CHAPTER 18

Optimizers and Statistics

In This Chapter

Comparing cost-based and rule-based optimization

Generating statistics

Discovering chained rows

Importing and exporting statistics

Choosing the correct optimizer and maintaining current statistics are two of the keys to getting good performance out of SQL queries that you execute against your database. Oracle supports two different optimizers; you have to choose which to use. Oracle also implements functionality that you can use to collect information about the data in your tables and indexes so that the optimizer can make intelligent decisions about how best to retrieve data for a query.

Comparing Cost-based and Rule-based Optimization

When you submit a `SELECT` query to be executed against your database, Oracle must provide some sort of algorithm to retrieve the data that you are after from the database. The part of Oracle that does this is called the *optimizer*. The algorithm for retrieving your data is referred to as an *execution plan*. Oracle supports two optimizers: One is rule-based, and the other is cost-based.

Examining the rule-based optimizer

The optimizer that Oracle first developed is rule-based. It looks at your query, checks for certain constructs such as joins, the use of indexed columns in the `WHERE` clause, and so forth, and then uses that information to determine an access path to the data. Oracle's rule-based optimizer looks only at the syntax of your SQL statements and at whether fields are indexed. That's it. It makes no attempt to compare the amount of disk I/O and CPU required for different execution plans.

As good as the rule-based optimizer is, in many cases it won't choose the best execution plan for a query. In part, this is because it doesn't look at the amount of I/O or CPU needed to execute any given plan; in part, it's because the plan chosen depends on how the SELECT statement is written. Something as simple as changing the order of the tables listed in the FROM clause can have a disastrous effect on performance.

Examining the cost-based optimizer

The cost-based optimizer represents Oracle's vision for the future. Introduced with the release of Oracle7, the cost-based optimizer chooses its execution plan based on the estimated cost of the various alternative approaches. This *cost* is determined largely by the amount of disk I/O and CPU resources that a given execution plan is expected to consume. Getting good performance out of the rule-based optimizer depends on getting programmers to structure their queries correctly based on their knowledge of the data being queried. The cost-based optimizer uses information about your data, such as how many rows are in a table, how many unique values are in an index, and so forth, to make intelligent decisions regardless of how a query is written. The key to getting good performance out of the cost-based optimizer is to gather statistics for your tables and keep them current.

Actually, the cost-based optimizer isn't perfect either. It won't always choose the absolute best execution plan. Still, Oracle has improved it tremendously since its initial release.

Choosing the optimizer to use

You should use the cost-based optimizer. It's that simple. Oracle has not enhanced the rule-based optimizer at all since the cost-based optimizer was released. The cost-based optimizer is the future, and Oracle has clearly stated its intention to drop support for the rule-based optimizer at some as-yet-unspecified date. Only the cost-based optimizer contains support for relatively new features such as index-organized tables, star joins, and parallel execution. Oracle's future development efforts are all going into the cost-based optimizer.

When it first came out, many developers and database administrators were leery of using the cost-based optimizer because of the kinks in the early implementation. Because it was predictable, the rule-based optimizer offered a certain comfort level. The cost-based optimizer wasn't so predictable because the results are based on the characteristics of the data being queried, not the structure of the SQL statement. Database administrators and programmers often viewed this unpredictability negatively, and many chose to continue using the rule-based optimizer.

The only reason that you might consider using the rule-based optimizer now is if you are running an older application that was originally designed with that

optimizer in mind. If all your SQL statements are tuned specifically for the rule-based optimizer, you may not be able to abruptly switch. In such a case, you may want to set your database to use the rule-based optimizer. However, you should also look at migrating your application. Until you do that, you won't be able to take advantage of the new features supported by the cost-based optimizer, and someday, the rule-based optimizer may disappear.

Choosing the Default Optimizer

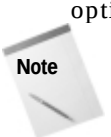
You choose the optimizer for a database by placing an OPTIMIZER_MODE entry in the database parameter file. If you want to go with the cost-based optimizer, that entry would look like this:

```
OPTIMIZER_MODE = CHOOSE
```

OPTIMIZER_MODE has four possible values, which are described in the following list:

CHOOSE	Enables cost-based optimization. Oracle will choose to use it if statistics are present.
RULE	Restricts Oracle to using the rule-based optimizer.
FIRST_ROWS	Enables cost-based optimization, with the goal of always returning the first few rows quickly, even if that leads to a higher overall cost.
ALL_ROWS	Enables cost-based optimization but with the goal of reducing overall cost. This is the same as CHOOSE.

With the optimizer mode set to CHOOSE, FIRST_ROW, or ALL_ROWS, the cost-based optimizer is used whenever statistics have been generated for at least one of the tables involved in the query. If no statistics have been gathered, no information exists for the cost-based optimizer to use, so Oracle falls back on the rule-based optimizer.



Whenever you change the OPTIMIZER_MODE parameter, you need to shut down and restart the database for that change to take effect.

The difference between FIRST_ROWS and ALL_ROWS is in the time it takes Oracle to begin returning rows from a query. If you are primarily running batch jobs, you will want to reduce overall I/O and CPU costs. In that case, specify ALL_ROWS or CHOOSE. The two are equivalent. If your queries are primarily generated by online users, you don't want them waiting and waiting for results to appear on their screens. Use FIRST_ROWS in this case to tell Oracle to optimize for quick response. When you use FIRST_ROWS, Oracle may choose a plan that takes longer overall but that gets a response back to the user quickly. Oracle may, for example, choose to use an index over a tablescan, even when the tablescan would take less I/O overall, because using the index allows a more immediate response.

Overriding the Optimizer Choice

You can override the optimizer choice both at the session level and at the statement level. To override the optimizer setting for the duration of your session, use the ALTER SESSION command, as shown in this example:

```
SQL> ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;

Session altered.
```

In this example, the statement was issued from SQL*Plus. You could also issue it from within a program. Changes you make with ALTER SESSION stay in effect until you disconnect from the database.

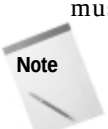
You can specify the optimizer mode at the individual statement level through the use of hints. For example, the following statement uses a hint to choose the cost-based optimizer and to specify that execution be optimized for overall throughput:

```
SELECT /*+ all_rows */ id_no, animal_name
FROM aquatic_animal
WHERE death_date IS NULL;
```

Hints are placed in SQL statements using a specific type of comment. The format for the comment looks like this:

```
/*+ hint */
```

The comment must begin with /*+, and it must end with */. Don't forget the plus sign (+). If you leave out the +, Oracle won't recognize the comment as a hint. Hints must also immediately follow the keyword beginning the statement.



Using hints can be tricky because if you get the syntax wrong, Oracle treats the hint as a comment. No error message is generated. To be sure that a hint is working correctly, you can use EXPLAIN PLAN to view Oracle's execution plan for the statement. Chapter 19, "Tuning SQL Statements," shows you how to do this.

You can use four hints to choose optimizer modes. The hints are named to match the parameter setting, which are CHOOSE, FIRST_ROWS, ALL_ROWS, and RULE.

Generating Statistics

For the cost-based optimizer to function, it must have information about your data. It must know details such as how many rows are in a table, how big your indexes are, and how closely the order of the data in a table matches the order in an index.

Collectively, all the pieces of information that the cost-based optimizer uses are known as *statistics*.

Collecting and maintaining statistics for the optimizer entails some overhead. That overhead is much more than can be supported during normal operations such as inserting and updating rows in a table. Consequently, Oracle leaves it to you to decide when to incur this overhead. You have to kick off statistics collection manually, using the `ANALYZE` command.

You should have some idea of the types of statistics that are generated to know how to best use the `ANALYZE` command. Different statistics are generated for different types of objects. This section explains the types of statistics that are collected for tables, indexes, and columns.

Looking at table statistics

Table 18-1 shows the statistics that Oracle generates for a table. You can access these statistics via columns in the `DBA_TABLES`, `ALL_TABLES`, and `USER_TABLES` data dictionary views. The column names corresponding to the statistic are also shown in Table 18-1.

Table 18-1
Table Statistics

Column Name	*Statistic*
`NUM_ROWS`	The number of rows in the table
`BLOCKS`	The number of data blocks currently being used for data
`EMPTY_BLOCKS`	The number of data blocks allocated to the table but that have never been used for data
`AVG_SPACE`	The average amount of free space, expressed in bytes, within each block
`CHAIN_CNT`	The number of chained rows
`AVG_ROW_LEN`	The average length of the rows in the table, in bytes
`LAST_ANALYZED`	The date on which the statistics for the table were generated

When you collect statistics for a table, unless you specify otherwise, Oracle also collects statistics for indexes on that table.

Looking at index statistics

Table 18-2 shows the statistics that Oracle generates for an index. Index statistics are returned by columns in the DBA_INDEXES, ALL_INDEXES, and USER_INDEXES data dictionary views.

Table 18-2
Index Statistics

Column Name	*Statistic*
BLEVEL	The depth of the index, or number of levels, from the index's root block to its leaf blocks
LEAF_BLOCKS	The number of leaf blocks, which are the blocks containing pointers to the rows in the table, in the index
DISTINCT_KEYS	The number of distinct index values
AVG_LEAF_BLOCKS_PER_KEY	The average number of leaf blocks containing entries for one value
AVG_DATA_BLOCKS_PER_KEY	The number of data blocks, on average, pointed to by one index value
CLUSTERING_FACTOR	A clustering factor, indicating how closely the order of rows in the table happens to match the ordering in the index
LAST_ANALYZED	The date on which the statistics for the index were generated

Looking at column statistics

Table 18-3 shows the statistics gathered for columns.

Table 18-3
Column Statistics

Column Name	*Statistic*
NUM_DISTINCT	The number of distinct values contained in the column
LOW_VALUE	The lowest value in the column, limited to the first 32 bytes of that value
HIGH_VALUE	The highest value in the column, limited to the first 32 bytes of that value

Column Name	*Statistic*
DENSITY	The column's density
NUM_NULLS	The number of rows that contain null values for the column
NUM_BUCKETS	The number of buckets in the column's histogram
LAST_ANALYZED	The date on which the statistics for the column were generated

Column statistics aren't always generated. By default, when you issue an ANALYZE TABLE command, only table and index statistics are generated. You have to specifically ask to generate column statistics.

The statistics shown in Table 18-3 aren't the only items generated when you generate column statistics. In addition to the items listed in the table, Oracle also generates histograms for the columns.

Displaying column histograms

Histograms are generated when you compute column statistics and indicate the distribution of data within a column. A histogram, if you aren't familiar with the term, is a type of graph. Figure 18-1 shows a histogram illustrating the distribution of tank numbers in the AQUATIC_ANIMAL table.

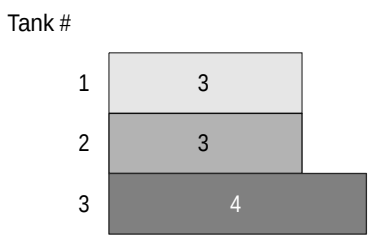


Figure 18-1: A histogram showing the distribution of tank numbers in the AQUATIC_ANIMAL table

The histogram shown in Figure 18-1 is called a width-balanced histogram, and it is the type most people are familiar with. However, it's not the type that Oracle uses. In a width-balanced histogram, the number of buckets is fixed and the length of the bars varies with the number of values that fall into each bucket. Oracle uses height-balanced histograms, where the length of the bars is fixed and the number of buckets is variable.

The concept of a height-balanced histogram can be difficult to grasp at first. Figure 18-1 represents a width-balanced histogram. There are three bars in the

graph representing three different tank values. The height of each bar corresponds to the number of rows in the AQUATIC_ANIMAL table with the given tank number value. Each bar represents a distinct value for the TANK_NO column. When width-balanced histograms are used, the height of each bar varies, but the number of values represented by each bar is a constant. Height-balanced histograms work using the opposite behavior.

When generating a height-balanced histogram, Oracle first determines how many buckets to use. You can control the number of buckets, but for now, assume that three buckets will be used. Oracle divides the number of rows in the table (10) by the number of buckets (3) to get a value for the number of rows to place in each bucket: in this case, 10 / 3 = 3. Finally, Oracle divides the ten rows into three buckets and records the highest value in each bucket. Figure 18-2 illustrates a height-balanced histogram based on the same data used for the width-balanced histogram in Figure 18-1.

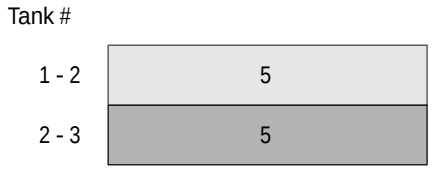


Figure 18-2: Oracle generates height-balanced histograms to record the distribution of values within a column.

Keep in mind that the number of buckets may vary, but that the number of rows represented by each bucket is the same. You can see the histograms that Oracle generates in the ALL_TAB_HISTOGRAMS data dictionary view, where each bucket is represented by one row in the view.

If the number of distinct values in a column is less than the number of buckets in the histogram that you create, Oracle actually creates a histogram more like the one shown in Figure 18-1, with one bucket for each distinct value. If you have more values than buckets, which is often the case, you'll get a histogram like the one shown in Figure 18-2.

Issuing the ANALYZE command

To actually get Oracle to generate all these statistics and histograms, you need to issue an ANALYZE command. Using the ANALYZE command, you can generate statistics for a table, its indexes, and its columns.

The ANALYZE command affects only one table at a time, which can be a bit inconvenient if you want to generate statistics for all the tables in a schema or for all the tables in your database. One approach to this problem is to generate a SQL*Plus script that does the job and to just invoke that script when you need it. Another approach is to use the DBMS_UTILITY package, which contains a procedure that allows you to analyze an entire schema with one command.

If you're using the cost-based optimizer, analyze your tables regularly. How frequently you do this depends on how fast your data changes. You may find, if you have extremely volatile tables, that it makes sense to analyze them more frequently than the others.

Analyzing a Table

When you analyze a table, you can choose between computing statistics exactly and estimating statistics based on a sampling of the rows in the table. Which approach you take depends on how much time you have and how large your table is. When you compute statistics exactly, Oracle is forced to read all the data in the table. With a large table, that can take quite some time. Even just a few minutes per table adds up if you have a lot of tables to go through. You can reduce the amount of time by choosing to estimate.

To analyze a table and all its indexes, issue an ANALYZE TABLE command, as shown in the following example:

```
ANALYZE TABLE tablename COMPUTE STATISTICS;
```

This command uses the COMPUTE keyword, which specifies to compute statistics exactly. To estimate statistics, use the ESTIMATE keyword, optionally followed by a SAMPLE clause, to indicate how large a sample to use. The following example shows statistics being generated for the AQUATIC_ANIMAL table based on a 5 percent sample:

```
SQL> ANALYZE TABLE aquatic_animal
  2  ESTIMATE STATISTICS SAMPLE 5 PERCENT;

Table analyzed.
```

As you can see, you won't get much feedback from this command. Oracle simply reads through the specified percentage of the table and the specified percentage of each index, and computes the statistics that you asked for.

You can use a FOR clause to control the scope of the statistics generation. By default, when you use ANALYZE TABLE, Oracle generates statistics for the table and for all indexes on the table. The FOR clause controls that behavior. The

following two commands limit the analysis to just the table, and just the indexes, respectively:

```
ANALYZE TABLE aquatic_animal
  COMPUTE STATISTICS FOR TABLE;

ANALYZE TABLE aquatic_animal
  COMPUTE STATISTICS FOR ALL INDEXES;
```

The FOR clause also allows you to generate statistics for individual columns within the table.

Analyzing Columns and Generating Histograms

If you want to generate column statistics for a table, make sure that you've generated statistics for the table first. Any time that you generate table statistics, you wipe out any column statistics. The following example shows how you might issue two commands, one after the other, to generate statistics for a table and its indexed columns:

```
SQL> ANALYZE TABLE aquatic_animal COMPUTE STATISTICS;

Table analyzed.

SQL> ANALYZE TABLE aquatic_animal
  2  COMPUTE STATISTICS FOR ALL INDEXED COLUMNS;

Table analyzed.
```

In this example, statistics are generated only for the indexed columns. You also have the options of generating statistics for all columns or for a specific list of columns.

It only makes sense to generate statistics for columns that are used in the WHERE clause of a query.

When you generate column statistics, Oracle creates histograms showing the distribution of data within those columns. By default, Oracle will use up to 75 buckets for each histogram. You can use the SIZE clause to specify a different limit, which can be anywhere from 1 to 254. The command in the following example specifies an upper limit of ten buckets and analyzes only the TANK_NO column:

```
SQL> ANALYZE TABLE aquatic_animal
  2  COMPUTE STATISTICS FOR COLUMNS tank_no SIZE 10;

Table analyzed.
```

Specifying a size of 10 doesn't mean that you will necessarily get ten buckets. Oracle may choose to use less depending on the number of rows in the table and the number of distinct values in the column being analyzed.

After analyzing a column, you can query the ALL_TAB_HISTOGRAMS view to see the histograms that Oracle generated. Consider this example:

```
SQL> SELECT endpoint_number,
  2         endpoint_value
  3  FROM all_tab_histograms
  4  WHERE owner = 'SEAPARK'
  5  AND table_name = 'AQUATIC_ANIMAL'
  6  AND column_name = 'TANK_NO'
  7  ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              3              1
              7              2
             10              3
```

In this example, there are only three distinct values and ten buckets, so Oracle creates a bucket for each value. You can see this because the first bucket has an endpoint greater than zero. The ENDPOINT_VALUE indicates the value represented in the bucket. The ENDPOINT_NUMBER indicates the cumulative number of rows:

- Three rows have a value of 1 for the tank number.
- Four rows (7–3) have a value of 2 for the tank number.
- Three rows (10–7) have a value of 3 for the tank number.

If you were to create a two-bucket histogram using SIZE 2, the results would look like the following:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              1
              1              2
              2              3
```

In this case, since the number of buckets is less than the number of distinct values in the column (two buckets is less than three distinct values), Oracle simply places five rows in each bucket. The following statements are true about this scenario:

- The bucket with an endpoint number of 0 is a special bucket that contains zero rows, and instead tells us the lowest value in the column. In this case, we see that the lowest tank number is 1.

- ✦ The endpoint number is used only to place the buckets in order and has no bearing on the cumulative number of rows in each bucket.
- ✦ Since there are ten rows in the table, buckets 1 and 2 have five rows each.
- ✦ There are five rows with tank numbers from 1 through 2.
- ✦ There are five rows with tank numbers from 2 through 3.

With this latter type of histogram, Oracle won't record duplicate buckets. You might, for example, see results like the following:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              1
              1              2
              4              3
```

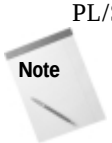
Notice the gap between endpoints number 1 and 4. There are really three buckets containing an endpoint value of 3. Oracle is saving space here. You have to interpret these results as a shortened version of the following:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              1
              1              2
              2              3
              3              3
              4              3
```

Once generated, Oracle can use these statistics to determine how best to execute a query when that query contains references to the `TANK_NO` column in its `WHERE` clause. If you write a query specifying `WHERE TANK_NO BETWEEN 1 AND 3`, the optimizer looks at the histogram and quickly determines that the query returns all rows in the table. Using this information, the optimizer likely chooses to forgo using any indexes and just does a full tablescan to return all the rows.

Analyzing an Entire Schema

If you want to analyze all the tables in a particular schema, you can use the `ANALYZE_SCHEMA` procedure in the `DBMS_UTILITY` package. This is a built-in PL/SQL procedure that Oracle supplies.



The `DBMS_UTILITY` package, together with all other built-in packages, is created when you run the catproc.sql script on a new database.

The syntax for calling DBMS_UTILITY.ANALYZE_SCHEMA is as follows:

```
DBMS_UTILITY.ANALYZE_SCHEMA (
   schema VARCHAR2,
   method VARCHAR2,
   estimate_rows NUMBER,
   estimate_percent NUMBER,
   method_opt VARCHAR2)
```

The following list describes each of the elements in this syntax:

- schema — The name of the schema for which you want to analyze a table.
- method — This is either ESTIMATE, COMPUTE, or DELETE, depending on whether you want to estimate statistics, compute them exactly, or delete them.
- estimate_rows — The number of rows to sample, if you're estimating. Omit this argument if you aren't estimating or if you are estimating based on a percentage.
- estimate_percent — The percentage of rows to sample, if you're estimating. Omit this argument if you aren't estimating or if you are estimating based on a specific number of rows.
- method_opt — This is the FOR clause that you want to use: for example, FOR TABLE, FOR ALL INDEXED COLUMNS, and so on.

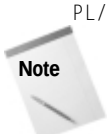
The following example demonstrates how to use DBMS_UTILITY.ANALYZE_SCHEMA to analyze all tables and then all indexed columns owned by the user SEAPARK:

```
SQL> EXECUTE dbms_utility.analyze_schema ('SEAPARK','ESTIMATE',NULL,20,'');

PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_utility.analyze_schema ('SEAPARK','ESTIMATE',NULL,20, -
>             'FOR ALL INDEXED COLUMNS');

PL/SQL procedure successfully completed.
```



The hyphens at the end of each line are SQL*Plus continuation characters. They must be preceded by at least one space.

A similar procedure in the DBMS_UTIL package, named ANALYZE_DATABASE, enables you to analyze all tables in all schemas in the database. The parameters are the same, except that ANALYZE_DATABASE doesn't require a schema name. The ANALYZE_DATABASE procedure takes only the other four parameters. You can read more about these and other procedures in the DBMS_UTILITY package, by reading the *Oracle8i Supplied Packages Reference* manual.

Deleting Statistics

Believe it or not, sometimes you may want to delete statistics that you have generated. For example, you might find that you have generated statistics by mistake. If the optimizer mode is set to CHOOSE, then generating statistics for a table will cause queries against that table to suddenly begin using the cost-based optimizer rather than the rule-based optimizer. This can have a drastic effect on performance.

You can delete statistics for one particular table by issuing the following command:

```
ANALYZE TABLE aquatic_animal DELETE STATISTICS;
```

You can delete the statistics for all tables in a schema by making a call to DBMS_UTILITY.ANALYZE_SCHEMA, as shown in the following example:

```
SQL> EXECUTE dbms_utility.analyze_schema ('SEAPARK', 'DELETE');

PL/SQL procedure successfully completed.
```

Remember, if no statistics are available for any of the tables involved in a query, Oracle will be forced to use the rule-based optimizer.

Discovering Chained Rows

When you analyze a table, Oracle counts the number of chained and migrated rows. You can access that count by selecting the CHAIN_CNT column from DBA_TABLES (or ALL_TABLE or USER_TABLES). Chained and migrated rows can become a significant performance issue if you access them frequently.

Chained rows are rows that are not entirely stored within one database block. When you access a chained row, Oracle reads the block containing the first part of the row, finds a pointer to a second block, reads the second block, picks up more of the row, possibly finds a pointer to a third block, and so on.

Migrated rows are those that have grown to the point where they will no longer fit in the block originally used to store the row. When that happens, Oracle may move the entire row to a new block, leaving a pointer to that new block in the original block.

Chained rows have two causes. If a row in a table is larger than the database block size, then it will never fit in one block and will always be chained. Row chaining can also occur whenever a row grows in size to the point where the size exceeds the free space in any available block assigned to the table. Row migration occurs when rows grow in size and require more space than remains in the block.

Chained and migrated rows present a performance problem because of the extra I/O involved. Instead of reading one block and getting an entire row, Oracle must read at least two blocks. If Oracle needs to read two rows from a block and both rows are chained, then one I/O turns into three I/Os.

Because of the potentially high I/O overhead involved, check for chained rows regularly. The simplest way to do this is to periodically run a report that lists chained row counts for all your tables. Listing 18-1 shows a query that lists the number of chained rows for each table in a user's schema.

Listing 18-1: **Listing chained row counts**

```
SQL> SELECT table_name, chain_cnt
  2  FROM user_tables
  3  ORDER BY chain_cnt DESC;

TABLE_NAME                     CHAIN_CNT
------------------------------ ---------
AQUATIC_ANIMAL                        37
CARETAKER                              0
CHECKUP                                0
CHECKUP_HISTORY                        0
PARK_REVENUE                           0
SEAPARK_STATISTICS                     0
TANK                                   0
```

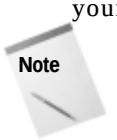
For this query to provide accurate results, you must analyze the tables first. Once you discover chained rows, you can get rid of them in a couple of different ways. Depending on how much data you have in the table and on how much time you have, you can use either of the following methods:

- ✦ Export the table, delete the table, and import the table back again. This approach is the most disruptive, but it gives you a chance to adjust storage parameters on the table.
- ✦ Specify that the `ANALYZE` command lists the chained rows. This allows you to reinsert them into the table.

Chapter 8, "Using Oracle8i's Export Utility," and Chapter 9, "Using Oracle8i's Import Utility," show you how to use the Export and Import utilities, respectively. Refer to those chapters if you aren't already familiar with Export and Import. The rest of this section shows you how to list the chained rows and reinsert them without having to re-create the entire table.

Creating the CHAINED_ROWS table

Before you can use the ANALYZE command to get a list of chained rows in a table, you must create another table to hold that list. Oracle supplies a script named UTLCHAIN1.SQL that you can use for this purpose. You'll find UTLCHAIN1.SQL in your $ORACLE_HOME/rdbms/admin directory.



For Oracle releases prior to 8.1.5 (8i), you must use UTLCHAIN.SQL. You can't use the CHAINED_ROWS table created by UTLCHAIN.SQL to hold a list of chained rows on an index-organized table (IOT).

Listing 18-2 shows you how to use UTLCHAIN1.SQL to create a chained rows table.

Listing 18-2: Creating the CHAINED_ROWS table

```
SQL> @e:\oracle\ora81\rdbms\admin\utlchain1.sql

Table created.

SQL> DESCRIBE CHAINED_ROWS
 Name                      Null?    Type
 ------------------------- -------- -------------
 OWNER_NAME                         VARCHAR2(30)
 TABLE_NAME                         VARCHAR2(30)
 CLUSTER_NAME                       VARCHAR2(30)
 PARTITION_NAME                     VARCHAR2(30)
 SUBPARTITION_NAME                  VARCHAR2(30)
 HEAD_ROWID                         UROWID
 ANALYZE_TIMESTAMP                  DATE
```

You do need the CREATE TABLE system privilege to create the CHAINED_ROWS table, but you don't need any special or unusual privileges besides that. The table name doesn't absolutely need to be CHAINED_ROWS. If you want a different name, you can use the ALTER TABLE command to rename it. You can place the table in any schema that you like, although it will be easiest to create it in the schema that you are analyzing.

Listing the chained rows

Having created a CHAINED_ROWS table, you can use the ANALYZE command to generate a list of chained rows for the table. You do this by adding a LIST CHAINED

ROWS clause to the command. The following example shows this being done for the AQUATIC_ANIMAL table:

```
SQL> ANALYZE TABLE aquatic_animal
  2  LIST CHAINED ROWS INTO chained_rows;

Table analyzed.
```

Now that you've listed the chained rows, you are ready to report on the results and reinsert the rows.

Viewing the results

The CHAINED_ROWS table will end up with one row for each chained row in the table that you analyzed. The most important piece of information contained in the CHAINED_ROWS table is the ROWID identifying the rows that are chained. Query the CHAINED_ROWS table, and you'll get results similar to the following:

```
SQL> SELECT * FROM chained_rows;

OWNER_NAME  TABLE_NAME      HEAD_ROWID           ANALYZE_T
----------- --------------- -------------------- ---------
SEAPARK     AQUATIC_ANIMAL  AAADViAAHAAADQ3AAS   10-SEP-99
SEAPARK     AQUATIC_ANIMAL  AAADViAAHAAADQ3AAT   10-SEP-99
SEAPARK     AQUATIC_ANIMAL  AAADViAAHAAADQ3AAU   10-SEP-99
SEAPARK     AQUATIC_ANIMAL  AAADViAAHAAADQ3AAW   10-SEP-99
SEAPARK     AQUATIC_ANIMAL  AAADViAAHAAADQ3AAX   10-SEP-99
```

As you can see, the schema name and table name are part of each record. That allows one CHAINED_ROWS table to contain information for multiple database tables. The HEAD_ROWID column contains the ROWIDs identifying the chained rows. You can use this information to delete and reinsert these rows, thus eliminating as much chaining as possible.

Reinserting chained rows

If you don't have the time or desire to export and import the entire table, you can still do something to eliminate chained rows. You can delete them and reinsert them by following these steps:

1. Copy the chained rows to a temporary work table.
2. Delete the chained rows from the main table.
3. Select the chained rows from the work table and reinsert them into the main table.

While using this method to eliminate chained rows, users will still be able to access the table, but after step 2, they won't be able to access any of the chained rows until the process is complete. Consider this as you decide when to do this.

Copying Chained Rows to a Work Table

You can use the ROWID in the CHAINED_ROWS table to identify the chained rows in the main table. An easy way to create a work table and populate it at the same time is to issue the CREATE TABLE statement using a subquery attached to it. The following example shows such a statement being used to create a work table to hold the chained rows from the AQUATIC_ANIMAL table:

```
SQL> CREATE TABLE aquatic_animal_chained
  2     AS SELECT *
  3        FROM aquatic_animal
  4        WHERE ROWID IN (
  5           SELECT head_rowid
  6           FROM chained_rows
  7           WHERE table_name = 'AQUATIC_ANIMAL'
  8           AND owner_name = 'SEAPARK'
  9           );

Table created.
```

At this point, the AQUATIC_ANIMAL_CHAINED table contains copies of all the chained rows from the AQUATIC_ANIMAL table.

Deleting Chained Rows from the Main Table

Once you've copied the chained rows into a work table, you need to delete them from the main table. You can use a DELETE statement to do this, together with the same subquery used in the previous CREATE TABLE statement. Consider this example:

```
SQL> DELETE FROM aquatic_animal
  2     WHERE ROWID IN (
  3        SELECT head_rowid
  4           FROM chained_rows
  5           WHERE table_name = 'AQUATIC_ANIMAL'
  6           AND owner_name = 'SEAPARK'
  7           );

37 rows deleted.
```

It's possible that deleting rows like this could lead to foreign key constraint violations. If so, you will need to disable those foreign key constraints using ALTER TABLE DISABLE CONSTRAINT commands, and reenable them after you've reinserted the rows.

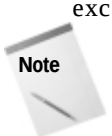
Reinserting the Rows

Reinserting the chained rows is the easy part. All you need to do is select all the rows from the work table and insert them back into the main table, as shown in this example:

```
SQL> INSERT INTO aquatic_animal
  2     SELECT * FROM aquatic_animal_chained;

37 rows created.
SQL> COMMIT;
Commit complete.
```

When you insert rows into a table, Oracle always looks for a block large enough to hold the entire row. So, most of the chained rows should be made whole again. The exceptions will be those rows that are simply too large to fit in a single block.



The only way to deal with rows that are too large to ever fit in one block is to re-create the entire database using a larger block size. This represents a serious amount of work. Think about it before you do it. Make sure that the potential performance gain is worth the effort you will expend. If only a few rows fall into the category of being too large for a block, it may not be worth the time and effort to re-create the database to eliminate the chaining.

After you've reinserted the rows back into the main table and you've committed the transaction, don't forget to reenable any constraints that you might have dropped to delete those rows. You may also want to drop the work table, delete the rows from the `CHAINED_ROWS` table, and reanalyze the main table to see the improvement that you made.

Importing and Exporting Statistics

Beginning with the 8i release, Oracle now contains support for exporting and importing statistics. You can use this feature to maintain several sets of statistics within one database, switching back and forth between them at will, or you can even use it to copy statistics between databases.

Why would you want to save and restore statistics? This feature is useful for several reasons:

- Before you reanalyze statistics for a schema, you can save the current statistics as a backup. If, after analyzing the tables to bring the statistics up to date, the optimizer starts generating wildly inefficient execution plans, you can ditch the new statistics and restore the previous set.

- ✦ If you have a good test environment, you may be able to generate a set of statistics there that results in efficient query execution plans from the optimizer. You can then transfer those statistics to your production database, ensuring that the same execution plans are generated there as well.
- ✦ If you are moving data from one database to another, you can move the statistics as well. This saves you the time and effort of reanalyzing the data after you've moved it.

Being able to save and restore statistics is convenient because the cost-based optimizer is a two-edged sword. It's great that it can adjust to changes in your data, but sometimes, those adjustments can lead to surprise performance problems. The ability to test new statistics before you apply them can help eliminate those surprises.

Using the DBMS_STATS package

The built-in `DBMS_STATS` package is the key to exploiting the new features that allow you to save and restore statistics and to transfer them between databases. The `DBMS_STAT` package contains procedures that perform the following functions:

- ✦ Export statistics from the data dictionary to a table
- ✦ Import statistics into the data dictionary from a table
- ✦ Gather new statistics, storing them in either a table or in the data dictionary
- ✦ Arbitrarily set values for specified statistics

To use `DBMS_STATS`, you must have been granted `EXECUTE` access on the package. The SYS user owns the package, so you must log on as SYS to grant yourself access to it.

Exporting statistics

Exporting statistics doesn't have quite the same meaning as when you use the Export utility to export data. The Export utility writes data from your tables to a flat file. When you export statistics, Oracle reads them from the data dictionary and writes them to a database table. You need the following to export statistics:

- ✦ A table in which to place the exported statistics
- ✦ A name for use in identifying the set of statistics created by a particular export

The table that is used to hold the exported statistics is referred to as the `statistics table`. A procedure in the `DBMS_STATS` package can create this

table for you. As for the name you use to identify the statistics, it can be anything that you dream up. You can have several sets of exported statistics, and the name is used to differentiate between them.

Creating the Statistics Export Table

The DBMS_STATS.CREATE_STAT_TABLE procedure will create a statistics table. The procedure header appears as follows:

```
DBMS_STATS.CREATE_STAT_TABLE (
ownname IN VARCHAR2,
stattab IN VARCHAR2,
   tblspace IN VARCHAR2)
```

The following list describes the elements in this example:

- ownname — Identifies the schema in which you want the table to be created.
- stattab — The name that you want to give the table.
- tblspace — The tablespace in which you want the table placed. This is an optional argument. If omitted, the default tablespace is used.

You can create as many statistics tables as you like. Each table may also have as many sets of statistics as you want. The following example shows you how to create a statistics table named SEAPARK_STATISTICS in the SEAPARK schema:

```
SQL> EXECUTE DBMS_STATS.CREATE_STAT_TABLE ( -
>     'SEAPARK','SEAPARK_STATISTICS', 'USERS');

PL/SQL procedure successfully completed.
```

Once you have created a statistics table, you are ready to export some statistics.

Exporting Statistics from the Data Dictionary

You can export statistics for a particular schema, a specific table or index, or for the entire database. The DBMS_STATS.EXPORT_SCHEMA_STATS procedure exports statistics for an entire schema. The syntax appears as follows:

```
DBMS_STATS.EXPORT_SCHEMA_STATS (
   ownname IN VARCHAR2,
   stattab IN VARCHAR2,
   statid IN VARCHAR2,
   statown IN VARCHAR2)
```

The following list describes the elements in this example:

- ownname — The name of the schema for which you want to export statistics.

- ✦ stattab — The name of the statistics table.
- ✦ statid — A name that you want to use to identify this particular export.
- ✦ statown — The name of the schema containing the statistics table. If this is omitted, the table is assumed to exist in the same schema as you are exporting.

To actually export the statistics for a schema and place them into a statistics table, issue a command like that shown in the following example:

```
SQL> EXECUTE DBMS_STATS.EXPORT_SCHEMA_STATS ( -
>    'SEAPARK','SEAPARK_STATISTICS', -
>    'SEAPARK_990909', 'SEAPARK');

PL/SQL procedure successfully completed.
```

The command in this example exports statistics for all tables, indexes, and columns in the SEAPARK schema, placing them into a table named SEAPARK_STATISTICS. The entire set of statistics generated by this export can be identified using the name SEAPARK_990909.

With the statistics in a table like this, you now have the option of restoring them sometime later. You also have the option of exporting the table and transferring the statistics to another database.

Transferring statistics to another database

You can move statistics from one database to another by following these steps:

1. Export the statistics to a table in the source database.
2. Use the Export utility to export the statistics table to a flat file.
3. Use the Import utility to import the statistics table into the target database.
4. Make a call to DBMS_STATS to import the statistics from the table into the target database's data dictionary.

You've already seen that you can use the DBMS_STATS.EXPORT_SCHEMA_STATS procedure to accomplish step 1. In the next section, you see how to use DBMS_STATS.IMPORT_SCHEMA_STATS to accomplish step 4. You accomplish steps 2 and 3 by using the standard Export and Import utilities. Listing 18-3 shows the SEAPARK statistics being transferred from one database to another.

Listing 18-3: **Transferring statistics between databases**

```
$exp seapark/seapark@jonathan file=seapark_stats log=seapark_stats_exp tables=se
apark_statistics

Export: Release 8.1.5.0.0 - Production on Thu Sep 9 19:37:17 1999

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table             SEAPARK_STATISTICS         30 rows exported
Export terminated successfully without warnings.

$imp seapark/seapark@coin file=seapark_stats log=seapark_stats_imp tables=seapar
k_statistics ignore=y

Import: Release 8.1.5.0.0 - Production on Thu Sep 9 19:37:32 1999

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production

Export file created by EXPORT:V08.01.05 via conventional path
import done in WE8ISO8859P1 character set and US7ASCII NCHAR character set
import server uses US7ASCII character set (possible charset conversion)
export server uses WE8ISO8859P1 NCHAR character set (possible ncharset conversio
n)
. importing SEAPARK's objects into SEAPARK
. . importing table           "SEAPARK_STATISTICS"         30 rows imported
Import terminated successfully without warnings.
```

In the Listing 18-2 example, the import was done using the `ignore=y` setting because the `SEAPARK_STATISTICS` table already exists in the target database.

Importing statistics

Importing statistics works in reverse from exporting them. You make a call to DBMS_STATS, and the statistics that you request are read from the statistics table and loaded into the data dictionary. The DBMS_STATS.IMPORT_SCHEMA_STATS procedure is used to import statistics for a schema, and the syntax is as follows:

```
DBMS_STATS.IMPORT_SCHEMA_STATS
   ownname IN VARCHAR2,
   stattab IN VARCHAR2,
   statid IN VARCHAR2,
   statown IN VARCHAR2)
```

The following list describes the elements of this syntax:

- ownname — The schema for which you want to import statistics.
- stattab — The name of the statistics table.
- statid — The name identifying the set of statistics that you want to import.
- statown — The name of the schema containing the statistics table. If this is omitted, the table is assumed to exist in the same schema as the one you are importing.

The following example shows the set of statistics named SEAPARK_990909 being imported for the SEAPARK schema:

```
SQL> EXECUTE DBMS_STATS.IMPORT_SCHEMA_STATS ( -
>    'SEAPARK','SEAPARK_STATISTICS', -
>    'SEAPARK_990909', 'SEAPARK');

PL/SQL procedure successfully completed.
```

The command in this example replaces any existing statistics for the SEAPARK schema with ones from the set named SEAPARK_990909.

Using other DBMS_STATS procedures

A number of other useful procedures exist in the DBMS_STATS package than just the few that you've seen here. The DBMS_STATS package contains functionality that allows you to do the following:

- Import and export statistics for an entire database
- Import and export statistics for a schema
- Import and export statistics for individual tables and indexes

- ✦ Arbitrarily make up your own statistics for an object
- ✦ Create your own histograms

You can find more information about the DBMS_STATS package in the *Oracle8i Supplied Packages Reference*.

Summary

In this chapter, you learned:

- ✦ Oracle supports two optimizers: one rule-based and one cost-based. The rule-based optimizer chooses execution plans based largely on the manner in which you structure the SQL statements that you write. The cost-based optimizer, on the other hand, bases its decisions on the amount of I/O and CPU resources required to execute any given query. You are strongly encouraged to use the cost-based optimizer. It supports more of Oracle's newer functionality, such as index-organized tables, partitioning, and so forth.
- ✦ You can use the ALTER SESSION command to change the optimizer default for a session. Use optimizer hints to change the setting for one statement.
- ✦ The cost-based optimizer requires statistics about your data to make any type of intelligent decision about how to retrieve it. Therefore, if you are using the cost-based optimizer, analyze your tables regularly. In fact, if you have no statistics at all on tables being queried, Oracle will be forced to use the rule-based optimizer regardless of which optimizer you choose.
- ✦ You can use the ANALYZE command to generate optimizer statistics for tables and indexes. However, if you want to analyze all the tables in a schema or in the entire database, you may find it easier to use procedures in the DBMS_UTILITY package. You can use the DBMS_UTILITY.ANALYZE_SCHEMA procedure to analyze all objects in a schema, and you can use DBMS_UTILITY.ANALYZE_DATABASE to analyze all objects in a database.
- ✦ Chained rows represent those rows with data spread over more than one database block. Chaining and migration entail significant I/O overhead. Check periodically for the presence of chained rows, and if the number becomes great enough to impact performance, take steps to eliminate them.
- ✦ Oracle8i contains new functionality that allows you to export statistics from the data dictionary into a database table. This allows you to save statistics before analyzing your objects, and it also allows you to restore those old statistics in the event that the new ones result in poorly performing execution plans. The DBMS_STATS package contains the procedures used to import and export statistics.

✦ ✦ ✦