|* 2  |  INDEX FAST FULL SCAN| T2_I1 | 26097 |  152K |  4049  (3)  | 00:00:49 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
2 - filter(UPPER("OWNER")='RIC')

DBMS_SQLTUNE.REPORT_TUNING_TASK('FIRST_TUNING_TASK13')
----------------------------------------------------------------------------------

2- Using New Indices
--------------------
Plan hash value: 2206416184

---------------------------------------------------------------------------------
| Id | Operation         | Name             | Rows  | Bytes |Cost (%CPU)| Time    |
---------------------------------------------------------------------------------
|  0 | SELECT STATEMENT  |                  |  1    |  6    |524  (2)   | 00:00:07|
|  1 |  SORT AGGREGATE   |                  |  1    |  6    |
|* 2 |  INDEX RANGE SCAN | IDX$$_00CF0001 |  237K |  1390K|  524  (2) | 00:00:07|
```

Tuning SQL Statements Automatically Using SQL Tuning Advisor

Now let's look at Oracle 11*g*'s Automatic SQL Tuning Advisor. Oracle 11*g*'s Automatic SQL Tuning Advisor analyzes Automatic Workload Repository data to find high-load SQL statements that have been executed repeatedly. It then uses SQL Tuning Advisor to tune those statements, creates SQL profiles, if needed, and tests them thoroughly. If it thinks implementing the SQL profile is beneficial, it automatically implements them. No intervention is needed. Automatic SQL Tuning Advisor runs during the normal maintenance window. The DBA can then run reports against those recommendations and validate those SQL profiles.

Enabling Automatic SQL Tuning Advisor

The following procedure is used to enable Automatic SQL Tuning Advisor:

```
BEGIN
DBMS_AUTO_TASK_ADMIN.ENABLE( client_name => 'sql tuning advisor',  operation =>
   NULL,  window_name => NULL);
END;
/
```

Configuring Automatic SQL Tuning Advisor

To query what is currently set, run the following query:

```
select parameter_name, parameter_value
from   dba_advisor_parameters
where  task_name = 'SYS_AUTO_SQL_TUNING_TASK'
and    parameter_name IN ('ACCEPT_SQL_PROFILES',
                          'MAX_SQL_PROFILES_PER_EXEC',
                          'MAX_AUTO_SQL_PROFILES');

PARAMETER_NAME                 PARAMETER_VALUE
------------------------------ ------------------------------
ACCEPT_SQL_PROFILES            FALSE
MAX_SQL_PROFILES_PER_EXEC      20
MAX_AUTO_SQL_PROFILES          10000
```

Now change SQL_PROFILE parameters as follows:

```
SQL> CONNECT / AS SYSDBA

BEGIN
  DBMS_SQLTUNE.set_tuning_task_parameter(
  task_name => 'SYS_AUTO_SQL_TUNING_TASK',
  parameter => 'ACCEPT_SQL_PROFILES',
  value     => 'TRUE');
END;
/
```

The next step is to force the execution of the task so you see the results immediately:

```
exec dbms_sqltune.execute_tuning_task(task_name=>'SYS_AUTO_SQL_TUNING_TASK');
```

Viewing Automatic SQL Tuning Results

The following procedure reports the most recent run:

```
VARIABLE p_report CLOB;
BEGIN
  :p_report := DBMS_SQLTUNE.report_auto_tuning_task(
    begin_exec   => NULL,
    end_exec     => NULL,
    type         => DBMS_SQLTUNE.type_text,      -- 'TEXT'
    level        => DBMS_SQLTUNE.level_typical, -- 'TYPICAL'
    section      => DBMS_SQLTUNE.section_all,   -- 'ALL'
    object_id    => NULL,
    result_limit => NULL);
END;
```

Print :p_report prints the report and recommendation:

```
Set long 1000000
PRINT :p_report

GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------
Tuning Task Name                    : SYS_AUTO_SQL_TUNING_TASK
Tuning Task Owner                   : SYS
Workload Type                       : Automatic High-Load SQL Workload
Execution Count                     : 14
Current Execution                   : EXEC_1259
Execution Type                      : TUNE SQL
Scope                               : COMPREHENSIVE
Global Time Limit(seconds)          : 3600
Per-SQL Time Limit(seconds)         : 1200
Completion Status                   : COMPLETED
Started at                          : 02/03/2011 17:14:17
Completed at                        : 02/03/2011 17:14:27
Number of Candidate SQLs            : 3
Cumulative Elapsed Time of SQL (s)  : 50

-------------------------------------------------------------------------------
SUMMARY SECTION
-------------------------------------------------------------------------------
                    Global SQL Tuning Result Statistics
-------------------------------------------------------------------------------

Number of SQLs Analyzed                       : 3
Number of SQLs in the Report                  : 3
Number of SQLs with Findings                  : 3
Number of SQLs with Statistic Findings        : 3
```

```
-------------------------------------------------------------------------------
    SQLs with Findings Ordered by Maximum (Profile/Index) Benefit, Object ID
-------------------------------------------------------------------------------
object ID  SQL ID  statistics profile(benefit) index(benefit) restructure
---------- ------------- ---------- ---------------- -------------- -----------
      42 4q8yn4bnqw19s    1
      43 fvzwdtr0ywagd    1
      44 5sp4ugqbs4ms6    1

-------------------------------------------------------------------------------
    Objects with Missing/Stale Statistics (ordered by schema, object, type)
-------------------------------------------------------------------------------
Schema Name             Object Name          Type         State Cascade
---------------------------- ---------------------------- ----- ------- -------
                    SYS OBJECT_TAB           TABLE MISSING NO

--------------------------------------------------------------------------------
DETAILS SECTION
-------------------------------------------------------------------------------
 Statements with Results Ordered by Maximum (Profile/Index) Benefit, Object ID
-------------------------------------------------------------------------------
Object ID   : 42
Schema Name : SYS
SQL ID      : 4q8yn4bnqw19s
SQL Text    : insert into object_tab select * from object_tab

-------------------------------------------------------------------------------
FINDINGS SECTION (1 finding)

1- Statistics Finding
---------------------
  Table "SYS"."OBJECT_TAB" was not analyzed.
  Recommendation
  --------------
  - Consider collecting optimizer statistics for this table.
    execute dbms_stats.gather_table_stats(ownname => 'SYS', tabname =>
          'OBJECT_TAB', estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,
          method_opt => 'FOR ALL COLUMNS SIZE AUTO');

  Rationale
  ---------
    The optimizer requires up-to-date statistics for the table in order to
    select a good execution plan.

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------
1- Original
-----------
Plan hash value: 622691728

| Id  | Operation           | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | INSERT STATEMENT         |            |  4674K|   419M|  7687 (1)| 00:01:33 |
|   1 |  LOAD TABLE CONVENTIONAL | OBJECT_TAB |       |       |          |          |
|   2 |   TABLE ACCESS FULL      | OBJECT_TAB |  4674K|   419M|  7687 (1)| 00:01:33 |
---------------------------------------------------------------------------------------

-------------------------------------------------------------------------------
Object ID  : 43
Schema Name: SYS
SQL ID     : fvzwdtr0ywagd
SQL Text   : select count(*) from object_tab where UPPER(owner)='SYS'

-------------------------------------------------------------------------------
FINDINGS SECTION (1 finding)
-------------------------------------------------------------------------------
1- Statistics Finding
---------------------
  Table "SYS"."OBJECT_TAB" was not analyzed.

  Recommendation
  --------------
  - Consider collecting optimizer statistics for this table.
    execute dbms_stats.gather_table_stats(ownname => 'SYS', tabname =>
          'OBJECT_TAB', estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,
          method_opt => 'FOR ALL COLUMNS SIZE AUTO');

  Rationale
  ---------
    The optimizer requires up-to-date statistics for the table in order to
    select a good execution plan.

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------

1- Original
-----------
Plan hash value: 2592930531
--------------------------------------------------------------------------------
-
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     1 |    17 |  7703   (1)| 00:01:33 |
|   1 |  SORT AGGREGATE    |            |     1 |    17 |                |          |
|*  2 |   TABLE ACCESS FULL| OBJECT_TAB   |  2000K|    32M|  7703   (1)| 00:01:33 |
----------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(UPPER("OWNER")='SYS')

-------------------------------------------------------------------------------
Object ID  : 44
Schema Name: SYS
SQL ID     : 5sp4ugqbs4ms6
SQL Text   : select count(*) from object_tab where UPPER(owner)='SCOTT'

-------------------------------------------------------------------------------
FINDINGS SECTION (1 finding)
-------------------------------------------------------------------------------

1- Statistics Finding
---------------------
  Table "SYS"."OBJECT_TAB" was not analyzed.

  Recommendation
  --------------
  - Consider collecting optimizer statistics for this table.
    execute dbms_stats.gather_table_stats(ownname => 'SYS', tabname =>
          'OBJECT_TAB', estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,
          method_opt => 'FOR ALL COLUMNS SIZE AUTO');
  Rationale
  ---------
    The optimizer requires up-to-date statistics for the table in order to
    select a good execution plan.

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------
1- Original
-----------
Plan hash value: 2592930531
---------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     1 |    17 |  7703   (1)| 00:01:33 |
|   1 |  SORT AGGREGATE    |            |     1 |    17 |            |          |
|*  2 |   TABLE ACCESS FULL| OBJECT_TAB |   311 |  5287 |  7703   (1)| 00:01:33 |
---------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(UPPER("OWNER")='SCOTT')
```

Check the recommendation section. Tuning Advisor has recommended that you collect statistics. Just by running the following statement, you would improve the performance of the problem SQL listed in the SQL Tuning Advisor report:

```
execute dbms_stats.gather_table_stats(ownname => 'SYS', tabname => -
          'OBJECT_TAB', estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,  -
          method_opt => 'FOR ALL COLUMNS SIZE AUTO');
```

Using SQL Performance Analyzer (SPA)

The concept of SQL Tuning Sets and the SQL Tuning Advisor were introduced in 10*g* as described in earlier sections of this chapter. Oracle 11*g* makes use of SQL Tuning Sets with the SQL Performance Analyzer, often referred to as SPA. The SPA compares the performance of specific SQL statements defined in a particular SQL Tuning Set, before and after a database change. The database change could be a major upgrade from 10*g* to 11*g*, an initialization parameter change, or simply an index or statistics collection change. Chapter 5 covers how to do this in Enterprise Manager. Because this chapter focuses on SQL Tuning, let's see what the SQL Performance Analyzer can do with queries before and after creating an index. In Chapter 9, I cover more uses for SPA, especially in database and application upgrades, as well as Real Application Testing and Database Replay. SPA is a part of Real Application Testing and is not available in the database by default. The use of SQL Performance Analyzer (SPA) and Database Replay requires the Oracle Real Application Testing licensing option (from Oracle's Real Application Manual).

Step 1: Set Up the Testing Environment

For this test, a table is created called OBJECT_TAB, and the table is populated to simulate a decent workload:

```
create table object_tab as
 select *
 from   dba_objects;
```

```
insert into object_tab
 select *
 from   object_tab;

commit;
```

The OBJECT_TAB table does not have any indexes; statistics are collected (as displayed here):

```
exec dbms_stats.gather_table_stats(USER,'OBJECT_TAB',cascade=>TRUE);
```

Next, the shared pool is flushed to clear out SQL statements in memory to get a new workload:

```
alter system flush shared_pool;
```

Step 2: Execute the Queries

Execute the following testing queries:

```
select count(*)
from   object_tab
where  object_id=100;

select  count(*)
from    object_tab
where   object_id<100;

select count(*)
from   object_tab
where  object_id=1000;

select count(*)
from   object_tab
where  object_id<=1000;
```

Later on you'll create an index for the OBJECT_ID column and compare the performance of the SQL statement before and after the index is created.

Step 3: Create SQL Tuning Set

```
exec DBMS_SQLTUNE.create_sqlset(sqlset_name=>'sql_replay_test');
```

Step 4: Load SQL Tuning Set

The following procedure loads the Tuning Set by obtaining SQL statements from the CURSOR_CACHE that query the OBJECT_TAB table.

```
DECLARE
  l_cursor DBMS_SQLTUNE.sqlset_cursor;
BEGIN
   OPEN l_cursor FOR
   SELECT VALUE(a)
   FROM TABLE(DBMS_SQLTUNE.select_cursor_cache(
      basic_filter => 'sql_text LIKE ''%object_tab%'' and parsing_schema_name =
 ''SYS''',
      attribute_list => 'ALL')
   ) a;
DBMS_SQLTUNE.load_sqlset(sqlset_name => 'sql_replay_test',populate_cursor=>
 l_cursor);
END;
/
```

Step 5: Query from the SQL Tuning Set

```
select sql_text
from   dba_sqlset_statements
where  sqlset_name = 'sql_replay_test';

SQL_TEXT
--------------------------------------------------------------
Select count(*) from object_tab where object_id=100;
Select count(*) from object_tab where object_id<100;
Select count(*) from object_tab where object_id=1000;
Select count(*) from object_tab where object_id<=1000;
```

Step 6: Print from the SQL Tuning Set

```
VARIABLE v_task VARCHAR2(64);
EXEC :v_task :=
DBMS_SQLPA.create_analysis_task(sqlset_name=>'sql_replay_test');
print :v_task
V_TASK
-----------------------
TASK_832
```

Don’t forget to note this TASK ID (record it somewhere for later use).

Step 7: Execute Before Analysis Task

Execute the contents of the Tuning Set before the database change to gather performance information:

```
BEGIN
   DBMS_SQLPA.execute_analysis_task(task_name => :v_task,execution_type => 'test
 execute',
      execution_name => 'before_change');
END;
/
```

Step 8: Make the Necessary Changes

Add an index that you already know you need to improve performance of the queries in the Tuning Set (and regather statistics):

```
create index object_tab_indx_id on object_tab(object_id);
exec dbms_stats.gather_table_stats(USER,'OBJECT_TAB',cascade=>TRUE);
```

Step 9: Execute after Create Index Analysis Task

```
VARIABLE v_task VARCHAR2(64);
BEGIN
   DBMS_SQLPA.execute_analysis_task(task_name => 'TASK_832',execution_type =>
 'test execute',
      execution_name => 'after_change');
END;
/
```

Step 10: Execute Compare Analysis Task

```
VARIABLE v_task VARCHAR2(64);
--EXEC :v_task := DBMS_SQLPA.create_analysis_task(sqlset_name =>
'sql_replay_test');
BEGIN
   DBMS_SQLPA.execute_analysis_task( task_name   => 'TASK_832', execution_type =>
'compare performance',
     execution_params => dbms_advisor.arglist(
                    'execution_name1',
                    'before_change',
                    'execution_name2',
                     'after_change'));
END;
/
```

Step 11: Print the Final Analysis

```
SET LONG 100000000
SET PAGESIZE 0
SET LINESIZE 200
SET LONGCHUNKSIZE 200
SET TRIMSPOOL ON
spool /tmp/report.txt

SELECT DBMS_SQLPA.report_analysis_task('TASK_832')
from   dual;

spool off
```

Report Output

```
General Information
---------------------------------------------------------------------------------
---------
 Task Information:                         Workload Information:
-----------------------------------------  ---------------------------------------
-----
  Task Name    : TASK_832                  SQL Tuning Set Name         :
sql_replay_test
  Task Owner   : SYS                       SQL Tuning Set Owner        : SYS
  Description  :                           Total SQL Statement Count   : 7
```

```
Execution Information:
-------------------------------------------------------------------------------------
-----
  Execution Name     : EXEC_847
  Started            : 02/04/2010 15:57:00
  Execution Type     : COMPARE PERFORMANCE
  Last Updated       : 02/04/2010 15:57:00
  Description        :
  Global Time Limit  : UNLIMITED
  Scope              : COMPREHENSIVE
  Per-SQL Time Limit : UNUSED
  Status             : COMPLETED
  Number of Errors   : 0
  Number of Unsupported SQL  : 1

Analysis Information:
----------------------------------------------------------------------------------
---------
 Before Change Execution:                       After Change Execution:
 ---------------------------------------------  ----------------------------------
---------
  Execution Name      : before_change           Execution Name      : after_change
  Execution Type      : TEST EXECUTE            Execution Type      : TEST EXECUTE
  Scope               : COMPREHENSIVE           Scope               : COMPREHENSIVE
  Status              : COMPLETED               Status              : COMPLETED
  Started             : 02/04/2010 15:50:08     Started             : 02/04/2010 15:56:13
  Last Updated        : 02/04/2010 15:51:41     Last Updated        : 02/04/2010 15:56:15
  Global Time Limit   : UNLIMITED               Global Time Limit   : UNLIMITED
  Per-SQL Time Limit  : UNUSED                  Per-SQL Time Limit  : UNUSED
  Number of Errors    : 0                       Number of Errors    : 0

 ---------------------------------------------
 Comparison Metric: ELAPSED_TIME
 ---------------------------------------------
 Workload Impact Threshold: 1%
 ---------------------------------------------
 SQL Impact Threshold: 1%
 ---------------------------------------------

Report Summary
----------------------------------------------------------------------------------
---------

Projected Workload Change Impact:
-------------------------------------------
 Overall Impact      :  99.59%
 Improvement Impact  :  99.59%
 Regression Impact   :  0%

SQL Statement Count
-------------------------------------------
 SQL Category  SQL Count  Plan Change Count
 Overall               7                  4
 Improved              4                  4
 Unchanged             2                  0
 Unsupported           1                  0

Top 6 SQL Sorted by Absolute Value of Change Impact on the Workload
----------------------------------------------------------------------------------
---------
|           |                | Impact on | Execution | Metric   | Metric | Impact  | Plan  |
| object_id | sql_id         | Workload  | Frequency | Before   | After  | on SQL  | Change|
---------------------------------------------------------------------------------
----------
|        19 | 2suq4bp0p1s9p |     27.56% |        1 | 11598790 |     34 |    100% | y     |
|        21 | 7j70yfnjfxy9p |     25.02% |        1 | 10532117 |   2778 |  99.97% | y     |
|        22 | c8g33h1hn04xh |     24.28% |        1 | 10219529 |    370 |    100% | y     |
|        23 | g09jahhhn7ft3 |     22.72% |        1 |  9564149 |   1123 |  99.99% | y     |
|        18 | 033g69gb60ajp |      -.04% |        2 |    42989 |  50359 | -17.14% | n     |
|        24 | gz549qa95mvm0 |         0% |        2 |    41798 |  42682 |  -2.11% | n     |
----------------------------------------------------------------------------------
---------
Note: time statistics are displayed in microseconds
```

Wow! Overall impact is a positive 99.59 percent! Having viewed the queries, this makes sense. The query accessed a 1.1-million row table, yet only 16 rows satisfied the OBJECT_ID = 100 condition. By adding an index on the OBJECT_ID column, performance is drastically improved!

Tips Review

- Query V$SQLAREA and V$SQL to find problem queries that need to be tuned.
- When a small number of rows ("small" is version dependent) are to be returned based on a condition in a query, you generally want to use an index on that condition (column), given that the rows are not skewed within the individual blocks.
- The first tip concerning slow queries is that you will have a lot of them if you are missing indexes on columns that are generally restrictive. Building indexes on restrictive columns is the first step toward better system performance.
- Bad indexes (indexing the wrong columns) can cause as much trouble as forgetting to use indexes on the correct columns. While Oracle's cost-based optimizer generally suppresses poor indexes, problems can still develop when a bad index is used at the same time as a good index.
- For crucial queries on your system, consider concatenated indexes on the columns contained in both the SELECT and the WHERE clauses.
- The INDEX_FFS processes *only* the index and will not take the result and access the table. All columns that are used and retrieved by the query *must* be contained in the index. This method is a much better way to guarantee the index will be used.
- When a query is run multiple times in succession, it becomes faster since you have now cached the data in memory. At times, people are tricked into believing that they have actually made a query faster when, in actuality, they are accessing data stored in memory.
- Caching an often-used but relatively small table into memory ensures that the data is not pushed out of memory by other data. Also, be careful—cached tables can alter the execution path normally chosen by the optimizer, leading to an unexpected execution order for the query (it can affect the driving table in nested loop joins).
- Oracle 11*g* provides a CACHE_RESULT feature, which you can use to cache the result of a query, allowing subsequent queries to access the result set directly instead finding the result through aggregating data stored in the database block buffers a subsequent time.

- When multiple indexes on a single table can be used for a query, use the most restrictive index. While Oracle's cost-based optimizer generally forces use of the most restrictive index, variations occur, depending on the Oracle version and the query structure.
- Any modification to the column side of the query results in the suppression of the index unless a function-based index is created. Function-based indexes can lead to dramatic performance gains when used to create indexes on functions often used on selective columns in the WHERE clause.
- Oracle's optimizer now performs OR-expansion, which improves the performance of certain queries that ran poorly in prior versions.
- Using the nested subquery with an EXISTS clause may make queries dramatically faster, depending on the data being retrieved from each part of the query. Oracle11*g* often makes this translation internally, saving you time and giving you performance gains!

References

Deb Dudek, *DBA Tips, or a Job Is a Terrible Thing to Waste* (TUSC).
Rich Niemiec, *DBA Tuning Tips: Now YOU Are the Expert* (TUSC).
*Oracle® Database Performance Tuning Guide 11*g *Release 2 (11.2).*
Query Optimization in Oracle 9*i*, An Oracle Whitepaper (Oracle).

Rama Balaji added several new sections and helped with the update for 11*g*. Thanks to Connor McDonald for his feedback on V$SQLAREA. Rob Christensen contributed the major portion of the previous update to this chapter.